A Feature-Oriented Modelling Language and a Feature-Interaction Taxonomy for Product-Line Requirements

by

Pourya Shaker

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Many organizations specialize in the development of *families* of software systems, called software product lines (SPLs), for one or more domains (e.g., automotive, telephony, health care). SPLs are commonly developed as a shared set of assets representing the common and variable aspects of an SPL, and individual products are constructed by assembling the right combinations of assets. The feature-oriented software development (FOSD) paradigm advocates the use of system *features* as the primary unit of commonality and variability among the products of an SPL [2]. A feature represents a "coherent and identifiable bundle of system functionality" [89], such as *call waiting* in telephony and *cruise control* in an automobile. Furthermore, FOSD advocates feature-oriented artifacts (FOAs); that is, software-development artifacts that explicate features, so that a clear mapping is established between a feature and its representation in different artifacts. The thesis first identifies the problem of developing a suitable language for expressing feature-oriented models of the functional requirements of an SPL, and then presents the feature-oriented requirements modelling language (FORML) as a solution to this problem. FORML's notation is based on standard software-engineering notations (e.g., UML class and state-machine models, feature models) to ease adoption by practitioners, and has a precise syntax and semantics to enable analysis.

The novelty of FORML is in adding feature-orientation to state-of-the-art requirements modelling approaches (e.g., KAOS [91]), and in the systematic treatment of modelling evolutions of an SPL via *enhancements* to existing features. An existing feature can be enhanced by extending or modifying its requirements. Enhancements that modify a feature's requirements are called *intended feature interactions*. For example, the *call waiting* feature in telephony intentionally overrides the *basic call service* feature's treatment of incoming calls when the subscriber is already involved in a call. FORML prescribes different constructs for specifying different types of enhancements in state-machine models of requirements. Furthermore, unlike some prominent approaches (e.g., AHEAD [11], DFC [94]), FORML's constructs for modelling intended feature interactions do not depend on the order in which features are composed; this can lead to savings in analysis costs, since only one rather than (possibly) multiple composition orders need to be analyzed.

A well-known challenge in FOSD is managing *feature interactions*, which, informally defined, are ways in which different features can influence one another in defining the overall properties and behaviours of their combination [93]. Some feature interactions are intended, as described above, while other feature interactions are *unintended*: for example, the *cruise control* and *anti-lock braking system* features of an automobile may have incompatible affects on the automobile's acceleration, which would make their combination inconsistent.

Unintended feature interactions should be detected and resolved. To detect unintended interactions in models of feature behaviour, we must first define a *taxonomy* of feature interactions for the modelling language: that is, we must understand the different ways that feature interactions can manifest among features expressed in the language. The thesis presents a taxonomy of feature interactions for FORML, which is an adaptation of existing taxonomies for operational models of feature behaviour.

The novelty of the proposed taxonomy is that it presents a formal definition of behaviour modification; and it enables feature-interaction analyses that report only unintended interactions, by excluding interactions caused by FORML's constructs for modelling intended feature interactions.

Acknowledgements

I cannot overstate my gratitude to my Ph.D. supervisor, Professor Joanne M. Atlee. With exemplary integrity and professionalism, and always with a smile, she has generously supported me throughout my Ph.D. with great ideas, great advice, constant encouragement, and many opportunities for professional development. I have learned a great deal from her, both professionally and personally, and feel very fortunate to have worked with her.

I am grateful to Professor Nancy A. Day, Professor Krzysztof Czarnecki, and Professor Derek Rayside for being on my Ph.D. committee. Their insightful feedback throughout my Ph.D. has been crucial for the development of this work. I thank Professor Stefania Gnesi for being the external examiner of my Ph.D. committee. Her valuable comments have provided a fresh perspective for improving and continuing this work.

I thank Dr. Shige Wang and Dr. S. Ramesh, our collaborators from General Motors, for their valuable feedback and for providing us with the knowledge and data needed to apply our approach to the automotive domain.

I thank all of my friends in Waterloo for the good times. The Watform research lab has been a perfect working environment for me and for this I have my Watform colleagues, past and present, to thank. Special thanks goes to my good friends and officemates Shahram Esmaeilsabzali and Vajih Montaghami, and to my friends and collaborators Sandy Beidu, David Dietrich, and Ana Krulec. I thank the wonderful staff of the Cheriton School of Computer Science for always being so helpful. I especially thank Margaret Towell and Wendy Rush for their constant support.

To my aunt, uncle, and cousins in Waterloo and Toronto, I thank you for your love and support.

To Avin, I thank you for your encouragement, patience, and love, which helped me get through the hard times and complete this work.

My greatest appreciation goes to my family. Your unconditional love and unwaivering support fills me with energy and hope. I dedicate this thesis to you.

To Mom, Dad, Ghazaleh, Ali

Table of Contents

List of Tables

List of Figures

Acronyms

AHEAD algebraic hierarchical equations for application design. 22–26, 30, 32, 34, 36, 208

AOP aspect-oriented programming. 25, 26

BCS basic call service. xvi, xviii, 26–30, 33, 35, 36, 39, 46, 47, 51, 52, 57, 74, 75, 81, 87, 93, 95–97, 101, 102, 115–118, 123, 162, 163, 166, 168, 176, 186, 192, 199, 212, 213

BDS basic driving service. xix, 39, 49, 164–166, 168, 185, 186, 235

BSML big-step modelling language. 85

CBM composed behaviour model. 137–139, 146, 147, 154, 160, 161, 176–178, 196, 200, 201, 208

CC cruise control. xix, 39, 49, 118, 164–166, 168, 179, 186, 236

CD caller delivery. xvi, xviii, 39, 45, 101–103, 115, 117, 118, 123–125, 163, 215, 216

CDB caller delivery blocking. xvii, xviii, 39, 45, 101–103, 115, 118, 123–125, 163, 166, 172, 216, 217

CFB call forwarding on busy. xviii, 163, 166, 172, 173, 179, 199, 217, 218

CT call transfer. xviii, 163, 170, 171, 219, 220

CTL computation tree logic. 15

CVL common variability language. 31

CW call waiting. xvi, xviii, 39, 96, 101–103, 115–117, 121, 123, 125, 163, 164, 166, 168, 176, 177, 214, 215

DFC distributed feature composition. 27, 28, 34, 36, 208

DMS driver monitoring system. xix, 165, 169, 171, 250

DOP delta-oriented programming. 30, 31, 35

EMTS extended modal transition system. 33

ER entity-relationship. 14, 15, 19, 20, 34, 40

FCA forward collision alert. xix, 164, 165, 239, 240

FOA feature-oriented artifact. iii, 21, 33–37, 175, 202

FODA feature-oriented domain analysis. 33, 34

FOL first-order logic. 14

FOP feature-oriented programming. 22, 23, 25, 34, 36

FORML feature-oriented requirements modelling language. iii, iv, ix–xi, xiv, xv, 7–11, 38, 40, 44, 46, 48, 51, 61, 62, 72–76, 79, 84, 85, 96, 98, 100–102, 111, 124, 126, 133, 136, 137, 157–166, 168, 171–175, 184, 187, 191, 195, 196, 199, 200, 202–209, 211, 234

FOSD feature-oriented software development. iii, 1–3, 8, 9, 11, 24, 133, 136, 174, 205, 206

FST feature structure tree. x, xiv, xvii, 24, 125–137, 159, 173, 205, 209

GEMTS generalized extended modal transition system. 33

GR group ringing. xviii, 163, 170, 171, 223, 224

HC headway control. xix, 39, 49, 164–166, 168–170, 179, 237, 238

HP headway personalization. xix, 164, 241, 242

LCA lane change alert. xix, 164, 165, 169, 170, 184–186, 238, 239

LCC lane centring control. xix, 164, 186, 244, 245

LSM layered state machines. 27, 28, 34, 36, 208

LTL linear temporal logic. 15

LXC lane change control. xix, 118, 164, 165, 184–186, 246, 247

MTS modal transition system. 33

OCL object constraint language. 14–16, 18

RBF ringback when free. xviii, 163, 225, 226

RC reverse charging. xix, 118, 163, 232, 233

RCA road change alert. xix, 165, 248, 249

RE requirements engineering. 2, 3, 7, 11–13, 18, 20, 34, 158

RSEB reuse-driven software-engineering business. 33

SB split billing. xix, 163, 233, 234

SLC speed limit control. xix, 164, 169, 171, 179, 243

SPL software product line. iii, xvii, 1–9, 11, 12, 21, 33–35, 38–40, 45–50, 54, 57, 73, 84, 101, 119–121, 123, 124, 137, 157, 158, 160–162, 168, 172, 173, 196, 201, 204–206, 211

TCS terminating-call screening. xviii, 163, 186, 192, 228, 229

TL teen line. xviii, 39, 46, 163, 179, 227, 228

TWC three-way calling. xviii, 163, 176, 177, 221, 222

UML unified modelling language. 2, 3, 7, 8, 14, 15, 17–20, 24, 33, 40, 42–44, 46, 51, 75, 76, 158, 204, 205

VM voice mail. xviii, 39, 46, 47, 51–53, 163, 170, 171, 186, 230, 231

WCA world-change action. xvii, 81–84, 87, 94, 95, 100, 102, 133, 136, 141, 144–146, 150, 158, 162, 170, 171, 193

WCE world-change event. 79, 80, 100, 143

Chapter 1

Introduction

1.1 Background

Software product lines (SPLs): Many organizations specialize in developing software systems for a few domains (e.g., automotive, telephony, health care). In such organizations, each newly developed software system is often a variant of an already developed system in the same domain; this results in a *family* of similar systems, called a software product line (SPL), for each domain [36]. A common approach to SPL development advocates exploiting the commonality among the members of an SPL (i.e., *products*), so as to improve the quality and productivity of the development process. In this approach, an SPL is developed as a shared set of assets representing the common and variable aspects of the SPL, and individual products are constructed by assembling different combinations of assets.

Feature-oriented software development (FOSD): The FOSD paradigm advocates the use of system *features* as the primary criterion to identify separate concerns when developing a single software system or an SPL [2]. A feature represents a "coherent and identifiable bundle of system functionality" [89][1]. Some prominent examples of features are the *call waiting*, *call forwarding*, and *voice mail* features of a telephone service; and the *cruise control*, *anti-lock braking system*, and *four-wheel drive* features of an automobile.

In FOSD, features are first-class entities throughout the software life cycle. Hence, features can be used as a shared vocabulary among a diverse group of stakeholders (e.g.,

[1] A feature can also represent a nonfunctional property of a system or SPL. However, such features are not considered in this thesis.

users, customers, developers) in all life-cycle phases. Two main aspects of FOSD are of interest in this thesis: (1) FOSD aims to explicate features within software-development artifacts, so that a clear mapping is established between a feature and its representation artifacts. (2) In applying FOSD to SPL development, features are the primary unit of commonality and variability among the products of an SPL; that is, a product is assembled from a set of common features and a selection of variable features, which together constitute the product's *feature configuration*.

Feature interactions: A well-known challenge in FOSD is managing *feature interactions*, which, informally defined, are ways in which different features can influence one another in determining the overall properties and behaviours of their combination [93]. Some feature interactions are *intended*: for example, the *call waiting* feature of a telephone service is designed to override the *basic call service* feature's treatment of incoming calls when the subscriber is busy. Some other feature interactions are *unintended*: for example, the *cruise control* and *anti-lock braking system* features of an automobile may have incompatible effects on the automobile's acceleration, which would make their combination inconsistent in the absence of some method of resolution/coordination. Managing feature interactions involves realizing intended interactions, detecting unintended interactions, and resolving those unintended interactions that are undesired [93].

The process of managing feature interactions depends on the language in which the features are described: The language is used to specify intended interactions and possibly resolutions to unintended interactions. Furthermore, to detect unintended interactions, we must first ascertain and define the *taxonomy* of feature interactions for the language: that is, we must understand the different ways that feature interactions can manifest among features expressed in the language.

Requirements modelling: The requirements engineering (RE) phase of the software-development process culminates in a specification of the software requirements. Requirements are often specified in a natural language. Natural languages are very expressive and can be easily understood by stakeholders. However, natural-language specifications can be ambiguous, and are not amenable to automatic analyses for checking properties such as consistency or, in the context of SPL development, detecting unintended feature interactions. *Requirements modelling* approaches have emerged to address these limitations: such approaches adapt and extend existing software-engineering modelling languages (e.g., the unified modelling language (UML) [71]) or propose new modelling languages (e.g., problem diagrams [49]) for the purpose of specifying various types of requirements information. The requirements models produced using such approaches vary in their degree of formality, and hence, the degree to which they can be analyzed.

This thesis addresses two problems concerning the application of FOSD to the RE phase of SPL development: (1) a suitable language for specifying feature-oriented models of SPL requirements, and (2) the taxonomy of feature interactions for such a requirements modelling language. The following gives a more detailed description of each problem, together with descriptions of the limitations of existing work in addressing these problems.

1.2 Feature-Oriented Modelling of Product-Line Requirements

We argue that a feature-oriented language for modelling SPL requirements should have the following properties:

Use of existing standards and best practices: First, any requirements modelling language should be based on Jackson and Zave's widely-accepted reference model for RE [50]. This framework defines a system's requirements as desired properties and behaviours of phenomena in a *world* – comprising the system, as a black box, and its environment – that are to be brought about by the system and possibly other agents in the world (e.g., humans, other systems). Following this notion of requirements, a requirements model should provide a model of the world, in terms of which the requirements are expressed.

Second, the language should adapt and extend standard software-engineering modelling languages (e.g., UML, feature models [54]). This can ease adoption by practitioners, since most practitioners are trained in the use of such languages.

Feature modularity: A feature-oriented requirements model should explicate features by modelling each feature's requirements as a separate *feature module*. The alternative approach for explicating features is to associate elements in an integrated model with the features to which they pertain. Feature modularity enables tracing a feature to the model of its requirements without tool support, and enables the independent development of feature modules: authors of different features need not coordinate access to a shared integrated model, and need not concern themselves with naming conflicts among elements introduced by different features (each feature module specifies a local namespace for the elements within the module).

There are two forms of feature modularity: (1) simply localizing a feature's elements into a feature module, and (2) the classical notion of modularity, in which a feature module hides details of a feature's behaviour behind a well-defined interface. Information hiding (the second approach) is critical when specifying feature behaviours at the implementation level. However, at the requiements level, peering into the behaviours of other features is necessary for understanding and developing feature-interaction requirements. As such, we argue that the language should support the first (weaker) form of feature modularity.

Support for modelling feature enhancements as differences: A feature-oriented SPL is primarily evolved by adding new features. Sometimes, the purpose of a new feature is to *enhance* an existing feature. As an example from the automotive domain, various advanced cruise-control features enhance traditional cruise control with additional criteria for maintaining the vehicle's speed (e.g., distance from the vehicle ahead, the speed limit). It is natural to model such enhancements in terms of *differences* from the features being enhanced. In this approach, the model of the enhanced feature is *reused* as the context for expressing the enhancement(s) of the new feature. In addition to the benefits of reuse, this approach has the benefit of making the task of modelling (at least some features) smaller and more focused on the new feature's essential enhancements – thereby, easing some modelling and evolution tasks.

Explicit modelling of intended feature interactions We advocate modelling intended feature interactions explicitly, so that they are more apparent to the modeller and the reviewer. Explicit models of intended interactions also enable analyses that automatically distinguish between intended and unintended interactions, and report only the unintended cases; otherwise, analysis results need to be inspected manually to distinguish between intended and unintended interactions.

Support for multi-product requirements: Some SPLs have requirements associated with the interaction between multiple products. For example, the requirements of an SPL of telephone services involve interactions between two or more users' telephone services, such as the processing of calls between users or the interactions between users' telephone services and voice-mail services. The language should support the modelling of such multi-product requirements.

Ease of evolution: Ideally, adding a new feature module to a requirements model should prompt little to no changes to the existing feature modules.

Precision: The language should be precise so that the models are amenable to analyses, for example, to detect unintended feature interactions.

Commutative and associative feature composition: With feature modularity, features are modelled separately. However, the modeller will eventually want to visualize and (manually or automatically) analyze *feature combinations* corresponding to products of the SPL. Models of feature combinations are obtained by composing the corresponding feature modules. The composition of feature modules should be commutative and associative, so that the composition result is insensitive to the composition order. Commutative and associative composition can lead to savings in analysis costs because only one, rather than (possibly) many, composition order needs to be analyzed; it can ease evolution, because adding a new feature will not require computing its appropriate position in the composition order (more on this topic in Section 2.2.4).

There is a wealth of existing work on requirements modelling (e.g., [91, 46, 25, 62]) and feature-oriented behaviour modelling (e.g., [44, 94, 11, 1, 42]). However, all such approaches have limitations of varying degrees in satisfying the desired properties given above. For example, there exist requirements modelling languages that are precise and that are based on standards and best practices (e.g., KAOS [91] and SCR [46]), but they are not feature-oriented. On the other hand, existing approaches for feature-oriented behaviour modelling lack support for explicitly modelling intended feature interactions (e.g., DFC [94]) and a non-commutative composition operator for feature modules (e.g., DFC and AHEAD [11]). The non-commutativity of feature composition in existing approaches is typically due to their use of composition order for realizing intended feature interactions as well as automatically resolving unintended interactions that are unknown. A more detailed account of existing work on requirements modelling and feature-oriented behaviour modelling, as well their limitations, is given in Sections 2.1 and 2.2, respectively.

1.3 Feature Interactions in Product-Line Requirements

Feature interactions have been studied for years, and there is extensive work on defining taxonomies of feature interactions. Some taxonomies are informal: for example, Cameron et al. [21] informally define *functional ambiguity* as a type of feature interaction that arises when two features are designed to respond to the same situation in different ways. Other taxonomies provide formal definitions of unintended feature interactions to support analyses for detecting such interactions. Such formal definitions naturally depend on the modelling language used to express the features: for example, functional ambiguity as defined by Cameron et al. can manifest as logical inconsistency in a logical model and as nondeterminism in an operational model. Informal taxonomies tend to be more comprehensive due to their independence from particular modelling languages.

We argued above that a (feature-oriented) modelling language for SPL requirements should be precise, so as to support analyses, including the detection of unintended feature interactions. As such, the taxonomy of feature interactions for such a language should also be precise. Furthermore, we argue that such a taxonomy should have the following properties:

Generalized adaptation of existing taxonomies: The taxonomy should adapt existing taxonomies that are applicable to the modelling language. Specifically, it should be based on taxonomies that were defined for languages of the same paradigm (e.g., operational, declarative) as the requirements modelling language. Furthermore, the taxonomy should aim for brevity by grouping feature-interaction types into more general types where possible.

Distinguishing intended feature interactions: Given a requirements modelling language with constructs for explicitly modelling intended feature interactions, the taxonomy should distinguish and exclude the feature interactions caused by these constructs. Such a taxonomy enables analyses that report only unintended feature interactions.

The majority of existing (formal) taxonomies of feature interactions are associated with modelling languages that do not support the explicit modelling of intended interactions. Also, even though a feature interaction is generally considered to be any scenario in which one feature modifies the behaviour of another feature, existing taxonomies do not include a precisely defined feature-interaction type that corresponds to this *general* case of behaviour modification. Instead, they include various feature-interaction types that are special cases of behaviour modification. There are two main reasons for this decision: (1) the taxonomies are developed for analyzers that detect the special cases, and (2) the special cases tend to be (with high probability) *undesired* interactions (e.g., looping), whereas the general case is perhaps likely to include unintended interactions are are acceptable and that do not need resolutions. A more detailed description of existing feature-interaction taxonomies is given in Section 2.3.

1.4 Thesis Overview

This thesis introduces the feature-oriented requirements modelling language (FORML) for specifying feature-oriented models of SPL requirements, as well as a feature-interaction taxonomy for FORML to support the detection of unintended feature interactions at the requirements level.

Thesis Statement: FORML is a novel language for specifying feature-oriented models of SPL requirements that exhibits the desired properties described in Section 1.2:

- FORML is based on Jackson and Zave's reference model for RE, in that it provides a model of the world, in terms of which the requirements are expressed.
- FORML adapts and extends standard software-engineering modelling languages such as the UML and feature models, to ease adoption by practitioners.
- FORML supports feature modularity to enable the independent development of each feature's requirements.
- FORML supports modelling a new feature's enhancements of existing features, in terms of differences from the existing features.
- FORML supports the explicit modelling of intended feature interactions.
- FORML supports the modelling of requirements of interactions between multiple products.
- The evolution of a FORML model is eased by the fact that adding a new feature module to a FORML model normally prompts few changes to existing feature modules – even if it specifies enhancements (including intended interactions) to existing features.
- FORML has a precise syntax and semantics, which enables analyses, such as the detection of unintended feature interactions.
- The composition of feature modules in FORML is commutative and associative.

FORML is accompanied by a formal taxonomy of feature interactions, which enables the detection of unintended feature interactions at the requirements level. This taxonomy exhibits the desired properties described in Section 1.3:

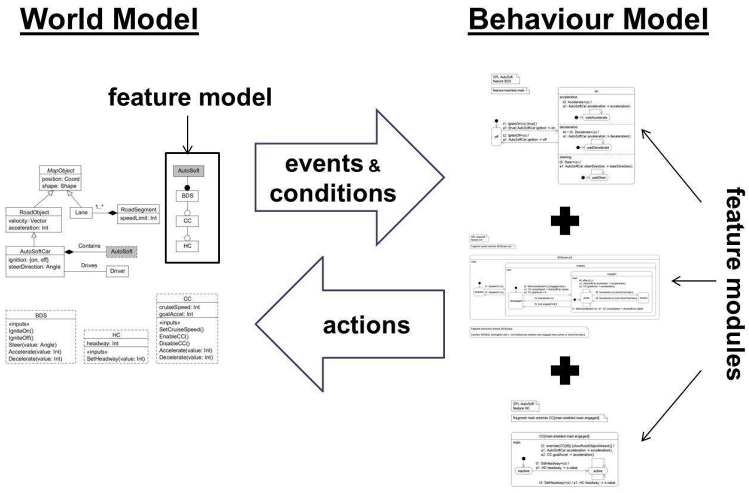


Figure 1.1: Overview of a FORML model (the shrunken models are not intended to be readable)

- The taxonomy adapts existing taxonomies that are applicable to FORML (i.e., taxonomies for operational modelling languages).
- The taxonomy includes a formal definition for behaviour modification. Behaviour modification covers several special cases of feature interactions in existing taxonomies.
- The taxonomy distinguishes and excludes feature interactions caused by FORML's constructs for explicitly modelling intended feature interactions. This enables analyses that only report unintended feature interactions.

1.4.1 FORML

FORML combines and adapts the best practices for requirements modelling with feature modularity techniques from FOSD research, to support the feature-oriented modelling of SPL requirements. A FORML model is decomposed into two views (see Figure 1.1):

- A **world model** is an ontology of concepts that describes a world comprising products of an SPL – as black boxes – and the environment in which they will operate. The concepts in the world model are expressed in the UML class-diagram notation.

- A **behaviour model** is a state-machine model that describes the requirements for an SPL's products. The model's inputs are events and conditions over the world model, and its outputs are actions over the world model. A behaviour model is decomposed into feature modules, which separately describe the requirements for the SPL's features. The syntax for the behaviour model is based on UML state machines.

Contributions: FORML is distinguished from existing requirements modelling and FOSD approaches in the following ways:

- A FORML world model includes a feature-oriented representation of an SPL's products: FORML introduces **SPL** and **feature concepts** to represent products and their feature configurations. Furthermore, a FORML world model includes a **feature model**, which specifies constraints on the valid feature configurations of products.

- FORML provides a systematic treatment of modelling feature enhancements as differences from existing feature requirements. FORML distinguishes between different types of enhancements: enhancements that *add* new requirements in the context of an existing feature's requirements (e.g., call waiting adds requirements for processing a second call, in the context of the call-processing requirements of basic call service), and enhancements that *modify* the requirements of existing features (e.g., call waiting modifies the usual busy-treatment of basic call service, by connecting a second incoming call rather than rejecting it). The latter enhancements correspond to intended feature interactions. Intended interactions are further distinguished based on whether they trigger, prohibit, or override the requirements of existing features; or specify an existing feature's priority over other (new or existing) features. In FORML, an enhancement is modelled as a *state-machine fragment* that *extends* the feature module of the existing feature being enhanced. FORML prescribes different types of fragments for modelling different types of enhancements. In particular, FORML introduces novel constructs for modelling requirements overrides in state-machine models of feature requirements: namely, special transitions that override other transitions in the same or different state machine, and special transition actions that override other actions in the same transition.

- The composition of FORML feature modules is commutative and associative, despite models of intended interactions. As is discussed in Section 2.2, existing FOSD approaches typically sacrifice commutativity for the sake of modelling intended interactions.

Validation: Two case studies have been performed, one from the automotive domain and one from the telephony domain, with the goals of (1) exploring the expressiveness of FORML and (2) evaluating the impact of evolving a FORML model with new features. The second goal is to evaluate the likelihood that a new feature module prompts changes to existing feature modules and to measure the extents of such changes. The automotive case study is adapted from GM Feature Technical Specifications for a set of 11 automotive software features. The telephony case study is adapted from the Second Feature Interaction Contest [59] and comprises 15 telephone-service features.

1.4.2 Feature Interactions in FORML

The proposed taxonomy of feature interactions for FORML is an adaptation of existing taxonomies for operational models of feature behaviour. The taxonomy consists of the following feature interaction types, which are precisely defined to enable analyses for detecting unintended feature interactions in FORML models:

- A **nondeterminism interaction** occurs when there is a nondeterministic choice between a set of concurrently enabled state-machine transitions from different features.
- A **conflict interaction** occurs when a set of features try to perform actions or transitions with inconsistent effects.
- A **modification interaction** occurs when one feature triggers, prohibits, or overrides transitions or actions of another feature in the same or in a different product.

Contributions: The taxonomy of feature interactions for FORML makes the following adaptations to existing taxonomies:

- The notion of a modification interaction generalizes existing feature-interaction types that are special cases of requirements modification. For completeness, the proposed taxonomy also includes feature-interaction types that correspond to two prominent special cases: namely, **deadlock interactions**, which occur when one feature prevents another feature from executing any transitions or actions in the future; and **looping**, which occurs when two features mutually trigger one another's transitions in an infinite loop.

- FORML's constructs for modelling intended feature interactions cause modification interactions. To enable feature-interaction analyses that report only unintended modification interactions, the proposed definition of modification interactions excludes the cases caused by such constructs.

Validation: A set of seven examples have been developed of the different feature-interaction types in the proposed taxonomy: one example of nondeterminism interaction, two examples of conflict interaction, and four examples of modification interaction. Six of the examples are based on the FORML models of the telephony and automotive case studies; the remaining example is pedagogical and was created to illustrate deadlock interaction.

1.5 Chapter Summary

This thesis addresses two problems concerning the application of FOSD to the RE phase of SPL development: (1) a suitable language for specifying feature-oriented models of SPL requirements, and (2) the taxonomy of feature interactions for such a requirements modelling language.

The remainder of this thesis is organized as follows. Chapter 2 describes related work and their limitations in addressing the above problems. Chapter 3 addresses the first problem by introducing the feature-oriented requirements modelling language (FORML) for specifying feature-oriented models of SPL requirements. Chapter 4 addresses the second problem by presenting a feature-interaction taxonomy for FORML. Chapter 5 concludes the thesis and presents possible directions for future work.

An overview of FORML has been published [81] and a description of behaviour modification interactions has been submitted for publication [80].

Chapter 2

Related Work

This chapter presents a summary of existing work in the related areas of requirements modelling (Section 2.1), developing feature-oriented artifacts (Section 2.2), and taxonomies of feature interactions (Section 2.3). The summary of each related area is followed by a discussion of the limitations of existing work in that area in addressing the research problems described in Sections 1.2 and 1.3.

2.1 Requirements Modelling

This section describes uses of different software-engineering modelling languages for describing different types of RE information (Section 2.1.1); followed by approaches that combine different such uses into a single multi-view language for requirements modelling (Section 2.1.2). A discussion of the limitations of existing approaches in modelling SPL requirements in discussed in Section 2.1.3.

2.1.1 Software-Engineering Modelling Languages

Different software-engineering modelling languages support abstractions that are suited to describing different types of RE information. Such languages help alleviate some of the problems raised by the use of natural language. First, they provide structure, which facilitates understanding, navigation, and change. Second, they are semi-formal (i.e., only part of the syntax may be formal and the language may not have a formal semantics) or formal, which enables automated analyses ranging from checking that a referenced item is declared to completeness checks and verification of desired properties. On the downside, software-engineering modelling languages are not as expressive as natural languages, and understanding their models requires more expertise from stakeholders than do natural-language descriptions. To address the latter concern, many such languages are graphical, so as to facilitate communication with stakeholders. The following describes how different software-engineering modelling languages can be used to describe different types of RE information. A meeting-scheduling system adapted from [91] is used as a running example. An objective of this system is to schedule meetings with maximum attendance by invited persons. Towards meeting this objective, the scheduler is required to produce meeting schedules that are convenient for all invited persons; that is, the time of a scheduled meeting should satisfy the availability constraints of invited persons.

Problem Diagrams A *problem diagram* [49] shows a system in the context of its environment, as a set of related *domains*. A domain represents a set of related phenomena: there is a system domain, shown as a rectangle with two vertical stripes, and a set of environment domains, shown as simple rectangles. A relationship between two domains, shown as a line between their rectangles, means that they have *shared phenomena*; i.e., phenomena that belong to both domains. The label of a domain relationship is of the form $D!P$, which denotes that the domain D in the relationship controls the set of shared phenomena P. Finally, a system requirement, shown as a natural-language statement in a dotted oval, is related to the environment domains that include the environment phenomena referred to and constrained by the requirement. Reference and constraint relationships are shown as dotted lines and dotted arrows, respectively, between a requirement oval and a environment-domain rectangle, and are labelled with the corresponding set of phenomena. Figure 2.1 shows a partial problem diagram for the meeting-scheduling system: the *Scheduler*, *MeetingInfo*, and *Invitee* domains represent the scheduler, information about a meeting, and a person invited to a meeting, respectively; and the *reqConstraints*, *constraints*, and *date* phenomena represent a request from the scheduler to an invitee for his or her availability constraints, an invitee's availability constraints, and the date of a scheduled meeting, respectively. The *Scheduler* domain controls the *reqConstraints* and *date* phenomena shared with the *Invitee* and *MeetingInfo* domains, respectively. The scheduler requirement refers to the *constraints* phenomena and constrains the *date* phenomena of the *Invitee* and *MeetingInfo* domains, respectively.

Conceptual Models A *conceptual modelling* language can be used to specify a structural model of phenomena in a world comprising the system and its environment, called

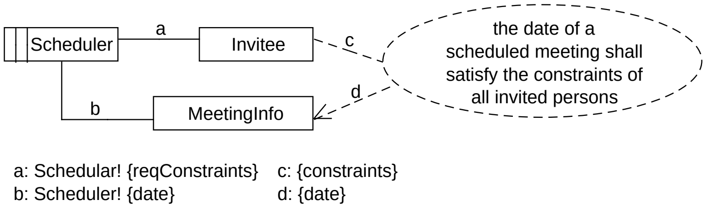


Figure 2.1: Partial problem diagram for a meeting-scheduling system

a *world model*. A conceptual world model consists of a set of *concepts*, each representing a type of *object* in the world. A popular instance of conceptual modelling is entity-relationship (ER) modelling. The core constructs of an ER model [82] are *entities*, *relationships*, and *attributes*. Entities correspond to concepts. Relationships between entities represent classes of relationships between entity instances. Each related entity has a particular role in the relationship. Entities and relationships can characterize their instances by a set of attributes, where an attribute has a name and a range of values (i.e., a *type*). ER models have been adapted into the UML as *class models*, which introduce additional constructs such as generalization relationships between entities, and special relationships such as aggregation and composition. Figure 2.2 shows a partial conceptual model for the meeting-scheduling system, expressed in the UML class-diagram notation: The classes *Person*, *MeetingInfo*, and *Date* represent persons that can be invited to meetings, meeting information, and dates, respectively. Persons are related to the meetings to which they have been invited (modelled as the *Invitation* association) and have availability constraints for each such meeting (modelled as the *Constraint* association class). An availability constraint consists of a set of dates on which a person cannot attend a meeting (modelled as the association between *Constraint* and *Date*). A scheduled meeting will have a date as soon as all of the invitees have determined their constraints (modelled as the association between *Meeting* and *Date*). An instance of a world model is called a *world state*. A world state represents a snapshot of the world; as an example, a world state of an ER world model consists of a particular set of objects, with particular attribute values and relationships. An evolution of the world can be discretely represented by a sequence of world states.

Formal Constraints A formal constraint language can be used to declaratively specify assumptions about the world, as well as the system's requirements. Flavours of classical logics such as first-order logic (FOL) [47] can be used to specify invariants about the world. Examples of such FOL-based languages include the object constraint language

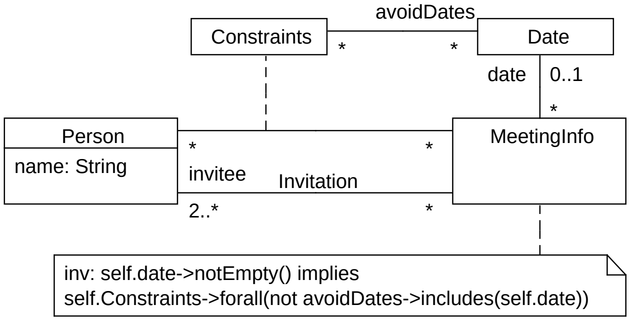

avoidDates
Constraints
Date
*
*
date
0..1
*
Person
name: String
MeetingInfo
*
invitee
Invitation
*
2..*
*
inv: self.date->notEmpty() implies
self.Constraints->forall(not avoidDates->includes(self.date))

Figure 2.2: Partial conceptual model for a meeting-scheduling system

(OCL) [73] and Alloy [48] constraint languages, which are used to specify invariants over UML class models and Alloy conceptual models, respectively. Both languages provide mechanisms for navigating instances of conceptual models. The OCL invariant constraint on the *MeetingInfo* class in Figure 2.2 specifies the following scheduler requirement: the date of a scheduled meeting (`self.date`) is not an element of the set of dates to be avoided as per the constraints of any of the invitees (`self.Constraints->forall(not avoidDates->includes(self.date))`). Also, in ER models (including UML class diagrams), each entity in a relationship can have a *multiplicity*, which is an invariant that constrains the number of instances of the entity that can be related to some set of instances of other entities in the relationship. For specifying constraints on world evolutions, temporal logic variants can be used: linear temporal logic (LTL) and computation tree logic (CTL) can be used to specify constraints over sequences and trees of world states, respectively.

Goal Models A *goal model* [91] is an and/or tree that specifies possible refinements of the system's high-level requirements. The term *goal* refers to a declarative specification of a system requirement. A node of a goal model is either a goal or a declarative specification of some world assumption. Non-leaf nodes correspond to goals pertaining to the system's high-level requirements. Such goals cannot be satisfied by the system alone and need to be further refined. A non-leaf goal is and-refined into sub-goals and world assumptions that should be consistent; taken together, the sub-goals and world assumptions should satisfy the parent goal. A non-leaf goal may have alternative and-refinements. A goal is not further refined (i.e., it is a leaf node) if it is either achievable by the system alone or it is expected to be achieved by some agent in the system's environment. Figure 2.3 shows

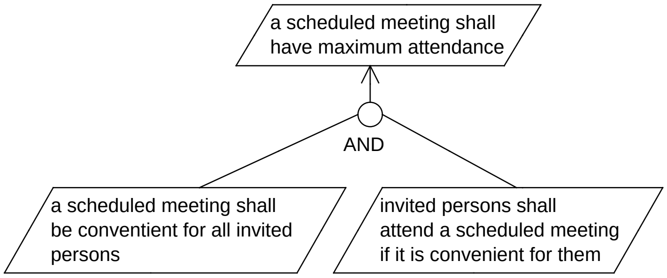

a scheduled meeting shall have maximum attendance
AND
a scheduled meeting shall be conventient for all invited persons
invited persons shall attend a scheduled meeting if it is convenient for them

Figure 2.3: Partial goal diagram for a meeting-scheduling system

a partial goal diagram for the meeting-scheduling system. The diagram shows a possible and-refinement of the system's objective of maximizing attendance for scheduled meetings into two sub-goals: (1) the scheduler requirement that scheduled meetings are convenient for all invitees, and (2) the world assumption that invitees will actually attend a scheduled meeting if it is convenient for them.

Operational Models It may be desirable to specify requirements, and possibly world assumptions, operationally; that is, by explicitly specifying sequences of world states, where each transition from one world state to the next results from changes to the world made by the system or by environment agents. Such changes are called *operations* [25, 62, 91]. An operation can be specified in terms of its name and parameters, and its pre- and post-conditions on world states of a conceptual world model. If the post-condition comprises multiple cases, the cases can be structured, such as in a decision table. To avoid the frame problem [68], an operation's specification may indicate the set of world-state elements that it modifies. Usually, an operation's specification does not include its triggering conditions. The main operation performed by a meeting-scheduling system is to determine the date of a scheduled meeting. This operation, named *determineDate*, is specified below. The meeting information on which the operation is performed is specified as parameter m. The *modifies* clause indicates that the operation will modify only the date of m. The pre- and post-conditions of the operation are specified in a mixture of English and OCL: The precondition specifies that all invitees have determined their constraints for m, and the post-condition specifies that the determined date satisfies all of these constraints (a primed variable denotes an element of the world state after the operation is performed).

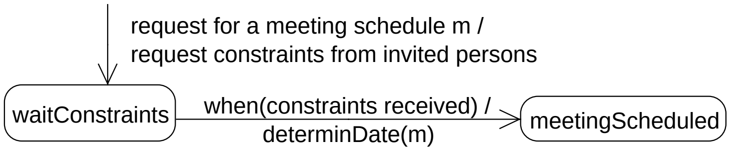

request for a meeting schedule m /
request constraints from invited persons
waitConstraints
when(constraints received) /
determinDate(m)
meetingScheduled

Figure 2.4: State machine fragment for a meeting-scheduling system

```
determineDate(m:MeetingInfo)
   modifies:
      m.date
   precondition:
      constraints received from m's invitees
   postcondition:
      let d = nearest future date such that
         m.Constraints->forall(not excluded->includes(d)) in
      m.date' = d
```

The order and triggering conditions of operation invocations can be expressed using state-machine models that specify all sequences of world-state transitions that result from requirements or world assumptions. A typical state-machine model consists of control states and transitions between control states, where a transition has a triggering event, a guard condition, and a set of actions that are performed when the transition fires. A transition fires when its triggering event occurs provided that its guard condition is satisfied and the state machine is in the transition's source control state. The behaviour of a system or environment agent is defined as a sequence of the model's transitions. When used for requirements modelling, the trigger, guard, and actions of transitions express changes to the world, conditions on the world, and operations on the world, respectively. Figure 2.4 shows a fragment of a state-machine model of the requirements of the meeting-scheduling system, expressed in the UML state-machine notation. The fragment shows the triggering conditions for invoking the *determineDate* operation that is modelled above: the operation is invoked as soon as the invitees of a scheduled meeting determine their constraints for the meeting, following a request for scheduling the meeting.

The SCR language [76] is designed to model system requirements. An SCR model consists of a set of *variables* and a set of *tables*. There are three types of variables: *monitored*, *controlled*, and *mode* variables. Monitored and controlled variables constitute a simple world model: controlled variables can be changed by the system, whereas monitored variables can only be read by the system and are changed by environment agents. An *input event* is a change in the value of a monitored variable, possibly conjoined with a guard condition on current values of monitored variables. Each table is a function that describes how the value of a single variable changes. There are three types of tables: *mode-transition*, *event*, and *condition* tables. A mode-transition table specifies how a mode variable V changes, as a function from the current value of V and an input event, to the next value of V. Changes to the value of a controlled variable V are specified by either an event table, as a function from the current values of mode variables and an input event to the next value of V; or by a condition table, as a function from the next values of mode and monitored variables to the next value of V.

2.1.2 Multi-View Requirements Modelling Languages

Some software-engineering modelling languages support describing more than one type of requirements-related information. For example, the UML (in conjunction with OCL) and Alloy support specifying a world model, as well as invariants and operations over world states. The UML also supports state-machine models of requirements or world assumptions. The RE-specific languages described above each focus on a specific type of requirements information[1]. RE methods have emerged that prescribe how to develop a *requirements model* using a combination of modelling languages. An important consideration in such methods is ensuring traceability and consistency among requirements model's different views, expressed in different languages.

Perhaps the most detailed of such methods is KAOS as presented by van Lamsweerde in [91]. Among software-engineering modelling languages designed for or applied to requirements modelling, the KAOS language comes the closest to satisfying the desired properties described in Section 1.2. For this reason, the following describes some of the distinguishing properties of KAOS models in relative detail.

A KAOS model is composed of the following interrelated models: an *object model*, a *goal model*, an *agent model*, an *operation model*, and a *behaviour model*[2].

[1] Technically, an SCR model has two views: a world model comprising a set of variables, and a model of requirements. However, its world model is simplistic and lacks structure. Also, problem diagrams have a simplistic representation of the world and a natural-language description of requirements. For this reason, SCR and problem diagrams are not presented as multi-view languages.

[2] A KAOS model also includes an *obstacle model* to support risk analysis, where a risk is a condition that can lead to the non-satisfaction of a system objective. Further detail on this model is not provided because risk analysis is beyond the scope of this thesis.

The object model is an ER world model expressed in the UML class-diagram notation. In the object model, concepts are grouped into four types: *entity*, *agent*, *association*, and *event* concepts. An entity object is *persistent* (i.e., it exists for longer than an instant) and has an independent existence. An agent object is like an entity object, except that it can monitor or control other objects; it represents the system or an environment agent. An association concept corresponds to a relationship in an ER model. An event object is transient and represents a change in the world or a world message (e.g., a request or notification). The object model supports multiplicity constraints and invariant constraints on world states.

The goal model is as described in Section 2.1.1, with the following additional properties. The goal model classifies goals as being *behavioural* or *soft*, which is orthogonal to being functional and non-functional goals. The distinction is that it is possible to determine conclusively whether a system satisfies a behavioural goal, but one can only determine the extent to which a system satisfies a soft goal. Behavioural goals that are leaf nodes in the goal model are assigned to agent objects for realization. A goal model also specifies potential *conflicts* between goals; that is, inconsistencies between goals under some conditions.

The agent model augments the object model to specify properties of agent concepts, such as their *capabilities*. A capability of an agent concept A is an attribute or association X of some concept C that instances of A can monitor or control. Each object attribute or association can be controlled by at most one agent instance; this is called the *unique-controller constraint*.

The operation model describes operations performed by agent objects to realize their assigned behavioural goals. An operation is specified in terms of intrinsic pre- and post-conditions, on world states, that hold for all occurrences of the operation. In addition, an operation description may include goal-specific pre- and post-conditions or a trigger condition — either of which further specifies the operation in a specific context. While preconditions specify the *permission* to apply the operation, trigger conditions specify an *obligation* to perform it. When the trigger conditions are false, but the preconditions are true, the operation may or may not be performed, which is a source of nondeterminism. This nondeterminism can be resolved by a *lazy* or *eager* scheme, in which the operation is or is not performed under such conditions, respectively. As a consistency rule, it is required that when an operation's intrinsic precondition is true and at least one of its goal-specific trigger conditions is true, then all of the associated goal-specific preconditions must be true. Therefore, all sequences of world states that result from requirements or world assumptions can be derived from the operation model.

A behaviour model includes a state-machine model of the behaviour of agent instances, which can be derived from the operation model. Specifically, the behaviour of an arbitrary instance of each agent concept A is modelled as a set of concurrent state machines, one for each attribute or association controlled by the instances of A. Each state of a state machine corresponds to a set of values of its corresponding attribute or association. The behaviour model may also include *sequence diagrams* that show sequences of *interactions* between particular agent instances, where an interaction is the synchronous generation and observation of an event by the source and target of the interaction, respectively.

The goal, object, and operation models can be made formal by expressing world-state constraints and operation pre-, post-, and trigger conditions using temporal logic.

Two other RE methods that prescribe multi-view languages for expressing requirements models are the object-oriented analysis method by Coleman et al., called the *Fusion method* [25], and the methodology by Larman for using the UML at the requirements level [62]; however, these two methods are much less detailed than KAOS. Both the Fusion method and Larman's method prescribe an extended ER model of the world, and informal descriptions of system operations over the world model. Fusion formally describes system behaviours as traces of input and output events, where an input event corresponds to the invocation of an operation by an external entity, and an output event corresponds to the response of some system operation that sends a signal/information to external entities in the world model. Larman uses "system sequence diagrams" to specify system behaviours, where the interactions are invocations of system operations by actors. A formalization of Fusion's ER and operation models using the language Z is given in [10].

Finally, Glinz et al. have developed the ADORA language for modelling system requirements and architecture [38]. An ADORA model is an integration of several views: a *context view*, which specifies environment agents that interact with the system; a *base view*, which is a hierarchical organization of system and environment objects; a *user view*, which specifies use cases associated with objects; a *structural view*, which specifies objects associations, the information flow between objects, and relationships between environment agents and use cases; a *behaviour view*, which specifies object behaviours; and a *functional view*, which specifies object operations as pre- and post-conditions. ADORA models can be semi-formal or formal, depending on whether their textual components are expressed in a natural language or in a declarative formal language (e.g., predicate logic).

2.1.3 Discussion

Existing requirements modelling languages are not feature-oriented, although the KAOS language comes close: KAOS supports abstractions for structuring system behaviours and means for representing variability, other than features. The behaviour model in a KAOS model structures system behaviours in terms of agents. One may consider decomposing a system into sub-agents, and using the sub-agents to represent features. However, because no world-state element can be controlled by more than one sub-agent instance (due to the unique-controller constraint), the sub-agents can only represent features with no shared context. This restriction is too strong for many domains, including the telephony and automotive domains. Rather than supporting feature-oriented variability, KAOS models support variation mechanisms including alternative goal refinements and alternative assignments of goals to agents. In [91], it is explained how such variability mechanisms can be used to model SPLs: each alternative is annotated with the name of the family member that it pertains to.

2.2 Feature-Oriented Artifacts

This section describes existing approaches for developing feature-oriented artifacts (FOAs); that is, approaches for making features explicit in software-development artifacts. Given a feature selection that specifies a product of an SPL, an FOA can be *statically* or *dynamically* configured into a view that reflects the product. In *static configuration*, a model transformation is applied to the FOA, generating an artifact that represents the selected family member. In *dynamic configuration*, run-time mechanisms embedded in the (behavioural) FOA restrict its behaviours to those of the selected family member.

Existing FOA approaches are grouped below into three categories: *compositional* (Section 2.2.1), *structural-annotation* (Section 2.2.2), and *variation-mechanism* (Section 2.2.3) approaches. A discussion of the limitations of existing approaches in developing feature-oriented requirements models is given in Section 2.2.4.

2.2.1 Compositional Approaches

In most compositional approaches, each feature is specified separately in a *feature module*. These approaches are described in Sections 2.2.1.1 to 2.2.1.4. In an alternative compositional approach, each module can specify combinations of features: given a feature selection, all of the modules that specify a subset of the selected features are statically composed. This approach is described in Section 2.2.1.5.

2.2.1.1 Feature-Oriented Programming

Early work in this category is feature-oriented programming (FOP) [78]. FOP is an extension of object-oriented programming: rather than specifying fixed object types, a family of related objects is specified in terms of a set of object features. An object is constructed by an ordered composition of its desired features; for example, `new` $\mathtt{F_n(F_{n-1}...(F_1)...)}$ creates an object by first adding feature F_1, then F_2, up to F_n. If feature A overrides some behaviour of feature B then, when the features are selected together, B's behaviour should be composed before that of A's. In [78], Prehofer presents an extension of Java that provides two new constructs: *feature* and *lifter modules*. A feature module is a mixin that specifies the core behaviour of the feature as a set of attributes and methods. A lifter module labelled "A lifts B" specifies how A overrides B's behaviour, when they are selected together, as a set of overrides of B's feature-module methods; not invoking *super* in such a method override represents the suppression of B's behaviour by A. The semantics of these extensions is given by two translations into Java: *inheritance-based* and *delegation-based*. These translations can be viewed as static-configuration techniques. Both translations map each feature selection (i.e., an ordered composition of features) to a single Java class. In the inheritance-based translation, the class representing the selection $F_n(F_{n-1}...(F_i)...)$ encapsulates the core behaviour of F_n (specified in its feature module), as well as its behaviour overrides of any feature F_j, $n-1 \leq j \leq i$ (specified in the lifter modules "F_n lifts F_j"). These classes are arranged into inheritance hierarchies, such that a class representing $F_n(F_{n-1}...(F_i)...)$ inherits from the class representing $F_{n-1}(F_{n-2}...(F_i)...)$. In delegation-based translation, the core behaviour of each feature is encapsulated in a separate class. The class representing a feature selection collects the methods of the selected feature modules and has a pointer to an instance of each selected feature's class. The implementation of each collected method that is not overridden in the selection simply delegates to the feature class's method. The implementation of an overridden method is a nesting of the implementations of the corresponding lifters, which may nest a call to the corresponding feature class's method.

2.2.1.2 AHEAD-Based Approaches

Another important body of research in this category is a class of *incremental-software-development* (ISD) approaches based on the algebraic hierarchical equations for application design (AHEAD) model [11]. AHEAD has its roots in the GenVoca model, in which a program is constructed as a refinement chain of feature modules.

GenVoca: In the GenVoca model, the root of a refinement chain is a *constant feature* module, which is a set of classes. Other elements in the refinement chain are *function feature* modules, which consist of classes and *class refinements*. A class refinement is a mixin that can add attributes and methods to an existing class, and can also override the class's existing methods. The composition of a function feature module with a program results in a refined program that is extended with the feature's classes, and is refined with the feature's class refinements. As in FOP, if a feature module A includes a refinement of class C, it must be composed after the feature B that introduces class C. A method override in this refinement represents an override of B's behaviour by A. Again, not invoking *super* in a method override represents the suppression of B's behaviour by A.

One implementation of the GenVoca model is the *Jak* language, which extends Java with syntax for denoting class refinements and for labelling classes and class refinements with the name of their feature module. Jak supports two kinds of static configuration, given an ordered composition of feature modules $F_n(F_{n-1}\ldots(F_i)\ldots)$. In the first kind (provided by the *jampack* tool), the resulting program comprises the union of the classes of the composed features, where a class C of a feature F_j, $n-1 \leq j \leq i$ has been refined, in order, by the corresponding class refinements of features F_{j-1} to F_n. The refinement of a method m with an overriding method results in replacing m's body with the overriding method's body, with the invocation of *super* replaced with m's original body. In the second kind (provided by the *mixin* tool), an inheritance hierarchy is generated for each class C of each composed feature, where C is at the root of the hierarchy, and the last class refinement applied to C in the composition is at the bottom of the hierarchy.

AHEAD: AHEAD generalizes GenVoca, in that it applies to hierarchies of artifacts, called *collectives*, instead of individual programs. Similarly to GenVoca, a collective is constructed as a refinement chain of feature modules. Each artifact in a collective has a name and a type (e.g., an artifact "X" of type program). An artifact type may be designated as *compound*, in which case, it can have sub-artifacts (e.g., a package of programs). An AHEAD constant feature module is a collective whose non-compound nodes are base artifacts. A base artifact can be a GenVoca constant feature module, or any artifact whose structure can be mapped to a set of classes. For example, a BNF grammar can be mapped to a class by mapping tokens to attributes and productions to methods. An AHEAD function feature module is a collective whose non-compound nodes are base artifacts and artifact refinements. Analogously to base artifacts, an artifact refinement can be a GenVoca function feature module, or any artifact whose structure can be mapped to a set of classes and class refinements. Because AHEAD feature modules are collectives, an AHEAD refinement chain is an ordered composition of collectives. Two collectives are composed by *superimposition*: Starting with the root, a pair of nodes from the two hierarchies are merged if they have matching names and types. Two compound nodes are merged by recursively merging their children where possible, and adding any unmerged children to the merged compound node[3]. Two non-compound nodes are merged either by a composition analogous to the *jampack* composition of Jak programs, which is applicable to any AHEAD artifact; or by a composition specific to the artifact type of the nodes (e.g., *mixin* composition for Jak programs). Apel et al. [6] have abstracted the ideas of AHEAD into an FOSD algebra, in which feature modules are represented as feature structure trees (FSTs) that are composed by superimposition. An FST is an abstraction of a collective, in which base artifacts and artifact refinements can themselves be designated as compound nodes and be further decomposed. For example, a GenVoca program can be decomposed into its constituent classes, which themselves can be decomposed into their constituent attributes and methods.

Examples of the application of AHEAD and its abstraction [6] to non-code artifacts can be found in [88] and [1]. In [88], Trujillo et al. report on a case study in which AHEAD is used in the context of model-driven development [88]. In this work, the AHEAD model of refinement is used for both code (e.g., Jak and JPS code) and non-code (e.g., XML) artifacts, in conjunction with *model derivations* that transform one model format to another (e.g., one XML format to another, XML to Jak code, and XML to JSP code). In [1], Apel et al. describe FST-based modelling and composition for UML class, state-machine, and sequence diagrams. In this work, the non-compound elements of class and state-machine diagrams (e.g., attributes, associations, and transitions) are merged by replacing existing elements with refinement elements. In sequence diagrams, interactions are ordered and each have unique names, and are therefore composed by concatenation.

AHEAD extensions: AHEAD has been extended in several ways. Kuhleman et al. [60] have investigated *non-monotonic* features and their composition. In addition to adding members to classes and overriding methods, non-monotonic features can additionally refine classes by renaming and deleting their members. Such features typically encapsulate object-oriented refactorings. Liu et al. [66] proposed separating a feature's refinement of an optional feature into a *derivative* feature module. In this sense, a derivative is similar

[3] It is assumed that the root of all feature modules have the same name and type, which is compound.

to a lifter in FOP. However, derivatives differ from lifters in that they are themselves feature modules and are composed with other feature modules by superimposition. Also, there exist *higher-order* derivatives that represent behaviour overrides due to more than one feature. Rosenmuller et al. [79] have proposed a dynamic configuration approach for AHEAD-based FOP. In this approach, the refinement chain of feature modules is transformed into a set of *decorator-pattern* instances, one instance for each refined class. The decorator pattern enables adding (removing) behaviour, encapsulated in a *decorator* class, to (from) instances of the decorated class at run-time. In [79], the decorators are refinements of the decorated class.

Apel et al. [4] have investigated the relationship between AHEAD-based FOP and aspect-oriented programming (AOP) [58]. AOP aims at modularizing *crosscutting concerns*[4]. The idea is that the concern-space of a program is multi-dimensional. A program is decomposed into modules based on the criteria of localizing the implementations of concerns along some "main" dimension of the program's concern-space. As a side-effect of this decomposition, the implementations of concerns along other dimensions get scattered across the modules; in other words, they *crosscut* the module boundaries. AOP introduces *aspects* as a unit of modularity for localizing crosscutting concerns relative to a *base program*. An aspect can affect the base program at multiple *join points* by pieces of code called *advice*. A join point is either a structural element of the base program (aka a *static join point*), or a point in its execution (aka a *dynamic* join point). An advice affects a static (resp. dynamic) join point by adding (resp. executing) the advice code at the join point. The set of join points at which an advice is to be applied is specified by a *pointcut*. Apel et al. point out that features are often cross-cutting concerns; that is, the implementation of a feature often crosscuts an object-oriented program. Feature modules localize this implementation by grouping together classes and class refinements that realize the feature. It would appear that feature modules and aspects are competing technologies. However, it is argued in [4] that they have complementary strengths and weaknesses. First, feature modules are suited to localizing *heterogeneous* crosscuts, which apply distinct advice (borrowing AOP terminology) at each join point. Aspects on the other hand are suited to localizing *homogeneous* crosscuts, which apply the same advice at several join points. On the flip side, using feature modules for homogeneous crosscuts leads to code replication, and using aspects for heterogeneous crosscuts hinders traceability from each distinct advice to the affected join point. Second, feature modules have good support for advising static join-points, but very limited support for advising dynamic join-points,

[4] A *concern* is some issue that is of interest to stakeholders. This definition is very close to that of a feature. According to [4], "all features are concerns (possibly crosscutting concerns), but not all concerns are features of a domain".

in the form of method overrides. Aspects (as available in the popular AOP language AspectJ [57]), have rich support for advising dynamic joint-points, but limited support for advising static join-points, in the form of inter-type declarations. To exploit the strengths of both approaches, Apel et al. propose *aspectual feature modules*; that is, feature modules that include aspects as another artifact type. In this approach, aspects themselves can be refined following the AHEAD model. This idea has been implemented in the FeatureC++ language [3].

2.2.1.3 Feature-Oriented ADORA

Meier et al. have proposed an extension of the ADORA requirements modelling language (see Section 2.1.2) that enables crosscutting concerns to be localized as aspects [69]. With respect to the behaviour view, an aspect can specify that a new state-machine transition should apply before, after, or (unconditionally) instead of an existing state-machine transition. Stoiber et al. further extended the approach to associate each aspect with a feature of a product line [83]. The ADORA model representing a particular feature selection is obtained by statically composing the corresponding aspects with a core ADORA model. Valid feature selections are specified in a tabular representation of a feature model.

2.2.1.4 FOAs used for Feature-Interaction Management

Feature-interaction management (i.e., detecting unintended interactions, deciding whether they are desired, and resolving undesired interactions) has been studied extensively for years [15]. Research in this area has its roots in, and has primarily focused on, the domain of telephony systems. However, there exist approaches that are more general and are not specific to the telephony domain (e.g., [61]). Most feature-interaction management approaches formally specify the behaviour of each feature in a feature module, and analyze the composition of the feature modules (one exception is [19], which is a *variation-mechanism* approach, discussed in Section 2.2.3). In the telephony domain, most features are optional, incremental units of functionality [93] that are enhancements to the basic call service (BCS). Many approaches specify BCS in a separate module (e.g., [94, 42, 16, 65, 44, 26, 87]), whereas some embed BCS in each feature specification (e.g., [37, 34]). In the latter approaches, a feature's suppression of BCS behaviour is specified implicitly in the feature module, which is an integrated model of the behaviour of both the feature and BCS. In the former approaches, various mechanisms are used to model such behaviour suppressions, as discussed below.

In [18], Calder et al. group feature-interaction management approaches into those whose feature specifications are a set of declarative properties (e.g., [37, 34, 61]), those whose specifications are operational models (e.g., [94, 42, 42, 16, 65, 44]), and those that are both (e.g., [26, 87]). The composition of feature and BCS modules is mostly dynamic. One exception is [42], where feature modules are merged statically. The following are a few existing approaches for dynamically composing feature modules.

Architecture: One approach is to compose feature and BCS modules as "plug-ins" to an architecture. Examples include distributed feature composition (DFC) [94] and layered state machines (LSM) [16], which are both adaptations of the pipe-and-filter architectural style. Here, the filters are feature or BCS module instances, which process events (e.g., telephony signals) in order of precedence along a pipe.

In DFC, upon a call request from one network address to another, instances of feature and BCS modules (DFC boxes) are assembled dynamically into a *usage graph* by the *DFC router*. A usage graph is typically a linear chain of DFC boxes, terminated at each end by the BCS modules for the caller and callee. The nodes in this graph are DFC box instances, and the edges are *internal calls* between these instances. An internal call comprises a two-way FIFO signalling channel and some media channels. Each network address is associated with a persistent BCS instance. A usage graph is assembled as follows: When a call request is made from address A to B, the BCS instance of A sends an internal call request, carrying the source and target of the call (in this case A and B) to the DFC router. Based on the feature subscriptions of A and B and feature-precedence relationships, the DFC router establishes an internal call to the next feature instance in the chain – the last of which is the target BCS. As each feature instance is added to the usage, it can continue the usage graph by sending its own internal call request to the DFC router. In doing so, the feature instance can provide source or target values that differ from those in its received internal-call (this is called *address translation*). Alternatively, it can discontinue the graph by tearing down its received internal call. This process continues until the usage is completely assembled with the internal call to the target BCS instance. During a call, a feature can act *transparently* by simply relaying signals between its internal calls with neighboring boxes; or it can take *autonomous* actions such as consuming signals (rather than propagating them along the chain) and generating new signals. The usage graph may change during a call, due to box instances requesting new internal calls or tearing down existing ones.

In LSM, a BCS or feature module is represented by one or more state machines. For example, the BCS module is represented by an originating and a terminating state machine.

When a call is made, a *call stack* is created for each user in the call. At the bottom of each stack is an instance of the originating or terminating state machine of the BCS module, depending on the role of the user in the call. On top of these are state-machine instances of features selected by the user, in order of precedence. Events travel up and down the call stack and are processed by state-machine instances in order. As in DFC, a state-machine instance can act transparently, or autonomously by consuming or generating events, or performing actions such as asserting an invariant, establishing a connection, and giving resources to users. Events can be to or from the network or users. Additionally, when a state-machine instance wants to make a transition, it first sends a *transition notification* event through the call stack from the top; it takes the transition only if this event traverses back to the state-machine instance (i.e., higher-priority features do not prohibit this event).

In DFC and LSM, a feature module's autonomous actions represent overrides of BCS behaviour. In particular, consuming a signal sent by or aimed at a BCS instance represents a suppression of BCS behaviour. In DFC, an address translation also represents an override of BCS behaviour: address translation effectively suppresses the BCS behaviour of specifying the original source or target of the call, which in turn changes the planned usage graph starting at the point of the address translation. In DFC, such overrides are specified by changing the signals to or from a BCS instance. However, in LSM, a feature can also prevent a particular transition of the BCS state machine from executing by consuming the corresponding transition-notification event.

Custom composition operator: A second approach is to compose feature and (if applicable) BCS modules by a custom composition operator. Examples include [61] by Laney et al., [44] by Hay and Atlee, and [42] by Hall, which adapt a composition operator provided by an existing formalism. In the approach by Laney et al., feature modules are expressed as event-calculus properties (a form of explicit-time temporal properties) and are composed by variants of logical conjunction. The variants behave differently from regular conjunction, and from each other, when one operand generates an event prohibited by the other. The latter is a manifestation of an unintended feature interaction. The variant "conjunctions" realize different resolutions of the interaction including prohibiting the event, ignoring the prohibition, or allowing any behaviour depending on the event and on the relative priorities of the operand features. In Hay and Atlee's approach, feature modules are expressed as state machines composed by a variant of parallel composition. Similar to the Laney et al. approach, the operator semantics differs from parallel composition in the case of unintended feature interactions. Specifically, when two feature state machines concurrently perform conflicting actions, only the action of the higher-priority feature is allowed to execute; and when the action of one feature state machine violates an assertion raised by another feature state machine, the action is disallowed, unless the violating feature is of higher priority. The priority relationship between features is a parameter of the composition operator. A similar composition operator is defined by Hall, between the feature state machine (which results from statically merging the selected feature state machines) and the BCS state machine, which gives priority to feature actions when the feature and BCS state machines concurrently perform conflicting actions.

In the above approaches, the composition operator resolves unintended feature interactions between a pair of features by suppressing the behaviour of one of the interacting features. The choice of which feature's behaviour to suppress is based on the parameters of the composition operator; for example, feature-priority relationships. Note that even though suppressions of BCS behaviour by enhancement features are intended, the Hay-Atlee and Hall approaches do not make this intention explicit; instead, such overrides are treated as resolutions to unintended feature interactions. Also, such resolutions are coarse-grained: any action of a higher-priority feature will override a conflicting action from a lower-priority feature regardless of the action.

Existing composition operator: A third approach is to use composition operators provided by the language in which the feature and (if applicable) BCS modules are expressed. Examples include [37] by Gammelgaard et al., [34] by Felty et al., [26] by Combes et al., [87] by Thomas, and [90] by Turner et al. In the approaches by Gammelgaard et al. and Felty et al., individual feature modules are expressed as the logical conjunction of a set of first-order and temporal-logic properties, respectively. Two feature modules are composed by logical conjunction. In the approach by Combes et al., a feature or BCS module is expressed as an SDL process, and a set of temporal-logic properties. An execution of the model comprises one instance of each SDL process per subscriber. The process instances are composed by parallel composition, and the feature instances (SDL processes) of a subscriber can communicate with the corresponding BCS instance (another SDL process) by asynchronous message passing. The temporal-logic properties are composed by logical conjunction. The parallel composition of a subscriber's feature and BCS instances should satisfy the logical composition of the corresponding temporal-logic properties. In the approach by Thomas, a feature or BCS module is expressed as a LOTOS process and a set of temporal-logic properties. There is one feature or BCS process per user; the processes of a user are composed by the LOTOS choice operator, and they synchronize over shared events and variables. As in the Combes et al. approach, the temporal-logic properties are composed by logical conjunction; and the composition of a user's feature and BCS-processes should satisfy the composition of the corresponding temporal-logic properties. In the approach by Turner et al., telephony features are modelled using the Appel language: Each feature is modelled as a set of *rules*, where a rule comprises a trigger, a guard condition, and a set of actions. The rules of the same or different features can be composed using APPEL's sequence, parallel, and choice composition operators. ter Beek et al. [85] have built on this work by translating features expressed in APPEL into UML state machines to detect conflicts between the actions of different features.

In the approaches by Gammelgaard et al. and Felty et al., a feature's suppression of BCS behaviour is specified implicitly in the feature module, which is an integrated model of the behaviour of both the feature and BCS. The SDL and LOTOS languages do not have direct support for suppressing the behaviour of a process (instance); therefore, the Combes et al. and Thomas approaches specify a feature's suppression of BCS behaviour indirectly. In the Combes et al. approach, behaviour suppression is specified following the AIN model [12]: between call-processing steps s_1 and s_2, a user's BCS-process instance b invokes a feature-process instance f by sending it a message, and awaits a response from f to resume execution. In its response, f can instruct b to resume execution at a call-processing step other than s_2, thereby suppressing b's default behaviour. In the Thomas approach, the actions of each feature or BCS process are guarded by a condition over a shared variable that denotes the enabledness of the process. A feature process can suppress the behaviour of a BCS process by setting the enabledness variable of the BCS process to false. In the Turner et al. approach, priorities between features can be expressed by composing their rules in sequence.

2.2.1.5 Delta-Oriented Programming

Delta-oriented programming (DOP) was introduced as an alternative to AHEAD-based approaches (see Section 2.2.1.2). A DOP module specifies combinations of features, rather than a single feature. Similar to AHEAD, a DOP program comprises a *core module* and a set of *delta modules* that refine other (core or delta) modules. The core module lists the set of features that it specifies, which includes the mandatory features and can also include optional features (the core module is not unique). A delta module declares the feature combinations that it specifies as an *application condition*, which is a propositional constraint on possible feature selections.

In an instantiation of DOP for Java, called DELTAJAVA, the core module comprises a set of classes and interfaces. A delta module can add classes and interfaces, and can refine classes and interfaces introduced in other modules. A class can be refined by adding new attributes and methods, changing the supertype and constructor, and removing and renaming existing attributes and methods. The supertype and methods of an interface can be similarly refined. Unlike Jak (an AHEAD-based approach), DELTAJAVA does not support method overrides. Given a feature selection, the delta modules whose application condition is satisfied by the feature selection are composed with the core module in some order. It is possible to have *conflicts* in composition: for example, a previously composed delta module may rename or remove an element referenced by a subsequently composed module. DeltaJava enables the modeller to resolve some conflicts by specifying a partial order on delta modules. Furthermore, DeltaJava has an integrated analyzer for proving, for a given program, that all composition orders that respect the partial order are conflict free.

DOP has also been applied in the context of model-based testing [67], where core modules comprise labelled transition systems, and delta modules can refine other modules by adding and removing states and transitions.

2.2.2 Structural-Annotation Approaches

In a structural-annotation approach, the structural elements of an artifact are annotated with expressions over the feature selection. Static configuration is performed by a model transformation that is based on the evaluation of these expressions.

In the approach by Czarnecki et al. [28], the expressions are either *presence conditions*, which annotate model elements (where the model is based on the meta-object facility or a comparable formalism); or *meta-expressions*, which annotate attributes of model elements (e.g., an element's name). Upon configuration, the model transformation removes elements whose presence conditions evaluate to false, and assigns to element attributes the evaluation of their meta-expressions. Heidenreich et al. [45] and Pohl et al. [77] take a similar approach by annotating the elements of a model with presence conditions; however, the presence conditions are over single features. Classen et al. [23] annotate the transitions of a labelled transition system with presence conditions (over single features), but also specify priority relationships between transitions. Upon configuration, in addition to removing transitions whose presence condition evaluates to false, the model transformation removes transitions for which the presence condition of a higher-priority transition evaluates to true. Finally, in the common variability language (CVL) [27], an artifact is annotated with a specification of its variation points (e.g., the presence or absence of an element, the value assignment to an element), and each variation point is bound to exactly one feature. Upon configuration, the model transformation resolves the variability based on the feature selection (e.g., removes an element, assigns a particular value to an element). When applied to behaviour models, structural-annotation approaches specify behaviour removal as the removal of the model's structural elements (e.g., transitions, actions).

Kästner et al. have compared structural-annotation approaches to AHEAD-compositional approaches for code artifacts [55]. Some of their findings are as follows:

- In terms of *feature traceability* (i.e., the ability to trace a feature to its representation in an artifact), compositional approaches provide direct support: the realization of a feature is localized in a feature module. In contrast, annotation approaches inherently offer poor feature traceability, especially for cross-cutting features. However, this weakness can be alleviated through tool support.

- In terms of the granularity of possible class refinements, the most fine-grained refinements possible using AHEAD-compositional approaches are method overrides. This means that it is not possible to interleave statements of the overriding method with those of the overridden method, without resorting to workarounds like *hook methods*. In annotation approaches, such refinements, and in general, finer-grained refinements at the level of statements, parameters, or expressions are possible.

- In terms of the syntactic correctness of the configured artifact, both annotation and AHEAD-compositional approaches pose similar challenges.

- In terms of language independence, annotation and AHEAD-compositional approaches perform equally well.

- In terms of ease of adoption for an existing code base, refactoring the code into feature modules requires more effort than annotating it.

Kästner et al. propose extending compositional approaches with annotations; that is, annotating elements of feature modules to express fine-grained refinements that are not possible with a compositional approach (without workarounds).

2.2.3 Variation-Mechanism Approaches

Some artifact description languages have built-in support for specifying variability (aka *variation mechanisms*) [29]. Examples include transition guards in state machines for specifying variant state-machine executions, alternative scenarios in use-case descriptions for specifying variant actor-system interactions, and inheritance in class diagrams for specifying class variants (i.e., subclasses). Approaches in this category associate features with variation mechanisms provided by the artifact description language. Some examples are described below.

In a sample application of the feature-oriented domain analysis (FODA) method [54] (briefly described in Section 2.2.4), features are explicated in state machines using conditions on the feature selection in transition guards. In a feature-interaction management approach by Calder et al. [19], conditional statements are used to embed feature behaviour into a Promela model of BCS behaviour; the conditions are over the feature selection (encoded using global variables). In a similar approach, Gnesi et al. [39] introduce the process algebra CL4SPL for modelling SPL behaviour, in which variability is encoded via expressions that are conditional on the feature selection.

Fischbein et al. [35] proposed the use of modal transition systems (MTSs) [63] to specify variability in SPL behaviours. An MTS is a labelled transition system that distinguishes between mandatory and optional transitions; mandatory (optional) transitions can be naturally associated with mandatory (optional) features. Fantechi et al. proposed two extensions to MTSs to enable specifying constraints on the selection of optional transitions, and hence optional features. The first, extended modal transition systems (EMTSs) [32], have constructs for specifying that at most one, or at least one, of a set of optional transitions leaving a given state must be selected. The second extension, generalized extended modal transition systems (GEMTSs) [33], generalize these constructs to specify that at least, at most, or exactly k of the optional transitions must be selected. Asirelli et al. [9] introduced the temporal logic MHML as an alternative approach for specifying constraints on the selection of optional transitions in MTSs.

Griss et al. have integrated feature modelling into the reuse-driven software-engineering business (RSEB) method [51] resulting in FeatureRSEB [40]. The original RSEB is an object-oriented software-development method with a focus on modelling variability. In RSEB, the UML is extended with the notion of *variation points* in model elements (e.g., use cases), which are associated with corresponding variants using the model's variation mechanisms (e.g., use-case extensions). In FeatureRSEB, variable features are mapped to variants of the variation points. Bertolino et al. [13] introduced constructs for annotating use cases with variability information, including making requirements conditional upon the feature selection.

2.2.4 Discussion

Existing approaches for developing FOAs have limitations with respect to the desired properties for a feature-oriented requirements modelling language (see Section 1.2):

Feature-oriented descriptions of SPL requirements: The idea of modelling the common and variable requirements of an SPL's products as features was introduced in Kang et al.'s FODA method [54]. FODA is best known for introducing feature models but it also proposes describing an SPL's requirements using a combination of an ER model and an integrated behavioural model parameterized by features. FODA does not prescribe a particular behavioural modelling language, but [54] gives an example that uses statecharts. However, the statechart and ER models in the example are not interrelated. In subsequent work, feature models have been integrated with several requirements-description techniques, including use cases (in [22], [40]), and goals and scenarios (in [75]). However, such approaches do not follow the Jackson and Zave RE reference model, do not result in precise models, and do not support the explicit modelling of intended feature interactions.

In most feature-interaction management approaches, the FOAs used are formal, and in some cases, are at the requirements-level. However, the purpose of the FOAs is not to serve as requirements models, but as analysis models used for feature-interaction management. Therefore, the FOAs generally do not employ RE abstractions (e.g., a world model). One exception is the approach by Laney et al. [61], in which a problem diagram is used to describe the world and event-calculus properties are used to describe world assumptions and feature requirements. However, most software engineers are not trained in the use of the event calculus, which can hinder adoption. In fact, in the Laney et al. approach, event-calculus is not proposed as a requirements-modelling language to be used by software engineers; rather, it is used as a tool to illustrate a feature-interaction resolution strategy.

In the feature-oriented extension of the ADORA requirements modelling language (see Section 2.2.1.3), the composition of a set of features with a core ADORA model is not commutative. Furthermore, the approach has limited support for explicitly representing intended interactions: explicit models of conditional behaviour overrides and the triggering of existing behaviours are not supported.

Feature-oriented behaviour modelling: Existing structural-annotation and variation-mechanism approaches do not support feature modularity. On the other hand, existing compositional approaches (which support feature modularity) have the following limitations:

In FOP, AHEAD-based approaches, and architectural approaches (e.g., DFC, LSM), intended feature interactions (e.g., behaviour overrides) and automatic resolutions to unknown interactions are realized by an order-sensitive and hence *non-commutative* composition of features: here, the composition order specifies a total priority order among the features, such that a subsequently-composed feature can modify the behaviours of previously-composed (lower priority) features. This approach has the benefit of resolving unknown conflicts between any pair of features through priority; however, such resolutions are not explicit and may not be desired. It may therefore be necessary to analyse multiple composition orders, to arrive at one that both realizes the intended feature interactions and does not cause undesired resolutions – and to redo the analysis when a new feature is added to the SPL and the new feature's position in the composition order needs to be determined. Finally, in such approaches, intended interactions are specified *implicitly*: in the composition order.

Composition order also matters in DOP and can result in conflicts (e.g., a previously composed delta module can remove an element referenced by a subsequently composed module). In DOP, behaviour removal is explicitly modelled (e.g., by the removal of methods in Java programs, or the removal of transitions in labelled transition systems); however, DOP does not support the explicit modelling of behaviour overrides.

Other feature-interaction management approaches have limited support for explicitly modelling intended feature interactions. The approaches by Hay and Atlee [44], Hall [42], and Thomas [87] support modelling behaviour overrides via feature priorities; however, such feature-level priorities are coarse-grained and do not enable specifying more specific behaviour overrides (e.g., between particular feature transitions). The approach by Combes et al. [26] provides a finer-grained mechanism for specifying behaviour overrides using the AIN model; however, this approach is specific to the telephony domain. In the approach by Turner et al. [90], priorities between features can be expressed by composing their rules in sequence. However, priorities cannot be applied to rules composed in parallel to resolve conflicts between their actions; rather, resolution requires restructuring the composition of the rules by changing the parallel composition into sequential composition.

2.3 Feature-Interaction Taxonomies

Unintended feature interactions are prevalent in the telephony domain: most telephony features enhance BCS, thus the features are highly prone to unintended interactions due to their small shared context. A research community was established for investigating feature-interaction management in this domain and has made substantial progress. Whereas Section 2.2 describes the aspects of modelling and FOAs in feature-interaction management approaches (along with other FOA approaches), this section focuses on the types of feature interactions defined by such approaches, for the purpose of detecting unintended interactions. A discussion of the limitations of such definitions is given in Section 2.3.1.

An early taxonomy proposed by Cameron et al. [20] groups feature interactions in telephony systems based on their *nature* and *cause*. The nature of a feature interaction is defined by whether the features involved are *customer* (e.g., call waiting) or *system* (e.g., billing) features, and by the number of users and network components involved (single or multiple). Cameron et al. identify several causes for feature interactions, such as the violation of a feature's assumptions (e.g., features that override BCS behaviour violate another feature's assumptions about BCS behaviour), network-support limitations (e.g., the limited set of signals used by many features), and resource contention between features (e.g., contention over the use of a three-way bridge by call waiting and three-way calling).

Whereas the taxonomy by Cameron et al. is independent of a particular feature-interaction management approach, each particular approach defines how feature interactions are *manifested* in its FOAs (the different types of FOAs used in such approaches is discussed in Section 2.2.1.4). Nhlabatsi et al. [70], Calder et al. [18], and Bruns [17] have proposed taxonomies that are based on surveys of feature-interaction management approaches and on the manifestations of feature interactions in different types of FOAs. A combination of these taxonomies is presented below, following the structure of the Calder et al. taxonomy.

- *FOAs that include an operational model:*
  1. The composition of the feature modules results in undesired properties such as
     (a) Nondeterminism
     (b) Deadlock
     (c) Two features modules performing concurrently conflicting actions
     (d) Resource contention between two feature modules (e.g., contention over access to a voice channel in DFC)
     (e) An action in one feature module violating an assertion raised by another feature module
     (f) Infinite loops (e.g., two feature modules cyclically triggering one another's behaviour)
  2. An action in one feature module triggers or suppresses behaviour in another feature module (e.g., in DFC, a feature instance generates or consumes a signal, as part of its enhancement of the signal's destination BCS instance, that another feature-instance in the route of the signal can react to).
  3. In approaches where features are sequentially composed (e.g., DFC, LSM, FOP, and AHEAD-compositional approaches), different composition orders result in different behaviours.

- *FOAs that include a set of properties:* The properties of two features are logically inconsistent.

- *FOAs that include both an operational model and a set of properties:* The operational models of two feature modules satisfy the properties in their modules, but the composition of the operational models does not satisfy the conjunction of their corresponding properties.

More recently, the study of feature interactions has been extended to the automotive domain. For example, Juarez-Dominguez et al. [53] define a feature interaction as two automotive features either making conflicting requests to the same actuator of the vehicle (e.g., conflicting requests to the vehicle's throttle); or making requests to different actuators, causing conflicting changes to the environment (e.g., requests to the vehicle's throttle and brakes, causing conflicting changes to the vehicle's speed). The definition includes both conflicting requests and changes that occur simultaneously (i.e., race conditions) as well as those that occur within some time threshold of one another.

2.3.1 Discussion

The majority of existing (formal) taxonomies of feature interactions are associated with modelling languages that do not support the explicit modelling of intended interactions, and, as such, the associated taxonomies do not distinguish and exclude intended interactions. Also, even though a feature interaction is often informally defined as one feature modifying the behaviour of another feature, existing taxonomies rarely include a precisely defined feature-interaction type that corresponds to the *general* case of behaviour modification. Instead, they include various feature-interaction types that are special cases of behaviour modification.

Chapter 3

FORML

This chapter describes the FORML language that has been designed for modelling SPL requirements:

- Section 3.1 describes two running examples used in the remainder of this chapter.
- Sections 3.2 to 3.4 describe the different parts of a FORML model: Section 3.2 describes the *world model*, which is an ontology of concepts that describes a world comprising the products of an SPL and the environment in which they will operate. Section 3.3 describes FORML's language for writing expressions over the world model (e.g., constraints over the world model). Section 3.4 describes the *behaviour model*, which is a state-machine model of the SPL's requirements, decomposed into feature modules.
- Section 3.5 describes the composition of feature modules into an integrated model of the requirements of the entire SPL.
- Section 3.6 describes the execution semantics for FORML, with a few simplifying assumptions made for ease of presentation; these assumptions are relaxed in Chapter 4.
- Section 3.7 describes an evaluation of FORML with respect to expressiveness and the desired properties of a feature-oriented modelling language for SPL requirements given in Section 1.2.

3.1 Running Examples

3.1.1 TelSoft

TelSoft is an SPL of telephone services with the following features:

- Basic call service (BCS): BCS responds to the subscriber's requests for starting, accepting, and ending calls.
- CW: CW allows the subscriber to accept a second call and switch between the two calls.
- Voice mail (VM): VM allows callers to leave messages for the subscriber when he or she does not accept a call within some timeout period, and allows the subscriber to check his or her messages.
- Teen line (TL): If the subscriber makes a call request within a designated curfew period, TL requires authentication before starting the call.
- Caller delivery (CD): When the subscriber receives a call, CD delivers the caller's identity to the subscriber.
- Caller delivery blocking (CDB): When the subscriber makes a call, CDB prevents the delivery of the subscriber's identity to the callee, if the callee subscribes to CD.

3.1.2 AutoSoft

AutoSoft is an SPL of automotive software controllers with the following features:

- Basic driving service (BDS): BDS responds to the drivers's requests for turning the vehicle's ignition on or off, accelerating, decelerating, and steering.
- Cruise control (CC): CC maintains the vehicle's speed at some driver-specified value.
- Headway control (HC): HC maintains the vehicle's headway distance from road objects ahead, at some driver-specified distance.

3.2 World Model

The *world* of an SPL is a set of phenomena comprising the following:

- Relevant phenomena in the environment in which the SPL's products will operate: that is, environment phenomena that are referenced or constrained by the SPL's requirements

- The SPL's products, characterized in terms of their constituent features (i.e., their feature configuration)

- *Product phenomena*, which are phenomena introduced by the SPL's products that are observable and possibly controllable by agents in the environment (e.g., messages between products and their environment, feature data that is observable by users)

A **world model** is a *conceptual model* of an SPL's world: that is, a description of the world in terms of a set of *concepts* and the relationships between them. A *concept* is an abstraction that describes a *category* or *type* of phenomena in the world, in terms of their shared properties and their shared relationships with other world phenomena. An *instance* of a concept, often called an *object*, represents a particular world phenomenon of the corresponding type. The appeal of conceptual modelling is that it corresponds to the natural way in which we understand and characterize a set of phenomena.

In FORML, an SPL's requirements are decomposed into features, and one could think about each feature having its own world. However, to ensure a consistent ontology relevant to multiple features, a FORML world model is an integrated model of the features' worlds. Consequently, authors of different features will need to coordinate their changes to the integrated world model. However, the changes to the world model, due to the addition of new features, are less frequent and smaller than changes to the behaviour model.

A world model can be more precisely defined in terms of the metamodel shown in Figure 3.1. The metamodel of Figure 3.1, as well as later metamodels in the thesis, are MOF-based metamodels [72], with the following additional syntax introduced to improve readability: a class with a dashed border represents a reference to a class that appears elsewhere in the metamodel; and if such a class is shaded, it references a class in a different figure.

Definition 3.2.1. *A world model comprises one or more concepts, a set of auxiliary constraints over the concepts, and a feature model.*

Section 3.2.1 describes concept types adapted from UML class diagrams and ER models, whereas Section 3.2.2 describes concept types introduced by FORML. Feature models are described in Section 3.2.3, and auxiliary world-model constraints are described in Section 3.2.6. Section 3.2.7 summarizes what FORML borrows from, and how it extends and adapts, UML class diagrams.

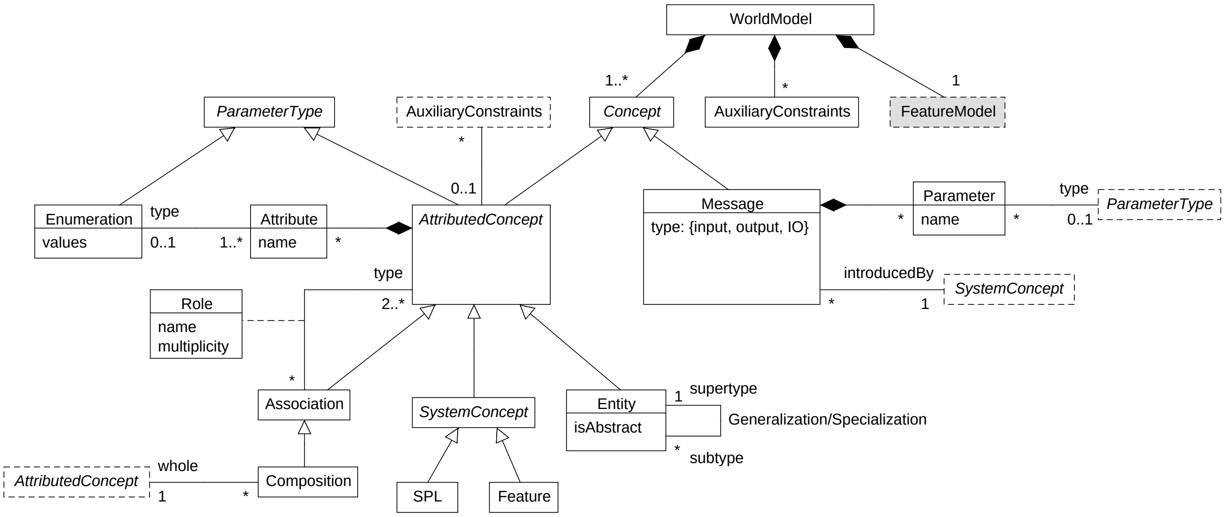


Figure 3.1: The world-model metamodel. The metamodel for the referenced class *Feature-Model* is presented in Figure 3.8.

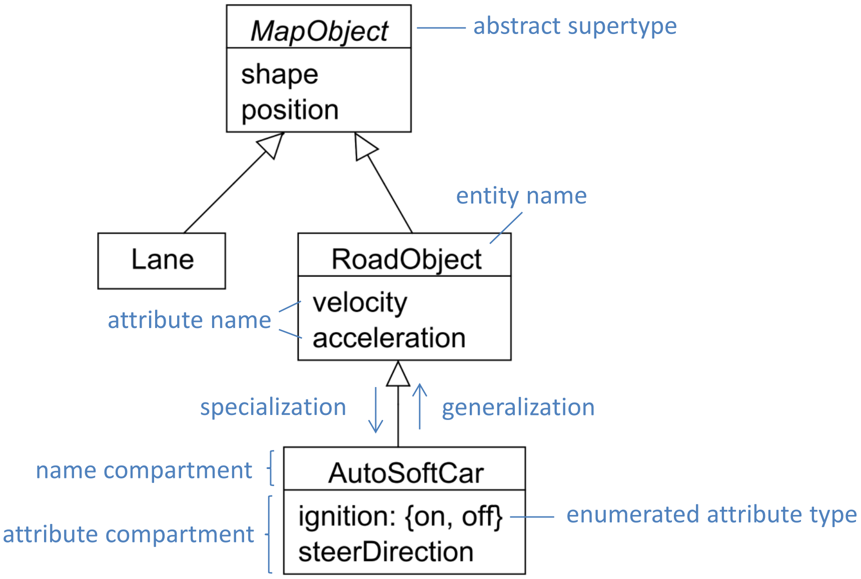


Figure 3.2: Entities

3.2.1 Entities, Associations, and Compositions

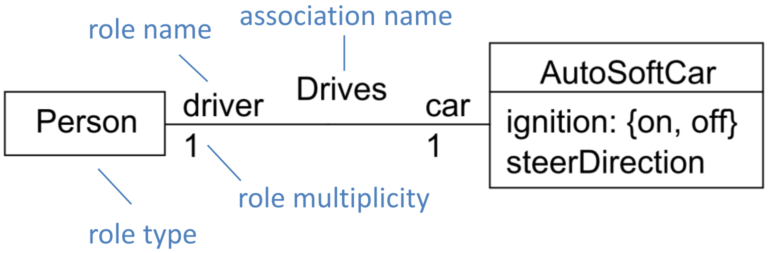


Figure 3.3: Basic binary associations

Entities: An **entity** is a concept that represents a type of environment phenomenon with an independent existence. An entity is characterized by a set of *attributes* and is shown as a UML class. For example, in Figure 3.2, the environment of an *AutoSoft* product includes the entity *RoadObject* with attributes *velocity* and *acceleration*. The *type* of an attribute (i.e., the set of possible values for the attribute) can be left undefined or can be explicitly specified as an enumeration of values.

An entity can be a *subtype* of another entity (the *supertype*), as shown by a UML generalization relationship; the subtype inherits the supertype's attributes, but can also have additional attributes. For example, in Figure 3.2, a vehicle equipped with an *AutoSoft* product is represented by the entity *AutoSoftCar*, which is a subtype of the *RoadObject* entity, with additional attributes *steerDirection* and *ignition*. A supertype can be declared as *abstract* by italicizing its name, which implies that the only objects of the supertype are those of its subtypes. For example, in Figure 3.2, the abstract entity *MapObject* is an abstract supertype defined to factor out the common attributes *shape* and *position* of *Lane* (which models road-segment lanes) and *RoadObject*.

Associations: An **association** is a concept that represents a type of relationship that can exist between the objects of other concepts. The instances of an association are called *links*. A binary association relates two concepts and is shown as a UML association. An association assigns a *role* to each related concept. For example, Figure 3.3 models the relationship between an *AutoSoft*-equipped vehicle and its driver as a binary association *Drives* that has a *driver* role of type *Person* and a *car* role of type *AutoSoftCar*. An association role can have an optional *multiplicity*, which constrains the number of different objects that can (simultaneously) be linked to each object in the other role. A multiplicity can take one of three forms: an exact number n, a range $n_1..n_2$ of numbers where $n_1 < n_2$, or a range $n..*$ of numbers greater than or equal to n. For example, in Figure 3.3, the multiplicity of the *driver* role of *Drives* specifies that each *car* has exactly one *driver* at a

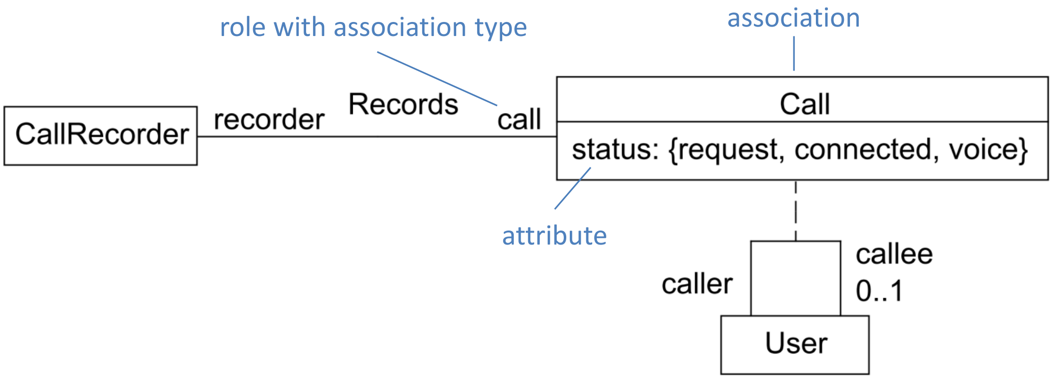


Figure 3.4: Binary associations with attributes or associations

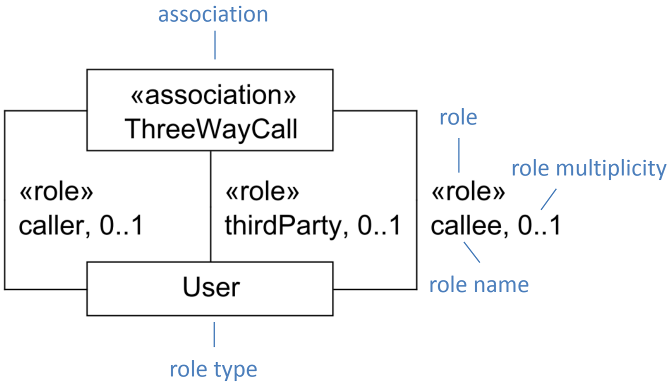


Figure 3.5: Associations with more than two roles

time.

An association can also have attributes and participate in other associations. A binary association with attributes or associations is shown as a UML association class. For example, Figure 3.4 models a telephone call between two *TelSoft* users as a binary association *Call* with a *status* attribute whose possible values represent the phases that a call goes through: a call starts off as a *request*, it gets *connected* if the callee is not busy, and a *voice* connection is established when the callee answers the call. The *Call* association also participates in a binary association with the entity *CallRecorder*; this association relates call-recording systems to the calls that they record.

Finally, an association can have more than two roles. Such an association is expressed as a UML class with the stereotype «*association*». Since there are more than two roles, role names and multiplicities cannot be shown at the (two) ends of a UML association as before.

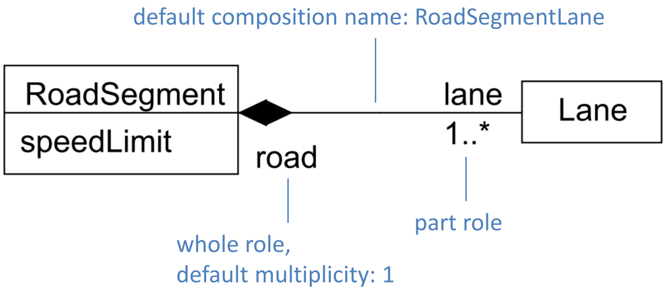


Figure 3.6: Compositions

Instead, each role of the FORML association is depicted as a separate UML association between the FORML association and the concept that is the role type; the role is labelled with the stereotype «*role*» and a comma-separated sequence of the role name and the role multiplicity (if any). For example, Figure 3.5 models a three-way call – established by adding a third party to a basic call – by the ternary association *ThreeWayCall*, which has the roles *caller* and *callee* as with *Call*, plus the role *thirdParty* of type *User*[1].

Compositions A **composition** is a special type of binary association, representing a *whole-part* relationship between a composite object and its component objects. A FORML composition is shown as a UML composition, with the *whole* role distinguished by a filled diamond. Three characteristics distinguish a composition from an ordinary binary association: (1) a composite object, its component objects, and the links between them always come into existence and disappear together; (2) there can be at most one composition link between any pair of whole and part objects; and (3) a component object can be related to only one composite object (i.e., the multiplicity of the whole role of a composition is always 1). For example, in Figure 3.6, the whole-part relationship between road segments and their lanes is modelled by a composition. A FORML composition need not be explicitly named: the implied name of a composition with the whole and part roles types *Whole* and *Part*, respectively, is *WholePart*. For example, the implied name of the composition in Figure 3.6 is *RoadSegmentLane*.

[1] The notion of multiplicities generalizes to associations with more than two roles: here, the multiplicity of a role constrains the number of different objects that can (simultaneously) be linked to each combination of objects in the other roles. For example, in Figure 3.5, the multiplicity of the *thirdParty* role of *ThreeWayCall* specifies that a *caller* and *callee* pair can add at most one *thirdParty* to their basic call.

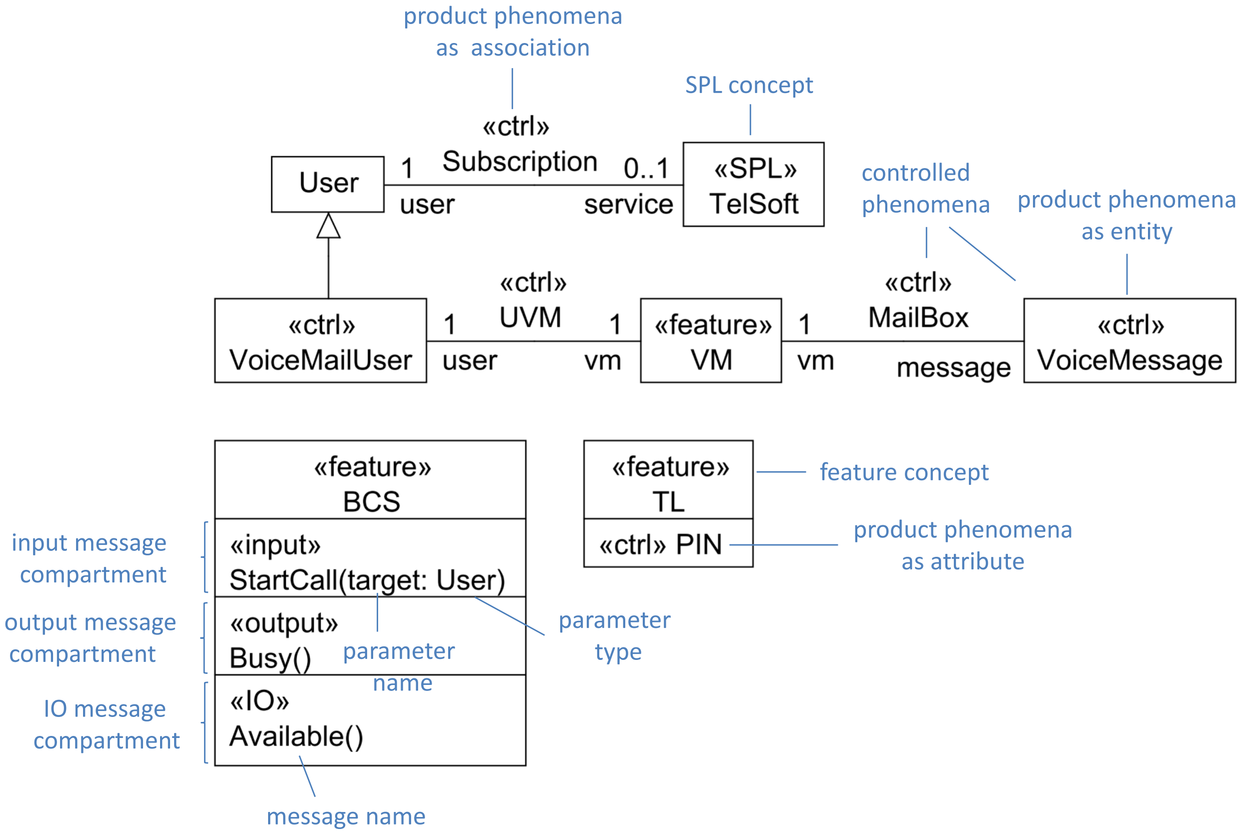


Figure 3.7: SPL, feature, and message concepts

3.2.2 SPL, Feature, and Message Concepts

A world model includes an explicit representation of an SPL's products and their feature configurations. Such a representation is needed for specifying requirements over multiple products (e.g., a telephone call involving the *TelSoft* products of two or more users). In such requirements, one product may refer to other products and their features (e.g., the CDB feature of a caller's *TelSoft* product activates when a callee's *TelSoft* product has CD).

Specifically, a world model includes an **SPL concept** and a set of **feature concepts**, representing an SPL and its set of features, respectively. An instance of the SPL concept, called a *product object*, represents a single product of the SPL. A product object is primarily characterized by its relationship to a set of feature objects, which constitute the feature configuration of the corresponding product[2]. The SPL concept and each feature concept is shown as a UML class with the stereotype «*SPL*» and «*feature*», respectively. For example, in Figure 3.7, the *TelSoft* SPL is represented by the SPL concept *TelSoft*, and its BCS, TL, and VM features are represented by the feature concepts *BCS*, *TL*, and *VM* respectively[3].

Product phenomena: An SPL's products introduce observable phenomena into the world, called *product phenomena*. These include the messages that are exchanged between products and their environment, such as the requests and notifications exchanged between products and their users. Other product phenomena are more varied, such as relationships involving products or features (e.g., the relationship between an *AutoSoft* product and its containing vehicle), feature data that are observable by agents in the environment (e.g., the voice messages stored by VM in a *TelSoft* product), and world phenomena whose existence or values are controlled by the system (e.g., calls between *TelSoft* subscribers).

To link product phenomena with products, models of product phenomena can be related to the SPL concept. However, it is often the case that some product phenomena are specific to particular features of the products: in the above examples, voice messages are specific to VM. In such cases, it is preferable to relate models of product phenomena to the corresponding feature concepts, so as to aid model reviewers in identifying the product phenomena specific to each feature.

Messages: A **message** is a concept that represents a type of communication exchanged between an SPL product and its environment. Although a product can communicate with its environment by reading or writing to shared world phenomena, messages are needed to model directed communications between specific products and environmental phenomena (e.g., a start-call request sent to a *TelSoft* product from its user). If a message represents feature-specific communications, it is related to the corresponding feature concept; otherwise, it is related to the SPL concept.

A message exchanged with an environment agent is either an *input* message representing a communication to an SPL product from an environment agent, or an *output* message

[2] Traditionally, a feature configuration is simply a set of feature identifiers. Although a feature object carries more information than just the feature's identity (e.g., the feature's links and attribute values), it is convenient to represent a product's feature configuration as a set of related feature objects because it enables navigating from a product or feature to related world phenomena.

[3] To simplify presentation, the thesis describes FORML models of single SPLs. To support models of multiple SPLs, the world model should allow multiple SPL concepts and should explicate associations between feature concepts and their corresponding SPL concepts. The description of FORML given in this thesis can be generalized to support models of multiple SPLs.

representing a communication from an SPL product to an environment agent. An input message object is related to its source product object, and an output message is related to its target product object.

Products can also communicate with other products; for example, two telephone services may communicate to establish calls. Such communications are modelled as *IO (input and output)* messages. An IO message object is related to both its source and target product objects.

A message has *parameters* that represent data carried by the communications, such as the desired change in acceleration carried in an *Accelerate* message sent to an *AutoSoft* product. As with attributes, the type of a parameter can be enumerated or left undefined. In addition, a parameter's type can be another (non-message) concept.

A message is shown as a *message signature* in a *message compartment* of the SPL or feature concept to which it is related. Input, output, and IO messages are listed in separate message compartments denoted with the stereotypes «*input*», «*output*», and «*IO*», respectively. A message signature is of the form **M(params)**, where **M** is the message name and **params** is the optional comma-separated list of message parameters. The type of a parameter (if any) is separated from the parameter name by a colon. For example, in Figure 3.7, a *start call* request sent to a *TelSoft* product is modelled as an input message *StartCall* in feature BCS (which is the feature responsible for call processing); the message has a parameter *target* of type *User* that represents the target callee. Figure 3.7 also models busy signals sent from a *TelSoft* product to its user as an output message *Busy* in feature BCS. An IO message *Available* models notifications sent from a callee's *TelSoft* product to a caller's *TelSoft* product, to signal whether the callee is available to participate in a new call. Since all of the above communications are introduced by BCS, the messages are related to the *BCS* feature concept.

Controlled phenomena: Some product phenomena can be controlled (changed) only by SPL products, and are thus called *controlled phenomena*. Examples include all output and IO communications of products, the relationships between *TelSoft* products and their subscribers, calls between *TelSoft* subscribers, and the above example of voice messages introduced by VM. Examples of product phenomena that are not controlled include all input communications of products, which are only controllable by the environment, and the physical containment relationship between *AutoSoft* products and vehicles. As is described in Section 3.4.2, the controlled phenomena must be known in order to correctly determine how the world can change in accordance with an SPL's requirements. Any entity, association, composition, or attribute representing controlled phenomena is denoted

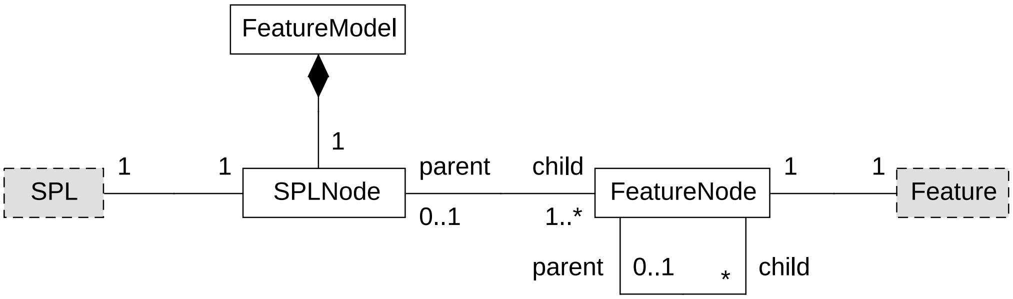


Figure 3.8: The feature-model metamodel. The referenced classes *Feature* and *SPL* are defined in the world-model metamodel in Figure 3.1.

with the stereotype «*ctrl*» (e.g., *VoiceMessage* in Figure 3.7)[4]; an attribute only needs a «*ctrl*» designation if its containing concept does not have one (e.g., the *PIN* attribute of *TL* in Figure 3.7).

3.2.3 Feature Model

The number of different products that can be derived from an SPL's set of features is exponential in the number of features. However, the products of an SPL are usually characterized by the subset of feature configurations that are *valid*, for various technical (e.g., functional dependencies between features) and non-technical (e.g., business decisions to sell certain feature together) reasons. A FORML world model includes a traditional **feature model** [54] that constrains the relationships between product and feature objects, so as to specify the valid feature configurations of an SPL.

A feature model can be described more precisely in terms of the metamodel shown in Figure 3.8.

Definition 3.2.2. *A feature model is a tree whose root references an SPL concept and every other tree node references a feature concept. A feature node topped with a filled circle denotes a mandatory feature, and a feature node topped with an empty circle denotes an optional feature. An optional feature can be present in a product only if its parent feature is present in the product.*

[4] The «*ctrl*» designation is not needed for messages: output and IO messages always represent controlled phenomena, and input messages always represent phenomena that are not controlled.

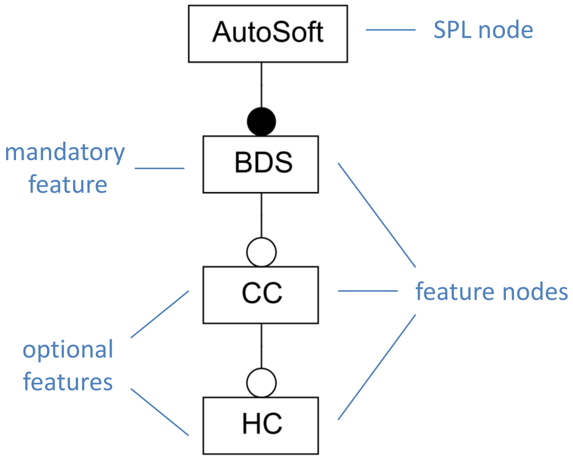


Figure 3.9: The feature model notation

For example, Figure 3.9 shows the feature model for a subset of *AutoSoft*'s features – namely BDS, CC, and HC[5].

[5] To support models of multiple SPLs, the world model would include a feature model for each SPL.

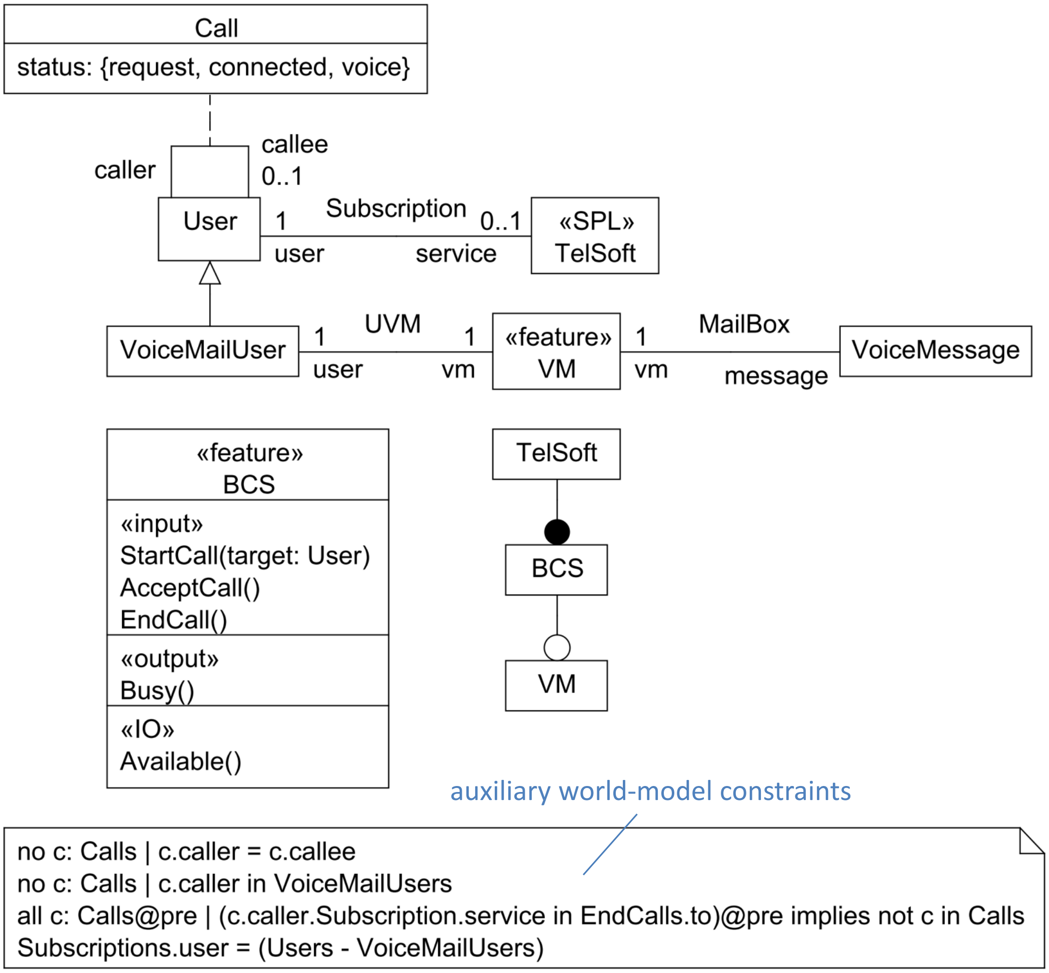

Call
status: {request, connected, voice}
caller
callee
0..1
User
1
Subscription
0..1
user
service
«SPL»
TelSoft
VoiceMailUser
1
UVM
1
user
vm
«feature»
VM
1
MailBox
vm
message
VoiceMessage
«feature»
BCS
«input»
StartCall(target: User)
AcceptCall()
EndCall()
«output»
Busy()
«IO»
Available()
TelSoft
BCS
VM
auxiliary world-model constraints
no c: Calls | c.caller = c.callee
no c: Calls | c.caller in VoiceMailUsers
all c: Calls@pre | (c.caller.Subscription.service in EndCalls.to)@pre implies not c in Calls
Subscriptions.user = (Users - VoiceMailUsers)

Figure 3.10: A partial world model for *TelSoft*

3.2.4 World States

A world model is effectively a concise description of all of the possible states that an SPL's world can be in. Each **world state** comprises a particular set of instances (objects) of the concepts in the world model, where each entity object, link, message object, product object, and feature object has particular values. The *product configuration* of a world state comprises the particular set of product objects in the world state, each with a particular feature configuration (i.e., a particular set of related feature objects).

For example, consider the world model shown in Figure 3.10, which describes part of

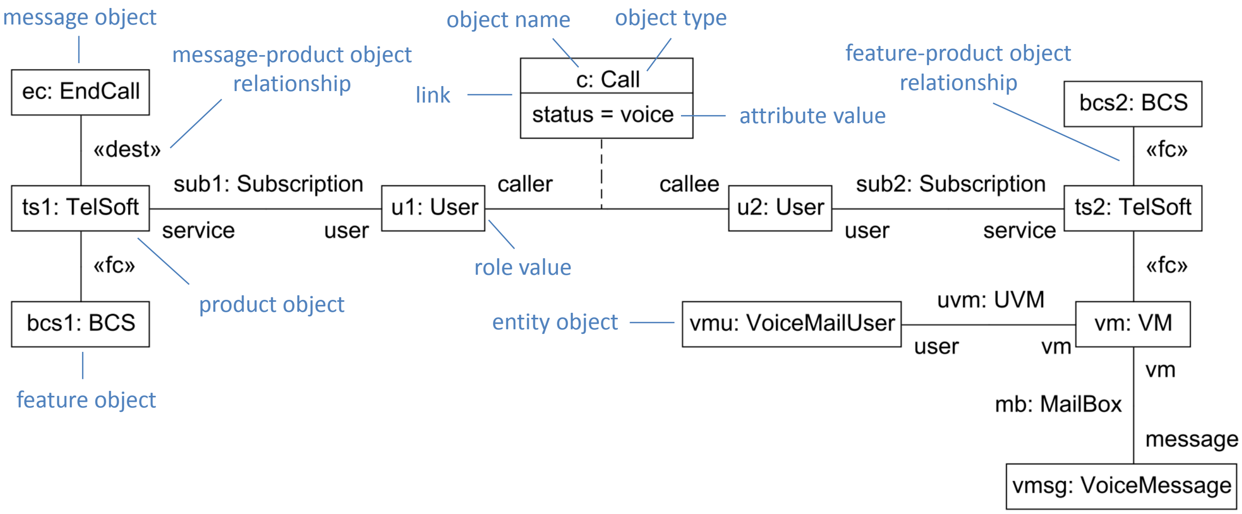


Figure 3.11: A possible world state of the world model shown in Figure 3.10

TelSoft's world. The concepts shown in Figure 3.10 are from the examples given in the previous subsections, with the addition of the messages *AcceptCall* and *EndCall* (introduced by BCS), which represent user requests to accept and end calls, respectively. The feature model specifies that BCS is mandatory in *TelSoft* products but VM is optional. The UML note at the bottom of Figure 3.10 is explained later in Section 3.2.6.

One possible world state of the world model of Figure 3.10 is shown in Figure 3.11 as a UML-like object diagram. The notation used in Figure 3.11 is not part of FORML[6] and is used here only for presentation purposes. The boxes and lines in Figure 3.11 represent objects and their relationships rather than classes and their relationships, respectively. Each object (including links) has a label of the form **n: C**, where **n** is the object's name – assigned for ease of reference – and **C** is the object's type (e.g., *c: Call*). The value **v** of an object's **a** attribute is expressed as **a** = **v** (e.g., *status* = *voice* for the *Call* link *c*). The objects related by a link are given by the link's end points (e.g., *User* object *u1* takes the *caller* role of the *Call* link *c*). Finally, the special relationships between product objects and their feature objects, and between message objects and their source/destination product objects, are distinguished from links using the stereotypes *«fc»* (e.g., the relationship between the product object *ts1* and the *BCS* feature object *bcs1*) and *«src»*/*«dest»* (e.g., the relationship between the *EndCall* message object *ec* and its destination product object *ts1*), respectively.

[6] FORML does not prescribe a notation for specifying individual world states.

Hence, the world state of Figure 3.11 represents the following state of *TelSoft*'s world: There are two users (*u1* and *u2*), each subscribed to a *TelSoft* product (*ts1* and *ts2*), where one user's product has only BCS (*bcs1*) and the other user's product has both BCS and VM (*bcs2* and *vm*). The user subscribed to VM can call a special voice-mail user (*vmu*) to check his voice messages. *vm* currently has one voice message (*vmsg*). The two users are in a voice-connected call (*c*), but *u1* has just requested to end the call (*ec*).

More formally, a world model *wm* defines a collection of types (sets of objects or values), and a collection of declarations of sets and functions over these types. A world state is a value assignment to the sets and functions of *wm*.

Definition 3.2.3. *The set* $\boldsymbol{WS_{univ}}$ *of possible world states of wm is the set of possible value assignments to the sets and functions declared by wm.*

wm defines the following types:

- For each concept C in *wm*, the type $\boldsymbol{T_C}$ is the infinite set of all possible instances (objects) of C.
- If C has an attribute or (if C is a message) a parameter x with an enumerated type, $\boldsymbol{T_{C.x}}$ is the enumerated type of x.
- $\boldsymbol{T_{und}}$ is the universal undefined type: it represents the type of any attribute or parameter in *wm* whose type is not defined.

The following describes each declared set and function $\boldsymbol{D}$ of *wm* in terms of a map $\boldsymbol{D(ws)}$ from a possible world state $\boldsymbol{ws} \in \boldsymbol{WS_{univ}}$ to a value assignment for $\boldsymbol{D}$:

- For each concept C in *wm*, $\boldsymbol{O_C(ws)}$ is the set of objects of type $\boldsymbol{T_C}$ in $\boldsymbol{ws}$:

$$\boldsymbol{O_C(ws)} \subset \boldsymbol{T_C}$$

- If C has attributes, then for each attribute a with type $\boldsymbol{T}$ (i.e., $\boldsymbol{T_{C.a}}$ or $\boldsymbol{T_{und}}$), function

$$\boldsymbol{C.a(ws)} : \boldsymbol{O_C(ws)} \mapsto \boldsymbol{T}$$

  states the value of a for each C object in $\boldsymbol{ws}$.

- If C is an association, then for each role r of C whose type is concept C', function

  $$\boldsymbol{C.r(ws)} : \boldsymbol{O_C(ws)} \mapsto \boldsymbol{O_{C'}(ws)}$$

  states the object that plays role r in each C link in $\boldsymbol{ws}$.
- If C is an SPL concept, then for each feature concept F related to C, function

  $$\boldsymbol{C.F(ws)} : \boldsymbol{O_C(ws)} \mapsto 2^{\boldsymbol{O_F(ws)}}$$

  relates each C product object $\boldsymbol{p}$ in $\boldsymbol{ws}$ to its corresponding F feature object (if any): that is, $\boldsymbol{C.F(ws)(p)}$ returns a singleton set if product $\boldsymbol{p}$ has an F feature object, and the empty set otherwise.
- If C is a message that has parameters, then for each parameter p with type $\boldsymbol{T}$ (i.e., $\boldsymbol{T_{C.p}}$ or $\boldsymbol{T_{und}}$ or $\boldsymbol{T_{C'}}$ if the type of p is concept C'), function

  $$\boldsymbol{C.p(ws)} : \boldsymbol{O_C(ws)} \mapsto 2^{\boldsymbol{T}}$$

  states the value of p (a subset of $\boldsymbol{T}$[7]) for each C message object in $\boldsymbol{ws}$.
- If C is an input or IO message related to the SPL concept *SPL*, function

  $$\boldsymbol{C.to(ws)} : \boldsymbol{O_C(ws)} \mapsto \boldsymbol{O_{SPL}(ws)}$$

  relates each C message object in $\boldsymbol{ws}$ to the *SPL* product object that receives the message object.
- Analogously, if C is an output or IO message related to *SPL*, function

  $$\boldsymbol{C.from(ws)} : \boldsymbol{O_C(ws)} \mapsto \boldsymbol{O_{SPL}(ws)}$$

  relates each C message object in $\boldsymbol{ws}$ to the *SPL* product object that generates the message object.

A world state's product configuration can be precisely defined based on the above sets and functions:

[7] The value of a message parameter can be a set of objects. This provides a convenient way of modelling cases where the data carried by a message object are references to a set of other objects (e.g., the VM feature can send back a set of *VoiceMessage* objects back to the user).

Definition 3.2.4. *Let SPL and $F_1 \ldots F_n$ be the SPL and feature concepts in the world model, respectively. The product configuration of world state $\boldsymbol{ws}$ is given by the function*

$$PC(\boldsymbol{ws}) = \{\ (\boldsymbol{p}, FC(\boldsymbol{ws}, \boldsymbol{p}))\ |\ \boldsymbol{p} \in \boldsymbol{O_{SPL}(ws)}\ \}$$

where $FC(\boldsymbol{ws}, \boldsymbol{p})$ denotes the feature configuration (i.e., set of feature objects) of product $\boldsymbol{p}$ in world state $\boldsymbol{ws}$:

$$FC(\boldsymbol{ws}, \boldsymbol{p}) = \bigcup_{1 \leq i \leq n} \boldsymbol{SPL.F_i(ws)(p)}$$

For example, let $\boldsymbol{ws}$ be the the world state shown in Figure 3.11 of the *TelSoft* world model shown in Figure 3.10. $\boldsymbol{ws}$ can be represented by the sets and functions shown in Figure 3.12. The product configuration of $\boldsymbol{ws}$ is the set

$$PC(\boldsymbol{ws}) = \boldsymbol{\{\ (ts1, \{bcs1\}), (ts2, \{bcs2, vm2\})\ \}}$$

Finally, to reduce the number of parentheses in expressions that evaluate a set or function $\boldsymbol{D}$ with respect to a world state $\boldsymbol{ws}$, in the remainder of this thesis, $\boldsymbol{D(ws)}$ is often equivalently denoted as $\boldsymbol{ws :: D}$.

$$\begin{aligned}
O_{User}(ws) &= \{u1, u2\}\\
O_{Call}(ws) &= \{c\}\\
Call.caller(ws) &= \{(c, u1)\}\\
Call.callee(ws) &= \{(c, u2)\}\\
Call.status(ws) &= \{(c, voice)\}\\
O_{VoiceMailUser}(ws) &= \{vmu\}\\
O_{VoiceMessage}(ws) &= \{vmsg\}\\
O_{TelSoft}(ws) &= \{ts1, ts2\}\\
O_{BCS}(ws) &= \{bcs1, bcs2\}\\
O_{VM}(ws) &= \{vm\}\\
TelSoft.BCS(ws) &= \{(ts1, bcs1), (ts2, bcs2)\}\\
TelSoft.VM(ws) &= \{(ts2, vm)\}\\
O_{Subscription}(ws) &= \{sub1, sub2\}\\
Subscription.user(ws) &= \{(sub1, u1), (sub2, u2)\}\\
Subscription.service(ws) &= \{(sub1, ts1), (sub2, ts2)\}\\
O_{UVM}(ws) &= \{uvm\}\\
UVM.user(ws) &= \{(uvm, vmu)\}\\
UVM.vm(ws) &= \{(uvm, vm)\}\\
O_{MailBox}(ws) &= \{mb\}\\
MailBox.vm(ws) &= \{(mb, vm)\}\\
MailBox.message(ws) &= \{(mb, vmsg)\}\\
O_{StartCall}(ws) &= \emptyset\\
StartCall.to(ws) &= \emptyset\\
StartCall.target(ws) &= \emptyset\\
O_{AcceptCall}(ws) &= \emptyset\\
AcceptCall.to(ws) &= \emptyset\\
O_{EndCall}(ws) &= \{ec\}\\
EndCall.to(ws) &= \{(ec, ts1)\}\\
O_{Busy}(ws) &= \emptyset\\
Busy.from(ws) &= \emptyset\\
O_{Available}(ws) &= \emptyset\\
Available.to(ws) &= \emptyset\\
Available.from(ws) &= \emptyset
\end{aligned}$$

Figure 3.12: The world state shown in Figure 3.11 represented using sets and functions

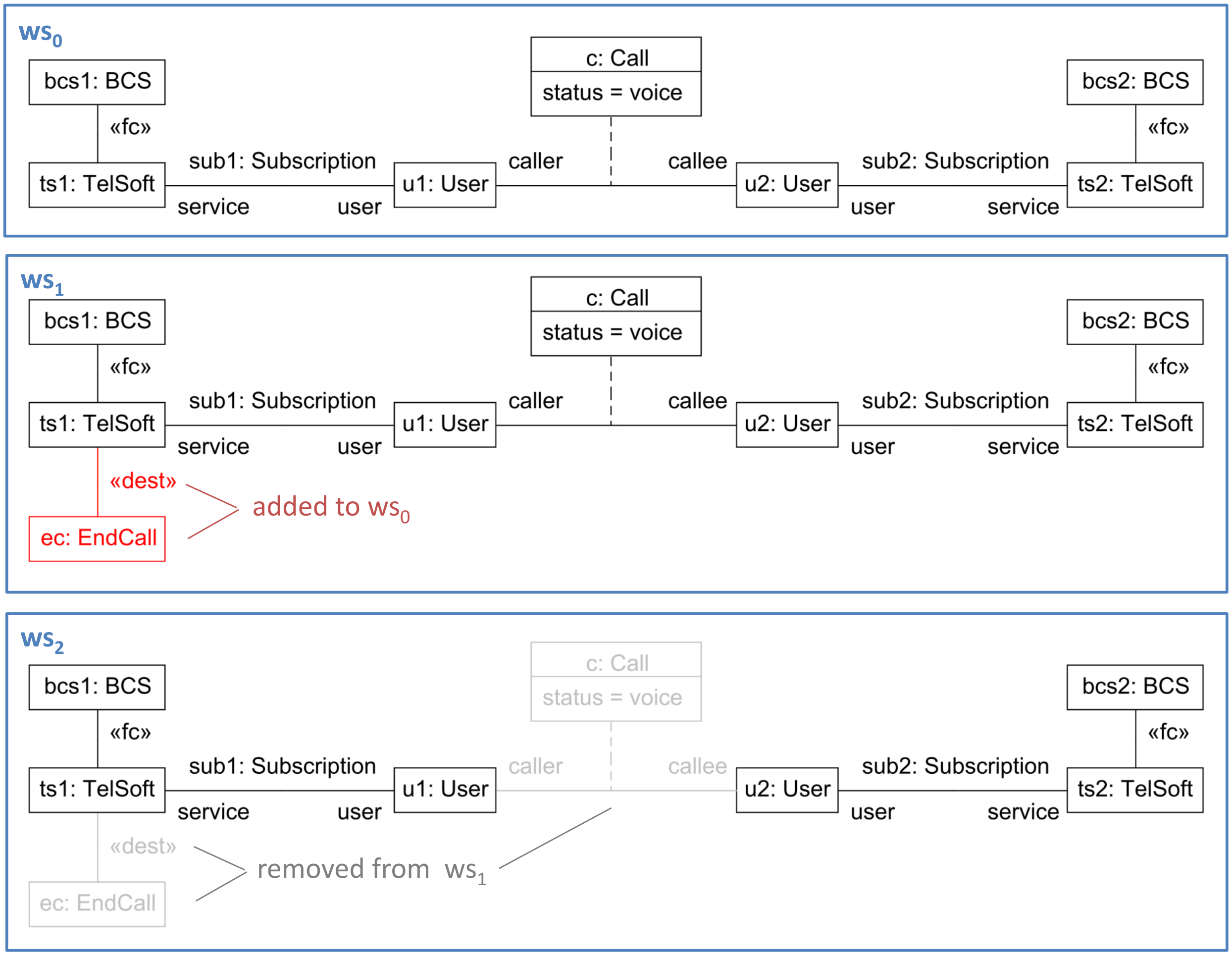


Figure 3.13: A finite world behaviour of the world model shown in Figure 3.13

3.2.5 World Behaviours

Less obviously, a world model provides a declarative description of the world's dynamics: a *world behaviour* is a sequence of world states that represents an evolution of the world starting from some initial world state. A transition from one world state to the next in a world behaviour is called a *world-state transition*; it is the result of applying a set of changes to the first world state, such as adding/removing objects and changing attribute values. We refer to the last world state in an ongoing world behaviour as the *current* world state.

For example, Figure 3.13 shows a short world behaviour comprising the sequence ws_0, ws_1, ws_2 of world states, shown in order from top to bottom. Initially there are two users in a voice-connected call (world state ws_0). Then, the caller makes a request to end the call (the *EndCall* message object ec and its link are added to world state ws_0, resulting in world state ws_1). Finally, in response to the caller's request, the call is ended (the *Call* link c, as well as the transient *EndCall* message object ec and its link, are removed from world state ws_1, resulting in world state ws_2).

More precisely, a world behaviour is an infinite sequence over the set $\boldsymbol{WS_{univ}}$ of possible world states. A world-state transition is a pair of consecutive world states in a world behaviour.

Definition 3.2.5. *We define $\boldsymbol{WB_{univ}}$ to be the set of all possible (valid and invalid) world behaviours over $\boldsymbol{WS_{univ}}$:*

$$\boldsymbol{WB_{univ}} = \boldsymbol{WS_{univ}^{\omega}}$$

where A^{ω} denotes the set of infinite sequences of the elements in A. Hence, a world behaviour $\boldsymbol{wb} \in \boldsymbol{WB_{univ}}$ is an infinite sequence $\boldsymbol{wb}[0], \boldsymbol{wb}[1], \boldsymbol{wb}[2], \ldots,$ where $\boldsymbol{wb}[\boldsymbol{i}] \in \boldsymbol{WS_{univ}}$ for all $i \geq 0$.

Definition 3.2.6. *We define $\boldsymbol{WST_{univ}}$ to be the set of all possible (valid and invalid) world-state transitions over $\boldsymbol{WS_{univ}}$:*

$$\boldsymbol{WST_{univ}} = \boldsymbol{WS_{univ}} \times \boldsymbol{WS_{univ}}$$

Hence, a world-state transition in a world behaviour $\boldsymbol{wb}$ is a pair $\boldsymbol{(wb[i-1], wb[i])}$ for some $i \geq 1$.

3.2.6 World-Model Constraints

Some world states or world behaviours are *invalid*: that is, they represent states or changes in the world that are either *unrealistic*, according to the modeller's knowledge about the SPL's world, or *undesired*, according to the SPL's requirements. For example, a *TelSoft* world state in which a user is the caller of two calls is undesired according to BCS's requirements, and a *TelSoft* world behaviour in which an end-call message persists is unrealistic.

A world model wm can specify *constraints* that *valid* world states and world behaviours must satisfy. Formally, a world-model constraint is a predicate over either a single world state or a pair of consecutive world states (i.e., a world-state transition).

Definition 3.2.7. *Let Con$_1$ denote the set of world-model constraints over singleton world states. We define $\boldsymbol{WS} \subset \boldsymbol{WS_{univ}}$ to be the set of valid world states of wm; that is, the world states that satisfy the constraints in Con$_1$:*

$$\boldsymbol{WS} = \{\boldsymbol{ws} \in \boldsymbol{WS_{univ}} \mid \forall con_1 \in Con_1 : con_1(\boldsymbol{ws})\}$$

Definition 3.2.8. *Let Con$_2$ denote the set of world-model constraints over consecutive world states. We define $\boldsymbol{WST} \subset \boldsymbol{WST_{univ}}$ to be the set of valid world-state transitions of wm; that is, the world-state transitions that satisfy the constraints in Con$_2$:*

$$\boldsymbol{WST} = \{(\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i}) \in \boldsymbol{WST_{univ}} \mid \forall con_2 \in Con_2 : con_2(\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})\}$$

where $i \geq 1$.

Definition 3.2.9. *We define $\boldsymbol{WB} \subset \boldsymbol{WB_{univ}}$ to be the set of valid world behaviours of wm; that is, infinite sequences of valid world states, in which all world-state transitions are valid:*

$$\boldsymbol{WB} = \{\boldsymbol{wb} \in \boldsymbol{WB_{univ}} \mid \boldsymbol{wb} \in \boldsymbol{WS}^{\omega} \ \wedge \ \forall i \geq 1 : (\boldsymbol{wb}[\boldsymbol{i-1}], \boldsymbol{wb}[\boldsymbol{i}]) \in \boldsymbol{WST}\}$$

Several world-model constraints are specified (explicitly or implicitly) by the world-model constructs described in Sections 3.2.1 to 3.2.3. Tables 3.1 and 3.2 list the constraints specified by the concepts and the feature model in a world model, respectively. Table 3.3 lists two additional constraints, on world behaviours, that we adopt by convention. The second constraint in Table 3.3 excludes from the scope of this thesis world behaviours in which the product configuration changes after the initial world state.

Construct	**Constraint**
Entity E_{sub} is a subtype of E_{sup}.	(1) All E_{sub} objects in $\boldsymbol{ws}$ are also E_{sup} objects: $\boldsymbol{ws::O_{E_{sub}}} \subseteq \boldsymbol{ws::O_{E_{sup}}}$
E_{abs} is an abstract entity with subtypes $E_1 \ldots E_n$.	(2) The set of E_{abs} objects in $\boldsymbol{ws}$ comprises the objects of E_{abs}'s subtypes: $\boldsymbol{ws::O_{E_{abs}}} = \bigcup_{1\leq i\leq n} \boldsymbol{ws::O_{E_i}}$
SPL is an SPL concept and F is a feature concept.	(3) An F feature object in $\boldsymbol{ws}$ belongs to exactly one SPL product object: $\forall \boldsymbol{f} \in \boldsymbol{ws::O_F} : \lvert \{\boldsymbol{p} \in \boldsymbol{ws::O_{SPL}} \mid \boldsymbol{ws::SPL.F(p)} = \{\boldsymbol{f}\}\} \rvert = 1$
Role r of association A has multiplicity $mult$ and type C, and A's other roles $r_1 \ldots r_m$ are of types $C_1 \ldots C_m$.	(4) Let $r(\boldsymbol{o_1},\ldots,\boldsymbol{o_m})$ denote the set of different objects in $\boldsymbol{ws}$ that assume role r in A links in which the other objects in the links are $\boldsymbol{o_1} \ldots \boldsymbol{o_m}$ in roles $r_1 \ldots r_m$, respectively: $r(\boldsymbol{o_1},\ldots,\boldsymbol{o_n}) = \{\ o \in ws::O_C \mid$ $\quad \exists\ a \in ws::O_A : ws::A.r(a) = \{o\}\ \wedge \bigwedge_{1\leq i\leq m} ws::A.r_i(a) = \{o_i\}\ \}$ $mult$ constrains the number of such objects, for all fixed sets of objects in roles $r_1 \ldots r_m$: • If $mult$ is a number n, $\forall \boldsymbol{o_1} \in \boldsymbol{ws::O_{C_1}}, \ldots, \boldsymbol{o_m} \in \boldsymbol{ws::O_{C_m}} : \lvert r(\boldsymbol{o_1},\ldots,\boldsymbol{o_n})\rvert = n$ • If $mult$ is a range $n_1..n_2$, $\forall \boldsymbol{o_1} \in \boldsymbol{ws::O_{C_1}}, \ldots, \boldsymbol{o_m} \in \boldsymbol{ws::O_{C_m}} : n_1 \leq \lvert r(\boldsymbol{o_1},\ldots,\boldsymbol{o_n})\rvert \leq n_2$ • If $mult$ is a range $n..*$, $\forall \boldsymbol{o_1} \in \boldsymbol{ws::O_{C_1}}, \ldots, \boldsymbol{o_m} \in \boldsymbol{ws::O_{C_m}} : n \leq \lvert r(\boldsymbol{o_1},\ldots,\boldsymbol{o_n})\rvert$
C is a composition with a whole role wr of type W and part role pr of type P.	(5.1) A whole object, its part objects, and the C composition links between them come into existence and disappear from existence together: $\forall \boldsymbol{c} \in \boldsymbol{ws_{i-1}::O_C} - \boldsymbol{ws_i::O_C}$: $\quad (\boldsymbol{ws_{i-1}::C.wr(c)} \notin \boldsymbol{ws_i::O_W}\ \wedge\ \boldsymbol{ws_{i-1}::C.pr(c)} \notin \boldsymbol{ws_i::O_P}) \ \wedge$ $\forall \boldsymbol{c} \in \boldsymbol{ws_i::O_C} - \boldsymbol{ws_{i-1}::O_C}$: $\quad (\boldsymbol{ws_i::C.wr(c)} \notin \boldsymbol{ws_{i-1}::O_W}\ \wedge\ \boldsymbol{ws_i::C.pr(c)} \notin \boldsymbol{ws_{i-1}::O_P})$ (5.2) There can be at most one C composition link between a single pair of whole and part objects[1]: $\forall \boldsymbol{c_1}, \boldsymbol{c_2} \in \boldsymbol{ws::O_C} : (\boldsymbol{c_1} \neq \boldsymbol{c_2} \implies$ $\quad (\boldsymbol{ws :: C.wr(c_1)} \neq \boldsymbol{ws :: C.wr(c_2)}\ \vee\ \boldsymbol{ws :: C.pr(c_1)} \neq \boldsymbol{ws ::}$ $\boldsymbol{C.pr(c_2)}))$
M is a message.	(6) M objects are transient: $\boldsymbol{ws_{i-1}::O_M} \cap \boldsymbol{ws_i::O_M} = \emptyset$

[1] The constraint that a component object can be related to only one composite object is specified by the whole role's default multiplicity of 1.

Table 3.1: World-model constraints specified by concepts. Constraints (1)-(5.a) are over a single world state $\boldsymbol{ws}$, and constraints (5.1) and (6) are over a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$.

Construct	Constraint
F is a mandatory feature node.	(1) All SPL products should have feature F: $\forall \boldsymbol{p} \in \boldsymbol{ws} :: \boldsymbol{O_{SPL}} : \boldsymbol{ws} :: \boldsymbol{SPL.F(p)} \neq \emptyset$
F_1 is an optional feature node whose parent is the feature node F_2.	(2) Any SPL product that has feature F_1 should also have F_2: $\forall \boldsymbol{p} \in \boldsymbol{ws} :: \boldsymbol{O_{SPL}} :$ $(\boldsymbol{ws} :: \boldsymbol{SPL.F_1(p)} \neq \emptyset) \implies (\boldsymbol{ws} :: \boldsymbol{SPL.F_2(p)} \neq \emptyset)$

Table 3.2: World-model constraints specified by a feature model. SPL is an SPL concept, and F, F_1, and F_2 are feature concepts, referenced by corresponding nodes in the feature model. Both constraints are over a single world state $\boldsymbol{ws}$.

Construct	Constraint
An association A with roles $r_1 \ldots r_n$	(1) The objects related by an A link never change (instead, old links are removed and new links are added): $\forall \boldsymbol{a} \in \boldsymbol{ws_{i-1}} :: \boldsymbol{O_A} \cap \boldsymbol{ws_i} :: \boldsymbol{O_A} : \bigwedge_{1 \leq j \leq n} \boldsymbol{ws_{i-1}} :: \boldsymbol{A.r_j(a)} = \boldsymbol{ws_i} :: \boldsymbol{A.r_j(a)}$
The SPL and feature concepts in the world model	(2) The product configuration (Definition 3.2.4) is static: $PC(\boldsymbol{ws_{i-1}}) = PC(\boldsymbol{ws_i})$

Table 3.3: World-model constraints adopted as conventions. Both constraints are over a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$.

However, many properties of valid world states and world behaviours cannot be (conveniently) expressed using concepts and feature models, such as the following properties related to *TelSoft* 's world:

1. A user cannot be in a voice-connected call with himself.
2. The special voice-mail user cannot initiate calls.
3. A call should end in response to an end-call request from the caller.
4. All users, except for voice-mail users, are subscribed to *TelSoft* products.

A world model includes auxiliary constraints written as predicates in FORML's expression language, described below in Section 3.3. The FORML expression language can express a wider range of constraints than the concepts and feature model in a world model can. For example, the above properties can be expressed as follows:

1. no c: Calls | c.caller = c.callee
2. no c: Calls | c.caller in VoiceMailUsers
3. all c: Calls@pre |
   (c.caller.Subscription.service in EndCalls.to)@pre implies not c in Calls
4. Subscriptions.user = (Users - VoiceMailUsers)

To improve readability, a world-model constraint can be associated with the concept to which it is most relevant. For example, the first three constraints above can be associated with the *Call* concept.

3.2.7 Comparison with UML Class Diagrams

FORML world models adopt the following constructs from UML class diagrams: entities (UML classes), abstract entites, subtype relationships among entities, binary associations, and compositions. In addition, FORML adapts UML class diagrams as follows:

- The «*association*» and «*role*» stereotypes over UML classs and associations, respectively, are used to specify FORML associations with more than two roles.

- The «*SPL*» and «*feature*» stereotypes over UML classes are used to specify SPL and feature concepts, respectively.

- UML operations are used to specify FORML messages; and the «*input*», «*output*», and «*IO*» stereotypes over UML class compartments are used to denote input, output, and IO messages, respectively.

- The «*ctrl*» stereotype over UML classes, associations, and attributes is used to specify controlled phenomena.

- A FORML world model extends a UML class diagram with a traditional feature model.

3.3 Expression Language

FORML has a language for writing *world-model expressions*, which are usually expressions over the elements (objects, attribute values, etc.) of one world state or two consecutive world states (i.e., a world-state transition)[8]. The expression language, which is based on Alloy [48] and OCL [73], can be used to write queries to access world-state elements, as well as to write predicates. The expression language is used not only for writing world-model constraints (see Section 3.2.6), but also for writing expressions in the behaviour model, as described in Section 3.4.

Sections 3.3.1 to 3.3.8 describe different types of world-model expressions and Section 3.3.9 compares these expression types to Alloy and OCL expressions. Three additional types of world-model expressions, which are used only in the behaviour model, are described in Sections 3.4.1.2, 3.4.1.3, and 3.4.3.2. In the following, an expression is typeset in san serif font (e.g., Calls), $<\ldots>$ denotes a placeholder to be replaced by the appropriate type of expression (e.g., $<S>$), optional elements are denoted in gray, and list(x) denotes a comma separated list of elements of the form x (e.g., list($<S>$)).

The formal semantics of an expression e is given as a mathematical expression $\llbracket e \rrbracket(\mathbb{s}, \boldsymbol{B})$ evaluated in a state $\mathbb{s}$ and a set $\boldsymbol{B}$ described below. $\mathbb{s}$ is a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$, where $\boldsymbol{ws_i}$ and $\boldsymbol{ws_{i-1}}$ are the current and previous world states, respectively[9]. Some types of expressions (e.g., selection expressions, quantified predicates)

[8] The behaviour model can include expressions over three world states. Such expressions are described later in Section 3.4.

[9] The state $\mathbb{s}$ will be expanded in Section 3.6 to account for the additional expression types that appear in the behaviour model.

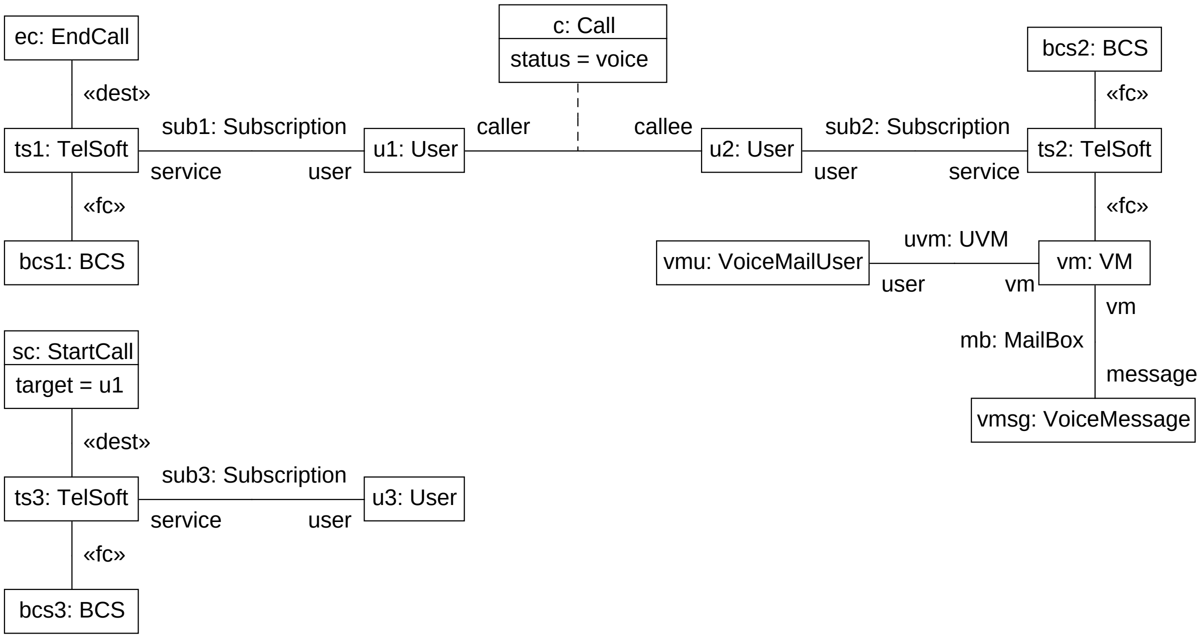


Figure 3.14: An extension of the world state shown in Figure 3.11

introduce *object variables* that are bound to objects in $\boldsymbol{ws_i}$ or $\boldsymbol{ws_{i-1}}$; an object variable can be referenced within subexpressions of its introducing expression. $\boldsymbol{B}$ is a set of bindings of the object variables introduced by e's containing expressions. More precisely, a member of $\boldsymbol{B}$ is a tuple $(\boldsymbol{v}, \boldsymbol{o})$ that binds object variable $\boldsymbol{v}$ to object $\boldsymbol{o}$; $\boldsymbol{B}(\boldsymbol{v_1})$ denotes the value of variable $\boldsymbol{v_1}$ in $\boldsymbol{B}$ (if any):

$$\boldsymbol{B}(\boldsymbol{v_1}) \equiv \{\boldsymbol{o_1} \mid (\boldsymbol{v_1}, \boldsymbol{o_1}) \in \boldsymbol{B}\}$$

and $\boldsymbol{B} \oplus (\boldsymbol{v_2}, \boldsymbol{o_2})$ denotes adding or replacing the binding of variable $\boldsymbol{v_2}$ in $\boldsymbol{B}$, such that $\boldsymbol{v_2}$ is bound to object $\boldsymbol{o_2}$ and all other bindings in $\boldsymbol{B}$ are unchanged:

$$\boldsymbol{B} \oplus (\boldsymbol{v_2}, \boldsymbol{o_2}) \equiv \boldsymbol{B} - \{(\boldsymbol{v_2}, \boldsymbol{o}) \mid (\boldsymbol{v_2}, \boldsymbol{o}) \in \boldsymbol{B}\} \;\cup\; \{(\boldsymbol{v_2}, \boldsymbol{o_2})\}$$

All of the examples in the following subsections are given with respect to the *TelSoft* world model shown in Figure 3.10; its world state shown in Figure 3.14 (an extension of the world state shown in Figure 3.11); and, in the case of `@pre` expressions (Section 3.3.7), its world behaviour shown in Figure 3.13.

Expression	Semantics
none	$\emptyset$
val	The literal constant *val* corresponding to an enumerated value e.g., request evaluates to the value (set) $\{request\}$ in the enumerated type of attribute *status* of association *Call*
Cs	The set of instances (objects) of concept C in $\boldsymbol{ws_i}$: $\boldsymbol{ws_i :: O_C}$ e.g., Users evaluates to the set of *User* objects $\{u1,\ u2\}$, and Calls evaluates to the set of *Call* objects $\{c\}$
v	The value of object variable v: $\boldsymbol{B(v)}$

Table 3.4: Atomic set expressions: The expressions are evaluated in state $\mathbb{s} = \boldsymbol{(ws_{i-1}, ws_i)}$ and set $\boldsymbol{B}$.

3.3.1 Set Expressions

A *set expression* is a query over a world state that returns a set of objects or values[10]. A set expression can be *atomic* or *derived* from subexpressions. An *atomic* expression is either the empty set, a value of an enumerated type, or an object-variable reference (Table 3.4). There are several types of derived set expressions:

Starting from a set of objects in a world state, a *navigation expression* derives related sets of objects or values by *navigating* the relationships among world-state elements (Table 3.5). A *selection expression* filters a set of objects in a world state to derive the subset of members that satisfy some condition (Table 3.6). A *conditional expression* returns one of two sets of objects or values of the same type in a world state, depending on some condition (Table 3.7). Finally, the basic operations of set union (+), intersection (&), and difference (-) can be applied to two sets of objects or values of the same type (Table 3.8).

[10] To simplify the expression language, an individual object (value) is treated as a singleton set of objects (values). In this manner, the modeller does not have to keep track of whether he is writing an expression over an individual object (value) or over a set of objects (values).

Expression	Semantics
$<O>$.a	The set of values of attribute a in $\boldsymbol{ws_i}$ for the objects $\llbracket <O> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$: $\{\boldsymbol{ws_i :: C.a(o)} \mid \boldsymbol{o} \in \llbracket <O> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ e.g., Calls.status evaluates to the value (set) $\{voice\}$, which is the set of the values of the *status* attribute of all *Call* objects
$<O>$.A-r	The set of A links in $\boldsymbol{ws_i}$ that relate, in role r, the objects $\llbracket <O> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$ to other objects: $\{\boldsymbol{a} \in \boldsymbol{ws_i :: O_A} \mid \exists\ \boldsymbol{o} \in \llbracket <O> \rrbracket(\mathbbm{s}, \boldsymbol{B}) : \boldsymbol{ws_i :: A.r(a) = o}\}$ e.g., Users.Call-caller evaluates to the *Call* link (set) $\{c\}$, which is the only *Call* link that relates a *User* object (*u1*) in the *caller* role
$<O>$.A	The set of A links in $\boldsymbol{ws_i}$ that relate the objects $\llbracket <O> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$ (in the implicit role r) to other objects[1]: $\llbracket <O>.\text{A-r} \rrbracket(\mathbbm{s}, \boldsymbol{B})$ e.g., Users.Subscription evaluates to the set of *Subscription* links {*sub1, sub2, sub3*} in which the *User* objects *u1*, *u2*, and *u3* participate
$<L>$.r	The set of objects in $\boldsymbol{ws_i}$ that play role r in the links $\llbracket <L> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$: $\{\boldsymbol{ws_i :: A.r(a)} \mid \boldsymbol{a} \in \llbracket <L> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ e.g., Calls.caller evaluates to the *User* object (set) {*u1*}, which is the only *User* object that plays the *caller* role in all *Call* links
$<P>$.F	The set of F feature objects in $\boldsymbol{ws_i}$ related to the product objects $\llbracket <P> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$: $\{\boldsymbol{ws_i :: SPL.F(p)} \mid \boldsymbol{p} \in \llbracket <P> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ e.g., TelSofts.BCS evaluates to the *BCS* feature object (set) {*bcs1, bcs2, bcs3*} of the *TelSoft* product objects *ts1*, *ts2*, and *ts3*
$<M>$.from and $<M>$.to	The set of product objects in $\boldsymbol{ws_i}$ that the message objects $\llbracket <M> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$ originate from and are destined to, respectively: $\{\boldsymbol{ws_i :: M.from(m)} \mid \boldsymbol{m} \in \llbracket <M> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ and $\{\boldsymbol{ws_i :: M.to(m)} \mid \boldsymbol{m} \in \llbracket <M> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ e.g., EndCalls.to evaluates to the *TelSoft* product object (set) {*ts1*}
$<M>$.p	The set of values of parameter p in $\boldsymbol{ws_i}$ for the message objects $\llbracket <M> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$: $\{\boldsymbol{ws_i :: M.p(m)} \mid \boldsymbol{m} \in \llbracket <M> \rrbracket(\mathbbm{s}, \boldsymbol{B})\}$ e.g., StartCalls.target evaluates to the value (set) {*u1*} of the *target* parameter of the *StartCall* message object *sc*

[1] The expression is ambiguous if the objects $\llbracket <O> \rrbracket(\mathbbm{s},\, \boldsymbol{B})$ can participate in more than one role of A (e.g., *User* objects can participate in both the *caller* and *callee* roles of *Call*). In such cases, the expression $<O>$.A-r can be used.

Table 3.5: Navigation expressions: The expressions are evaluated in state $\mathbbm{s} = \boldsymbol{(ws_{i-1}, ws_i)}$ and set $\boldsymbol{B}$. $<O>$, $<L>$, $<P>$, and $<M>$ are set expressions.

Expression	Semantics
$<O>$ [v | $<P>$]	The subset of the objects $\llbracket <O> \rrbracket(\mathbb{s},\ \boldsymbol{B})$ that satisfy the predicate $<P>$: $\{\boldsymbol{o} \in \llbracket <O> \rrbracket \mid \llbracket <P> \rrbracket(\mathbb{s}, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))\}$ e.g., Users [u | some u.Calls-caller or some u.Calls-callee] evaluates to the subset $\{u1, u2\}$ of *User* objects that participate in one or more *Call* links

Table 3.6: Selection expressions: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<O>$ is a set expression, $<P>$ is a predicate, and v is an object variable that iterates over $\llbracket <O> \rrbracket(\mathbb{s},\ \boldsymbol{B})$.

Expression	Semantics
if $<P>$ then $<S1>$ else $<S2>$	The set of objects $\llbracket <S1> \rrbracket(\mathbb{s},\ \boldsymbol{B})$, if $\llbracket <P> \rrbracket(\mathbb{s},\ \boldsymbol{B})$ is true, and otherwise, the set of objects $\llbracket <S2> \rrbracket(\mathbb{s},\ \boldsymbol{B})$: $\begin{cases} \llbracket <S1> \rrbracket(\mathbb{s}, \boldsymbol{B}) & \text{if } \llbracket <P> \rrbracket(\mathbb{s}, \boldsymbol{B}), \\ \llbracket <S2> \rrbracket(\mathbb{s}, \boldsymbol{B}) & \text{if } \neg \llbracket <P> \rrbracket(\mathbb{s}, \boldsymbol{B}) \end{cases}$ e.g., if (Calls.callee in VoiceMailUsers) then none else Calls.callee evaluates to the *User* object (set) $\{u1\}$, which is the normal (i.e., not a special voice-mail user) *callee* of the *Call* link *c*

Table 3.7: Conditional expressions: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<P>$ is a predicate, and $<S1>$ and $<S2>$ are set expressions.

Expression	Semantics
$<S1>$ + $<S2>$	$\llbracket <S1> \rrbracket(\mathbb{s}, \boldsymbol{B})\ \cup\ \llbracket <S2> \rrbracket(\mathbb{s}, \boldsymbol{B})$ e.g., Calls.caller + Calls.callee evaluates to the set of *User* objects $\{u1,\ u2\}$
$<S1>$ & $<S2>$	$\llbracket <S1> \rrbracket(\mathbb{s}, \boldsymbol{B})\ \cap\ \llbracket <S2> \rrbracket(\mathbb{s}, \boldsymbol{B})$
$<S1>$ - $<S2>$	$\llbracket <S1> \rrbracket(\mathbb{s}, \boldsymbol{B}) - \llbracket <S2> \rrbracket(\mathbb{s}, \boldsymbol{B})$

Table 3.8: Basic set-operation expressions: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<S1>$ and $<S2>$ are set expressions.

3.3.2 Integer Expressions

An *integer expression* returns either an integer constant, the cardinality of a set of objects or values, or the result of adding or subtracting set cardinalities and constants (Table 3.9). Integer expressions are used as subexpressions within predicates.

Expression	**Semantics**
n	The integer n e.g., 2 evaluates to the integer 2
$\#<S>$	$\lvert\llbracket<S>\rrbracket(\mathbb{s},\, \boldsymbol{B})\rvert$ e.g., #Users evaluates to 3
$<I1>$ + $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) + \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$ e.g., #Users + #Calls evaluates to 4
$<I1>$ - $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) - \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$

Table 3.9: Integer expressions: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<S>$ is a set expression, and $<I1>$ and $<I2>$ are integer expressions.

Expression	**Semantics**
$<S1>$ = $<S1>$	$\llbracket<S1>\rrbracket(\mathbb{s}, \boldsymbol{B}) = \llbracket<S2>\rrbracket(\mathbb{s}, \boldsymbol{B})$ e.g., Calls.caller = Calls.callee evaluates to false
$<S1>$ in $<S1>$	$\llbracket<S1>\rrbracket(\mathbb{s}, \boldsymbol{B}) \subseteq \llbracket<S2>\rrbracket(\mathbb{s}, \boldsymbol{B})$ e.g., VoiceMailUsers in Users evaluates to true
$<I1>$ = $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) = \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$ e.g., #Users = #TelSofts evaluates to true
$<I1>$ <> $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) \neq \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$
$<I1>$ < $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) < \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$
$<I1>$ > $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) > \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$
$<I1>$ <= $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) \leq \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$
$<I1>$ >= $<I2>$	$\llbracket<I1>\rrbracket(\mathbb{s}, \boldsymbol{B}) \geq \llbracket<I2>\rrbracket(\mathbb{s}, \boldsymbol{B})$

Table 3.10: Basic set and integer predicates: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<I1>$ and $<I2>$ are integer expressions.

3.3.3 Predicates

A *predicate* is an expression over a world state that evaluates to either *true* or *false*. The most basic types of predicates are the constants true or false with obvious meanings. The other types of predicates are described below.

Predicates can be formed by checking the equality of sets, the inclusion of one set in

Expression	Semantics
no $\langle S\rangle$	$\llbracket\langle S\rangle\rrbracket(\mathbb{s},\ \boldsymbol{B})$ has *no* elements: $\lvert \llbracket\langle S\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \rvert = 0$ e.g., no Calls evaluates to false, because the world state includes the *Call* link c
lone $\langle S\rangle$	$\llbracket\langle S\rangle\rrbracket(\mathbb{s},\ \boldsymbol{B})$ has *zero or one* element: $0 \le \lvert \llbracket\langle S\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \rvert \le 1$
one $\langle S\rangle$	$\llbracket\langle S\rangle\rrbracket(\mathbb{s},\ \boldsymbol{B})$ has *exactly one* element: $\lvert \llbracket\langle S\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \rvert = 1$
some $\langle S\rangle$	$\llbracket\langle S\rangle\rrbracket(\mathbb{s},\ \boldsymbol{B})$ has *one or more* elements: $\lvert \llbracket\langle S\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \rvert \ge 1$

Table 3.11: Set-cardinality predicates: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $\langle S\rangle$ stands for a set expression.

Expression	Semantics
not $\langle P\rangle$	$\neg\llbracket\langle P\rangle\rrbracket(\mathbb{s},\boldsymbol{B})$ e.g., not (#Calls = 0) evaluates to true
$\langle P1\rangle$ and $\langle P2\rangle$	$\llbracket\langle P1\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \ \wedge\ \llbracket\langle P2\rangle\rrbracket(\mathbb{s},\boldsymbol{B})$
$\langle P1\rangle$ or $\langle P2\rangle$	$\llbracket\langle P1\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \ \vee\ \llbracket\langle P2\rangle\rrbracket(\mathbb{s},\boldsymbol{B})$
$\langle P1\rangle$ implies $\langle P2\rangle$	$\llbracket\langle P1\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \ \Rightarrow\ \llbracket\langle P2\rangle\rrbracket(\mathbb{s},\boldsymbol{B})$
$\langle P1\rangle$ iff $\langle P2\rangle$	$\llbracket\langle P1\rangle\rrbracket(\mathbb{s},\boldsymbol{B}) \ \Leftrightarrow\ \llbracket\langle P2\rangle\rrbracket(\mathbb{s},\boldsymbol{B})$

Table 3.12: Basic logical-operation predicates: The expressions are evaluated in state $\mathbb{s} = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $\langle P\rangle$, $\langle P1\rangle$, and $\langle P2\rangle$ are predicates.

another, and integer comparisons involving set cardinalities (Table 3.10). Common predicates about the cardinality of a set in the world state – namely, that a set contains zero (no), zero or one (lone), one (one), or some elements (some) – can be expressed using *set-cardinality predicates* (Table 3.11). A predicate can be formed by applying the negation, conjunction, disjunction, implication, and equivalence logical operators to operand predicates (Table 3.12). Finally, a *quantified predicate* asserts that a predicate $\langle P\rangle$ is satisfied by some number of a set's members; that is, zero, zero or one, one, some, or all members (Table 3.13).

Expression	Semantics
no v: $<S> \mid <P>$	*No* member of $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$ satisfies $<P>$: $\mid \{\boldsymbol{o} \in \llbracket <S> \rrbracket(\$, \boldsymbol{B}) \mid \llbracket <P> \rrbracket(\$, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))\} \mid = 0$ e.g., no c: Calls $\mid$ c.caller = c.callee evaluates to true
lone v: $<S> \mid <P>$	*At most one* member of $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$ satisfies $<P>$ $0 \leq \mid \{\boldsymbol{o} \in \llbracket <S> \rrbracket(\$, \boldsymbol{B}) \mid \llbracket <P> \rrbracket(\$, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))\} \mid \leq 1$
one v: $<S> \mid <P>$	*Exactly one* member of $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$ satisfies $<P>$ $\mid \{\boldsymbol{o} \in \llbracket <S> \rrbracket(\$, \boldsymbol{B}) \mid \llbracket <P> \rrbracket(\$, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))\} \mid = 1$
some v: $<S> \mid <P>$	*One or more* members of $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$ satisfy $<P>$ $\exists\ \boldsymbol{o} \in \llbracket <S> \rrbracket(\$, \boldsymbol{B}) : \llbracket <P> \rrbracket(\$, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))$
all v: $<S> \mid <P>$	*All* members of $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$ satisfy $<P>$ $\forall\ \boldsymbol{o} \in \llbracket <S> \rrbracket(\$, \boldsymbol{B}) : \llbracket <P> \rrbracket(\$, \boldsymbol{B} \oplus (\boldsymbol{v}, \boldsymbol{o}))$

Table 3.13: Quantified predicates: The expressions are evaluated in state $\$ = (\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ and set $\boldsymbol{B}$. $<S>$ is a set expression, $<P>$ is a predicate, and v is an object variable that iterates over $\llbracket <S> \rrbracket(\$,\ \boldsymbol{B})$.

Expression	Semantics
U(list($<S>$))	The type T of an undefined expression U is determined by the context in which U is used. U evaluates to a nondeterministically chosen value of type T. e.g., in Calls[status = status()], status() evaluates to a nondeterministically chosen value from the enumerated type *{request, connected, voice}* of the *status* attribute of the *Call* association; and in if timeout() then Calls[status = request] else Calls[status = connected], timeout() evaluates to a nondeterministically chosen truth value

Table 3.14: Undefined set and predicate expressions. $<S>$ is a set expression.

3.3.4 Undefined Set and Predicate Expressions

An *undefined set expression* is an abstraction that represents a set of objects or values resulting from an undefined computation over a world state. Analogously, an *undefined predicate* represents an undefined condition over a world state. Such expressions are particularly useful for modelling computations and conditions involving world phenomena that are abstracted away in the world model (e.g., time, the undefined types of some attributes). An undefined set or predicate expression can (optionally) take as arguments one or more sets of objects or values in the world state (see Table 3.14).

Macro	Semantics
let m = $<E>$	defines a macro m, which stands for expression $<E>$ e.g., using the macro defined by let callers = Calls.caller, the expression Calls.caller.Subscription.service can be simplified to callers.Subscription.service

Table 3.15: Macros: $<E>$ is an expression.

3.3.5 Parenthesized Expressions

The subexpressions in an expression can be parenthesized to indicate a particular order for evaluating the subexpressions. For example, over the world state shown in Figure 3.14

(#Calls = 2 and #Users = 3) or #TelSofts = 3

returns true (since and is evaluated before or), whereas

#Calls = 2 and (#Users = 3 or #TelSofts = 3)

returns false (since or is evaluated before and).

3.3.6 Macros

A *macro* is an abbreviation defined for an expression, which enables simplifying occurrences of the expression by using the macro instead (see Table 3.15).

3.3.7 @pre

The suffix @pre (borrowed from OCL [73]) can be applied to atomic set expressions of the form Cs and to navigation expressions to indicate that the attributed expressions are evaluated with respect to the previous world state (Table 3.16). Furthermore, @pre can be applied to a parenthesized expression or macro (provided that the paranethesized expression or macro does not have @pre subexpressions), in which case @pre distributes over the subexpressions in the parenthesized expression or macro. For example, the expression

(c.caller.Subscription.service in EndCalls.to)@pre

is equivalent to the expression

c.caller@pre.Subscription@pre.service@pre in EndCalls@pre.to@pre

Expression	Semantics
Cs@pre	The set of instances (objects) of concept C in $\boldsymbol{ws_{i-1}}$: $\boldsymbol{ws_{i-1}::O_C}$ e.g., In world state ws_2 in Figure 3.13, Calls@pre evaluates to the *Call* object (set) $\{c\}$
$<O>$.a@pre	The set of values of attribute a in $\boldsymbol{ws_{i-1}}$ for the objects $[\![<O>]\!](\$, \boldsymbol{B})$: $\{\boldsymbol{ws_{i-1}::C.a(o)} \mid \boldsymbol{o} \in [\![<O>]\!](\$, \boldsymbol{B})\}$
$<O>$.A-r@pre	$\{\boldsymbol{a} \in \boldsymbol{ws_{i-1}::O_A} \mid \exists\, \boldsymbol{o} \in [\![<O>]\!](\$, \boldsymbol{B}) : \boldsymbol{ws_{i-1}::A.r(a) = o}\}$
$<O>$.A@pre	$[\![<O>.\text{A-r@pre}]\!](\$, \boldsymbol{B})^1$
$<L>$.r@pre	$\{\boldsymbol{ws_{i-1}::A.r(a)} \mid \boldsymbol{a} \in [\![<L>]\!](\$, \boldsymbol{B})\}$
$<P>$.F@pre	$\{\boldsymbol{ws_{i-1}::SPL.F(p)} \mid \boldsymbol{p} \in [\![<P>]\!](\$, \boldsymbol{B})\}$
$<M>$.from@pre and $<M>$.to@pre	$\{\boldsymbol{ws_{i-1}::M.from(m)} \mid \boldsymbol{m} \in [\![<M>]\!](\$, \boldsymbol{B})\}$ and $\{\boldsymbol{ws_{i-1}::M.to(m)} \mid \boldsymbol{m} \in [\![<M>]\!](\$, \boldsymbol{B})\}$
$<M>$.p@pre	$\{\boldsymbol{ws_{i-1}::M.p(m)} \mid \boldsymbol{m} \in [\![<M>]\!](\$, \boldsymbol{B})\}$

[1] The expression is ambiguous if the objects $[\![<O>]\!](\$, \boldsymbol{B})$ can participate in more than one role of A. In such cases, the expression $<O>$.A-r can be used.

Table 3.16: @pre expressions: The expressions are evaluated in state $\$ = \boldsymbol{(ws_{i-1}, ws_i)}$ and set $\boldsymbol{B}$. $<O>$, $<L>$, $<P>$, and $<M>$ are set expressions.

3.3.8 Precedence

Table 3.17 gives the default precedence order for evaluating expressions. Parentheses can be used to override the default precedence order.

Set expressions
@pre and @curri (see Section 3.4.3.2) expressions
Navigation and selection expressions
Basic set-operation expressions - set intersection (&) - set union (+), set difference (-)
Integer expressions
Set cardinality expressions
Basic integer-operation expressions
Predicates
Basic set and integer predicates, and set cardinality predicates
Basic logical operations - Negation (nor) - Conjunction (and) - Implication (implies), equivalence (iff) - Disjunction (or)
Quantified predicates

Table 3.17: Default precedence order for expressions: Expressions are listed in decreasing order of precedence for each category (set, integer, predicate) of expressions. All binary operators (i.e., basic set, integer, and logical operators) are left associative.

3.3.9 Comparison with Alloy and OCL

FORML's expression language adopts from Alloy most atomic expressions (see exception below), conditional expressions, basic set-operation expressions, integer expressions, and predicates; and adopts from OCL several navigation expressions (see exceptions below) and @pre expressions. FORML extends and adapts OCL and Alloy as follows:

- FORML introduces atomic expressions of the form Cs to reference the set of objects of a given type C in the current world state.

- FORML introduces the following navigations expressions: $<O>$.A-r expressions to navigate links starting from a particular role; $<P>$.F expressions to navigate from products to their features; and $<M>$.from, $<M>$.to, and $<M>$.p expressions to navigate from message objects to their senders, receivers, and parameters, respectively.

- FORML introduces undefined set and predicate expressions.

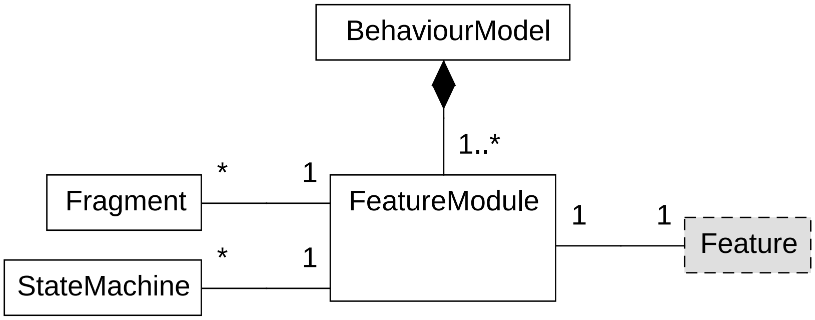


Figure 3.15: A partial behaviour-model metamodel. The referenced class *Feature* is defined in Figure 3.1.

- FORML adapts selection expressions in OCL to use a more compact notation.

3.4 Behaviour Model

As explained in Section 3.2, some of the requirements of an SPL can be modelled *declaratively* as world-model constraints. However, the main view in a FORML model for describing an SPL's requirements is the **behaviour model**. The behaviour model is executable: each execution step of the behaviour model specifies desired changes to world phenomena and the step-by-step changes over the course of an execution realize a *desired* world behaviour (according to the SPL's requirements).

A behaviour model can be described more precisely in terms of the partial metamodel shown in Figure 3.15.

Definition 3.4.1. *A behaviour model is structured in terms of features: the requirements of each feature of an SPL are localized in a separate feature module. Requirements of a feature that are independent of existing features are expressed as a set of parallel state machines. If the feature enhances (i.e., extends or modifies) existing features, the enhancements are expressed as a set of state-machine fragments that extend existing feature modules.*

This section describes the syntax of a behaviour model with an informal description of the semantics. FORML state machines are described in Section 3.4.1, and state-machine fragments are described in Section 3.4.6. The semantics of a behaviour model is formally defined in Section 3.6.

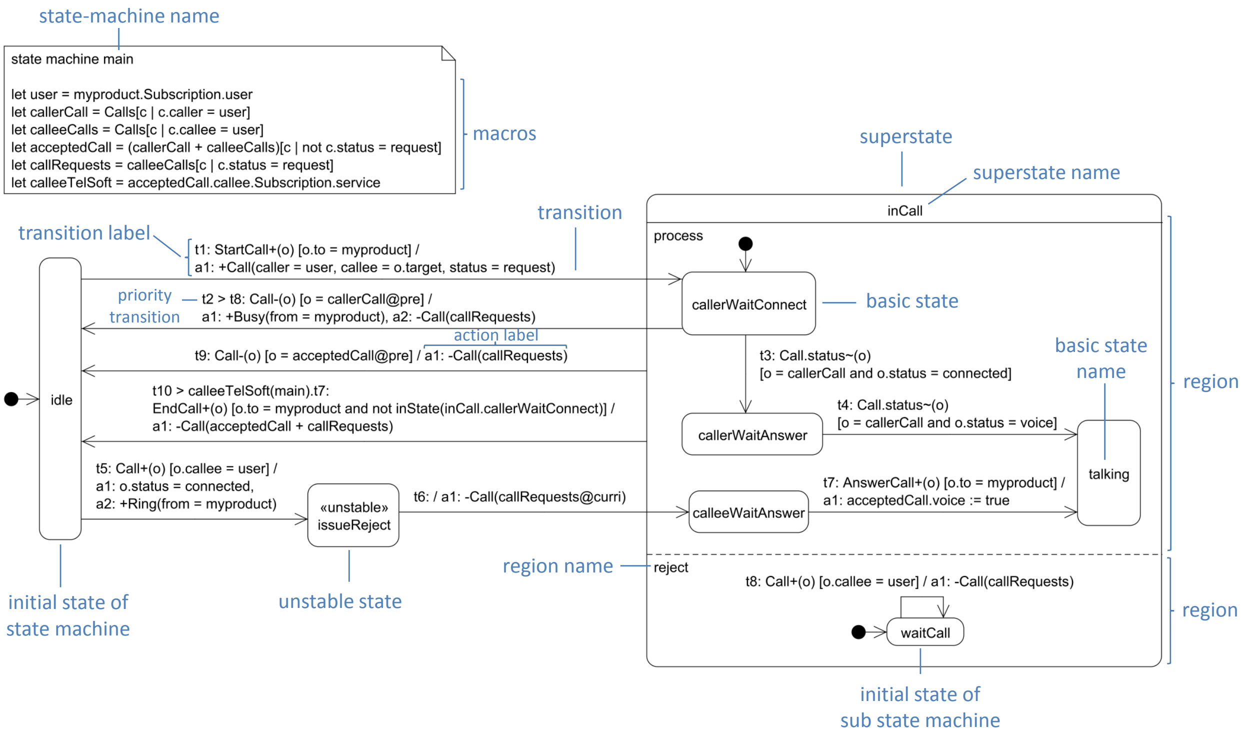

state-machine name
state machine main
let user = myproduct.Subscription.user
let callerCall = Calls[c | c.caller = user]
let calleeCalls = Calls[c | c.callee = user]
let acceptedCall = (callerCall + calleeCalls)[c | not c.status = request]
let callRequests = calleeCalls[c | c.status = request]
let calleeTelSoft = acceptedCall.callee.Subscription.service
macros
superstate
superstate name
inCall
transition
transition label
t1: StartCall+(o) [o.to = myproduct] /
a1: +Call(caller = user, callee = o.target, status = request)
process
callerWaitConnect
basic state
priority
transition
t2 > t8: Call-(o) [o = callerCall@pre] /
a1: +Busy(from = myproduct), a2: -Call(callRequests)
action label
t9: Call-(o) [o = acceptedCall@pre] / a1: -Call(callRequests)
basic state
name
t3: Call.status~(o)
[o = callerCall and o.status = connected]
region
idle
t10 > calleeTelSoft(main).t7:
EndCall+(o) [o.to = myproduct and not inState(inCall.callerWaitConnect)] /
a1: -Call(acceptedCall + callRequests)
callerWaitAnswer
t4: Call.status~(o)
[o = callerCall and o.status = voice]
talking
t5: Call+(o) [o.callee = user] /
a1: o.status = connected,
a2: +Ring(from = myproduct)
«unstable»
issueReject
t6: / a1: -Call(callRequests@curri)
t7: AnswerCall+(o) [o.to = myproduct] /
a1: acceptedCall.voice := true
calleeWaitAnswer
region name
reject
t8: Call+(o) [o.callee = user] / a1: -Call(callRequests)
region
waitCall
initial state of
state machine
unstable state
initial state of
sub state machine

Figure 3.16: The feature module for the BCS feature of *TelSoft*

3.4.1 FORML State Machines

Definition 3.4.2. *A FORML state machine is a tuple* $(n, M, S, R, S_H, S_0, L, T)$ *where:*

- *n is the state-machine's name.*
- *M is a set of macro definitions. The macros defined in M are abbreviations of FORML expressions that are used to simplify transition labels described below.*
- *S and R are finite sets of states and orthogonal regions, respectively, organized into a state hierarchy defined by the containment relation S_H over states and regions.*
- *$S_0 \subset S$ is the set of initial states.*
- *L is a set of transition labels. A transition label specifies the transition's name, trigger, guard, and actions; and the transition's priorities relative to other transitions.*

- *T is a set of transitions between states. A transition $t \in T$ is a tuple (s_{src}, l, s_{dst}), where $s_{src} \in S$ is the transition's source state, $s_{dst} \in S$ is the transition's destination state, and $l \in L$ is the transition's label.*

A FORML state machine is expressed in a notation based heavily on that of UML state machines [71]. Section 3.4.5 summarizes what FORML borrows from, and how it extends and adapts, UML state machines. For example, *TelSoft*'s BCS feature module is shown in Figure 3.16. The name of a state machine is specified as a state-machine declaration in a UML note (e.g., *state machine main*). The macros of a state machine are also defined in a UML note (e.g., *user*). The state hierarchy of a state machine is described in Section 3.4.1.1, and state-machine transitions and actions are described in Sections 3.4.1.2, 3.4.1.3, and 3.4.4.

A state machine of a feature is instantiated for every product in the world to reflect the feature's contribution to that product's effects on the world. Each state-machine instance is called a *machine*. A desired world behaviour is specified by the execution of all the products' features' machines. In each step of an execution, the machines execute a set of concurrently enabled transitions. When a transition executes, it performs a set of enabled actions that specify how the current world state should change. The enabledness of a machine's transitions and actions depends on the current and past world states. For example, the short world behaviour shown in Figure 3.17 is specified by an execution of two instances $smi1$ and $smi2$ of BCS's state machine *main* (Figure 3.16), corresponding to two $TelSoft$ product objects $ts1$ and $ts2$, respectively. Each state machine has an internal object variable named *myproduct* that relates each of its instances to the corresponding product object in the world state[11]. Thus, the values of the *myproduct* variables of $smi1$ and $smi2$ are $ts1$ and $ts2$, respectively. Sections 3.4.2 and 3.4.3 describe machine executions in more detail.

[11] The *myproduct* object variable is ubiquitous and, as such, is not explicitly declared in the state-machine models.

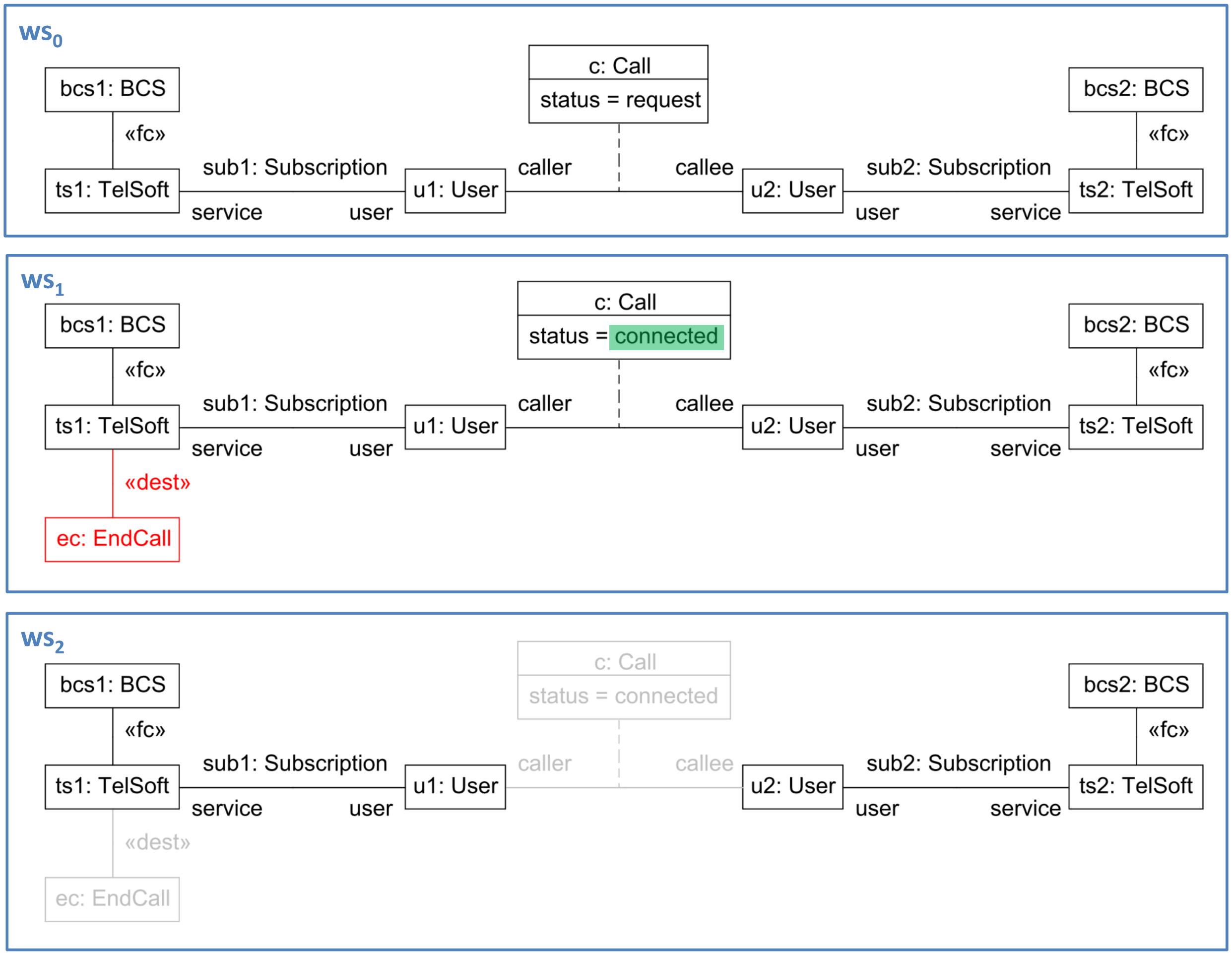


Figure 3.17: A world behaviour of the *TelSoft* world model shown in Figure 3.10: Added elements are coloured red, removed elements are coloured gray, and new attribute values are shaded green.

3.4.1.1 States

A *state* of a FORML state machine is an abstraction that represents a class of past world behaviours. For example, the *inCall* state in Figure 3.16 represents the set of world behaviours that end in a (current) world state in which the *TelSoft* user participates in a call. Following the UML, each state is either a *superstate*, which contains other states (e.g., *inCall*), or a *basic state*, which contains no other states (e.g., *idle*). A superstate contains one or more *orthogonal regions* (*regions* for short), where each region models a concurrent sub-machine. For example, superstate *inCall* includes two regions *process* and *reject*, which model the processing of a call and the rejection of additional incoming calls, respectively. The containment relationship between states and regions defines a *state hierarchy*, where

- The root of the state hierarchy represents the state machine. Its child nodes are the *top-level states*: that is, the states that are not internal to any sub-machine.
- A superstate has one or more child regions.
- A region's child nodes are the states of its sub-machine.
- A basic state is a leaf node.

The remainder of this thesis makes use of the following definitions to access information about a state hierarchy:

Definition 3.4.3. *The ancestors of a state or region x are all of the nodes along the path from the root to x. The descendants of a node x are all of the states and regions in the subtrees of x. The rank of a node x in the state hierarchy is the length of the path from the root to x. The nearest common ancestor of a state or region x_1 and a state or region x_2 is the maximum-rank node that has both x_1 and x_2 as descendants. A region r_1 is orthogonal to a region r_2 if the nearest common ancestor of r_1 and r_2 is a state.*

A state or region x of a state machine *sm* is uniquely identified within its feature module by its *global name*, which is the dot-separated list of node names along the path from the root node of *sm*'s state hierarchy to x's corresponding node. For example, in Figure 3.16, the global name *main.inCall.process.talking* refers to state *talking* in region *process* in state *inCall* of state machine *main*. x can also be identified by its *local name* relative to some containing state machine, superstate, or region y; the local name is derived from the path from y's node in the state hierarchy to x's node. For example, the equivalent local name, relative to state *inCall*, of the global name *main.inCall.process.talking* is *process.talking*.

One state of a state machine is designated as the *initial state* (e.g., *idle*); similarly, one state of every sub-machine can (optionally) be designated as the initial state (e.g., *waitCall* in region *reject*). The remainder of this thesis uses the following definition to identify all regions and their initial states that are descendants of a given state:

Definition 3.4.4. *The initial descendants of state s are all of s's regions (if any), the initial state of each of s's regions, and the initial descendants of each of those initial states.*

Lastly, there is a distinction between basic states that are *stable* (e.g., *idle*) and those that are *unstable* (e.g., *issueReject*). An unstable state is denoted by the stereotype «*unstable*». Briefly, the current world state is observable in a stable state, but is unobservable in an unstable state.

Definition 3.4.5. *An unstable state is a basic state that marks an intermediate stage in computing the next observable world state.*

The distinction between stable and unstable states is described in more detail in Section 3.4.3.

A state machine defines the set of possible *configurations* of its instances.

Definition 3.4.6. *A configuration of a machine is the set of current states and regions of its execution. More precisely, a configuration c of a state-machine sm is a subset of the states and regions in sm's state hierarchy that satisfies the following constraints:*

- *Exactly one child of the root is in c.*
- *If a superstate is in c, so are all of its children (regions).*
- *If a region is in c, so is exactly one of its children (a state).*
- *If a state is in c, so are all of its ancestors, except the root.*

The set of sm's possible configurations is denoted by $Config_{sm}$.

For example, the state machine *main* in Figure 3.16 has six possible configurations including the following three configurations:

$\{idle\}$
$\{issueReject\}$
$\{inCall, process, reject, callerWaitConnect, waitCall\}$

The *initial configuration* c_{init} of a state machine *sm* is the configuration in which each *sm* machine (instance) starts execution. The initial state of *sm* is in c_{init}, and if a region is in c_{init}, so is its initial state. For example, the initial configuration of the state machine in Figure 3.16 is $\{idle\}$. Finally, a configuration is *stable* if it includes only stable basic states and is *unstable* if it includes one or more unstable basic states. The initial configuration of a state machine must be stable.

3.4.1.2 Transitions

A transition in a FORML state machine specifies a change in the machine's configuration and a set of actions on the world. The most common type of transition is a *non-preemptive* transition, which unlike *preemptive* transitions (see Section 3.4.4), does not have a higher priority than other transitions. A non-preemptive transition has a label of the form

$$id : te\ [gc]\ /\ al_1,\ \ \ldots,\ \ al_n$$

where *id* is the name of the transition, *te* is an optional trigger expression, *gc* is an optional guard condition, and $al_1 \ldots al_n$ are labels (described in detail below) that specify a set of concurrent actions. A non-preemptive transition t in machine *smi* is *enabled* for execution if the configuration of *smi* includes t's source state and, as described below, t's trigger and guard evaluate to true.

As expected, transitions in different concurrent regions can be concurrently enabled. However, it is also possible for *nonconcurrent* transitions to (unexpectedly) be enabled together.

Definition 3.4.7. *Two transitions t1 and t2 of state machine are nonconcurrent if either they have the same source state, or the source state of one is a descendant of the other.*

For example, in Figure 3.16, transitions *t1* and *t5*, and transitions *t2* and *t9*, are nonconcurrent. For now, we assume that nonconcurrent transitions cannot be concurrently enabled; we relax this assumption in Section 4.1.

The trigger expression of a transition is a world-change event (WCE): a predicate over two consecutive world states that identifies a single primitive change in the world state. There are three types of WCEs: (1) a new object in the world, (2) an object removed from the world, and (3) a change in value of an object attribute (see Table 3.18). A trigger expression is evaluated together with the transition's guard: a trigger evaluates to true if its WCE occurs *and* the guard evaluates to true with respect to the (new, removed, changed) object that raised the WCE. The guard refers to the object associated with a WCE using the object variable introduced by the trigger. This object variable can also be referenced by the transition's actions. If the trigger expression and guard are satisfied by more than one object's WCE (e.g., if multiple objects are added to the world at once), one object is chosen nondeterministically, so that any transition actions that apply to the trigger's object are applied to only one object in the world.

The guard of a transition is a predicate that is normally over the most recent world-state transition. A guard condition is usually used to check a property about the current

Expression	Semantics
C+(o)	Some C object $\boldsymbol{c}$ was added to world state ws_{i-1}, for which $[\![<gc>]\!](\mathbb{s}, \{(\text{o}, \boldsymbol{c})\})$ is true e.g., over the consecutive world states *ws0* and *ws1* in Figure 3.17, where the transition has no guard, EndCall+(o) evaluates to true for $o = ec$, where ec is only added *EndCall* message object
C-(o)	Some C object $\boldsymbol{c}$ was removed from world state ws_{i-1}, for which $[\![<gc>]\!](\mathbb{s}, \{(\text{o}, \boldsymbol{c})\})$ is true e.g., over the consecutive world states *ws1* and *ws2* in Figure 3.17, where gc is the predicate o.status@pre = connected, Call-(o) evaluates to true for $o = c$, where c is the only removed *Call* link whose *status* attribute has the value connected
C.a~(o)	The a attribute of some C object $\boldsymbol{c}$ in world state ws_{i-1}, for which $[\![<gc>]\!](\mathbb{s}, \{(\text{o}, \boldsymbol{c})\})$ is true, changed value e.g., over the consecutive world states *ws0* and *ws1* in Figure 3.17, where the transition has no guard, Call.status~(o) evaluates to true for $o = c$, where c is the only *Call* link whose *status* attribute changed value (from *request* in *ws0* to *connected* in *ws1*).

Table 3.18: Trigger expressions: $<gc>$ is the transition's guard condition, and o is an object variable that refers to the (new, removed, changed) objects in the WCE. The expressions are evaluated in a state $\mathbb{s}$ that includes a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$ $(i \geq 1)$.

Expression	Semantics
$<p>$(sm).inState(s)	The configuration of the sm machine of product $[\![<p>]\!](\mathbb{s}, \boldsymbol{B})$ includes state s.
sm.inState(s)	The configuration of the sm machine that belongs to the same product as *this* machine (i.e., the machine for which the predicate is evaluated), includes state s.
inState(s)	The configuration of this machine includes state s.

Table 3.19: *inState* predicates: s is a state's local name within its state machine, and $<p>$ is a set expression that evaluates to a singleton set (if $<p>$ does not evaluate to a singleton set, the *inState* predicate evaluates to true). $<p>$ is evaluated in a state $\mathbb{s}$ that includes a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$ $(i \geq 1)$, and in a set $\boldsymbol{B}$ of object-variable bindings.

world state. However, there are cases where the desired behaviour depends on how the world has changed (e.g., how a vehicle's speed has changed). Hence, the guard condition is over both the current world state and the previous world state (i.e., the most recent world-state transition) rather than just the current world state[12]. However, as with Harel statecharts [43], the guard can also check whether a machine is in a particular state. Such checks, used to synchronize the execution of different machines, are expressed as *inState* predicates (Table 3.19). For example, the expression inState(inCall.process.talking) checks whether an instance of state machine *main* of BCS (Figure 3.16) is in state *talking*. The syntax for referring to states (e.g., inCall.process.talking) and other machine elements are described in Section 3.4.1.4.

3.4.1.3 Actions

An action specifies a constraint on how the current world state should change (into the next world state). Based on the usual interpretation of state-machine actions, the reader may expect an action to be interpreted as a change to a single part of the world state (such that the rest of the world state is left unchanged). In contrast, we interpret an action as a constraint on how the current world state should change, allowing the unconstrained part of the world state to change arbitrarily. In this manner, state-machine actions naturally complement the world model's constraints on world-state transitions.

The most common type of action is a *non-preemptive* action, which unlike *preemptive* actions (see Section 3.4.4), does not have a higher priority than other actions. A non-preemptive action has a label of the form

$$id : [gc]\ a$$

where *id* is the name of the action; *gc* is an optional guard condition; and *a* is a WCA, described below. A non-preemptive action is enabled as part of the execution of its containing transition if its guard condition evaluates to true.

A WCA is a predicate over two consecutive world states that specifies a primitive change to the current world state. There are three types of WCAs: (1) a new entity object, link, or output or IO message object is to be created and added to the world state; (2) a set of objects are to be removed from the world state; (3) an object's attribute value is to be set to a given value. Table 3.20 gives the syntax and informal semantics of WCAs. For example, a WCA of the form

$$\mathsf{var} = + \mathsf{A}(list(\mathsf{r} = <o>),\ list(\mathsf{a} = <val>))$$

[12] Section 3.4.3 describes cases where guard conditions can refer to a third world state.

Expression	**Semantics**
var = +E(list(a = $<val>$))	A new entity object of type E is to be added to world state ws_i, with each attribute a of the new object initialized to a corresponding value $\llbracket <val> \rrbracket(\mathbb{s},\ \boldsymbol{B})$. E is (optionally) referenced by object variable *var*.
var = +A(list(r = $<o>$) , list(a = $<val>$))	A new A link is to be added to world state ws_i, with each role r and attribute a of the new link initialized to a corresponding object $\llbracket <o> \rrbracket(\mathbb{s},\ \boldsymbol{B})$ and value $\llbracket <val> \rrbracket(\mathbb{s},\ \boldsymbol{B})$, respectively. A is (optionally) referenced by object variable *var*.
+MO(from = $<p>$, list(param = $<e>$))	A new output message object of type MO is to be added to world state ws_i, with each parameter *param* of the new message object set to a corresponding value $\llbracket <e> \rrbracket(\mathbb{s},\ \boldsymbol{B})$. The new message object is generated by the product object $\llbracket <p> \rrbracket(\mathbb{s},\ \boldsymbol{B})$.
+MIO(to = $<p1>$, from = $<p2>$, list(param = $<e>$))	Analogous to the +MO WCA above, but for IO messages
-C($<O>$)	The C objects $\llbracket <O> \rrbracket(\mathbb{s},\ \boldsymbol{B})$ are to be removed from world state ws_i. The objects to be removed cannot be product or feature objects. E.g., the consecutive world states *ws1* and *ws2* in Figure 3.17 satisfy the WCA Call-(Calls).
$<o>$.a := $<v>$	The value of attribute a of object $\llbracket <o> \rrbracket(\mathbb{s},\ \boldsymbol{B})$ in world state ws_i is to be set to $\llbracket <v> \rrbracket(\mathbb{s},\ \boldsymbol{B})$. E.g., the consecutive world states *ws0* and *ws1* in Figure 3.17 satisfy the WCA Calls.status := connected.

Table 3.20: World change actions: $<o>$, $<val>$, and $<p>$ are set expressions that return a single object, value, and product object, respectively; $<e>$ is a set expression that returns a single object or value; $<O>$ is a set expression that returns a set of objects. The aforementioned expressions are evaluated in a state $\mathbb{s}$ that includes a pair of consecutive world states $(\boldsymbol{ws_{i-1}}, \boldsymbol{ws_i})$ $(i \geq 1)$, and in a set $\boldsymbol{B}$ of object-variable bindings.

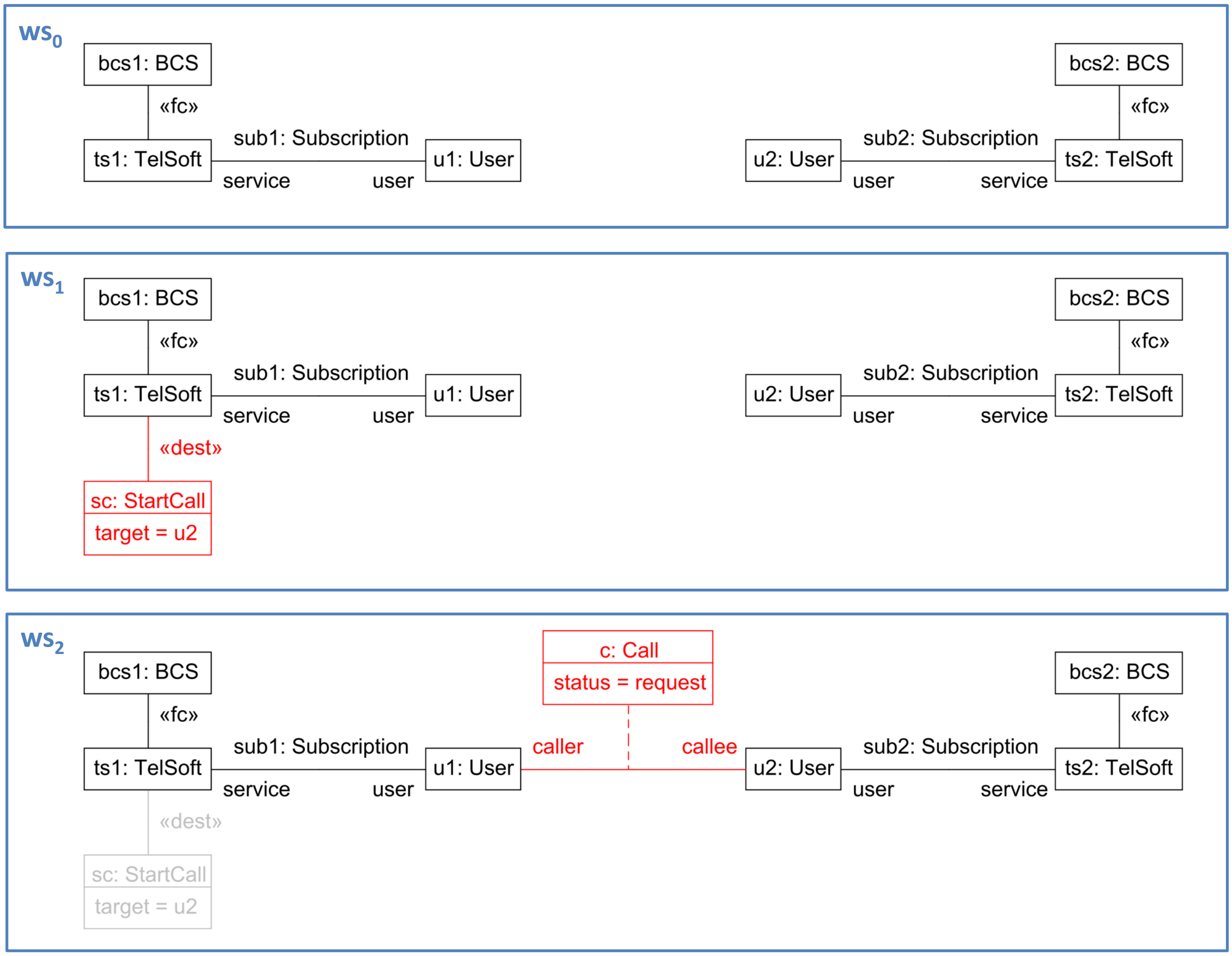


Figure 3.18: A world behaviour of the *TelSoft* world model shown in Figure 3.10: added elements are coloured red and removed elements are coloured gray.

specifies the creation of a new link of type A to be added to the world state. The object that participates in each role r of the new link and the value of each attribute a of the new link (if A has any attributes) are listed as name-value pairs in the WCA. Finally, a reference to the new link can optionally be stored in an object variable var. As an example of such a WCA,

$$\mathsf{Call} + (\mathsf{caller} = \mathsf{StartCalls.to.Subscription.user}, \mathsf{callee} = \mathsf{StartCalls.target}, \mathsf{status} = \mathsf{request})$$

is satisfied by the consecutive world states *ws1* and *ws2* in Figure 3.18, since the *Call* link c between $u1$ and $u2$ is to be added to *ws1* (as per the *StartCall* request made by telephone user $u1$).

Note that as per the conventions described in Section 3.2.6, there are no WCA types for adding or removing product and feature objects, for changing the relationships between product and feature objects, or for changing the role values of links. Also, there is no WCA type for adding new input message objects because the behaviour model specifies only an SPL's requirements, and the creation of input messages is solely controlled by the environment.

3.4.1.4 Referencing Machine Elements

Several FORML constructs reference states, transitions, or actions of state-machine instances (machines). For example, *inState* predicates (Table 3.19) reference states that are in the same machine as the *inState* predicate, in another machine of the same product, or in a machine of a different product. Constructs that reference machine transitions and actions are described in Section 3.4.4.

An element that is in the same machine as the expression being evaluated is referenced simply by its name. In the case of a state, it can be referenced by its local name within its state machine. When referencing a state or transition of another machine, a prefix is used to identify the machine[13]: the prefix sm. identifies the machine of type *sm* that is in the same product as the predicate, and the prefix $<p>$(sm). identifies the machine of type *sm* of product $<p>$[14]. When referencing a transition, the prefix is applied to the transition name. For example, $<p>$(sm).t references transition t in the *sm* machine of product $<p>$. However, when referencing a state in an *inState* predicate, the prefix is applied to the predicate itself to improve readability (see Table 3.19).

To avoid ambiguity in expressions that reference machine elements, the following uniqueness constraints apply to the names of such elements: a state machine's name should be unique within its feature module, a state's name should be unique within its containing state machine or region, a region's name should be unique within its containing state, a transition's name should be unique within its state machine, and an actions's name should be unique within its transition.

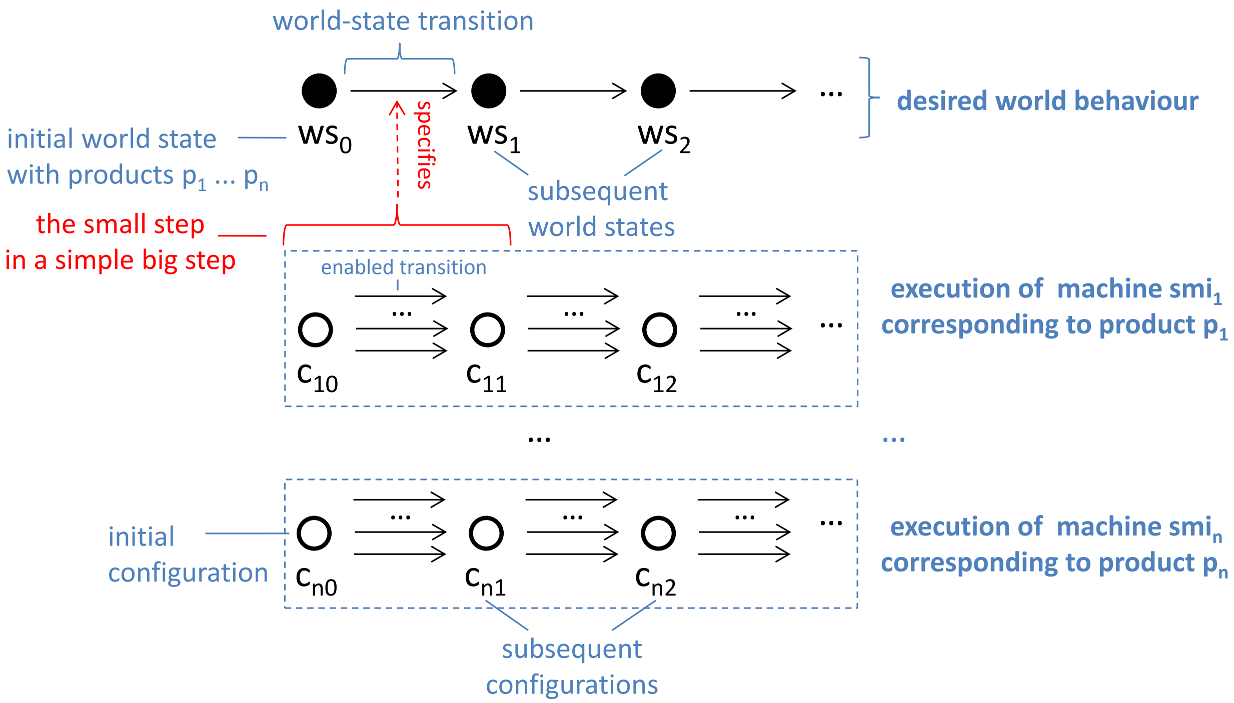


Figure 3.19: A desired world behaviour specified by simple big steps of a set of machines

3.4.2 Simple Big Step

A desired world behaviour starts in some initial (valid) world state *ws0* and is affected by the execution of all products' features' machines. Recall that a world behaviour is a sequence of world states. The world transitions from one world state to the next as specified by a *big step* in the machines' execution. In the simplest case, shown in Figure 3.19, a big step comprises the execution of a single set of concurrently enabled transitions, called a *small step*. The notion of big steps and small steps is adopted from the big-step modelling language (BSML) framework for the semantics of behavioural modelling languages [30]. In more complex cases (described in Section 3.4.3), a world-state transition is specified by a big step comprising a *sequence* of two or more small steps. Big steps are described informally in this chapter, and are formally described in Section 3.6 on semantics.

Definition 3.4.8. *A simple big step in the concurrent execution of a set of machines comprises a single small step.*

[13] The actions of other machines are never referenced in FORML

[14] Recall that there can only be at most one instance of any state machine in any product.

Definition 3.4.9. *A small step comprises the execution of a single set of concurrently enabled transitions of the machines. The enabled actions of the transitions contribute to the next world state, and the destination states of each machine's transitions determine the machine's next configuration.*

We first describe the effects of actions on the world state. Next, we describe the effects of transitions on machine configurations. In Section 3.4.4, we describe the more complex effects of actions and transitions, due to action and transition priorities.

3.4.2.1 Updating the world state

Let $a_1 \dots a_n$ be the set of actions from the set of transitions enabled in the current world state ws_i[15]. The next world state ws_{i+1} is obtained by changing ws_i in a way that satisfies the actions $a_1 \dots a_n$. However, the next world state is usually not unique, as the actions only partially constrain how ws_i should change. The *uncontrolled* parts of ws_i that are not affected by the actions can change arbitrarily within the constraints of the world model. By convention, changes to controlled phenomena (except for the removal of transient output and IO message objects) are specified only by actions in the behaviour model[16]. When there are multiple possible next world states, one is chosen nondeterministically.

For example, consider in Figure 3.20 the execution of machines smi_1 and smi_2 whose state machine is shown in Figure 3.16, starting in world state ws_0 of Figure 3.18. Although the machines do not react to ws_0, uncontrolled phenomena can change (e.g., due to environmental agents): in our example, $User\ u1$ initiates a call to $User\ u2$ by issuing a $StartCall$ message, resulting in world state ws_1. smi_2 does not react to the change from ws_0 to ws_1; however, smi_1's transition $t1$ is enabled:

- The configuration of smi_1 includes $t1$'s source state *idle*.
- $t1$'s trigger StartCall+(o) and guard o.to = myproduct are satisfied by the added $StartCall$ object sc (bound to object variable o): looking at world state ws_1 in Figure 3.18, sc is sent to the *TelSoft* product $ts1$, whose machine is smi_1.

[15] The first big step in an execution is in reaction to the initial world state, which has no previous world state.

[16] A change to controlled phenomena is either the addition or removal of a controlled object or a change in the value of a controlled attribute (e.g., the addition of a *Call* link, where *Call* is a controlled concept in the *TelSoft* world model shown in Figure 3.10).

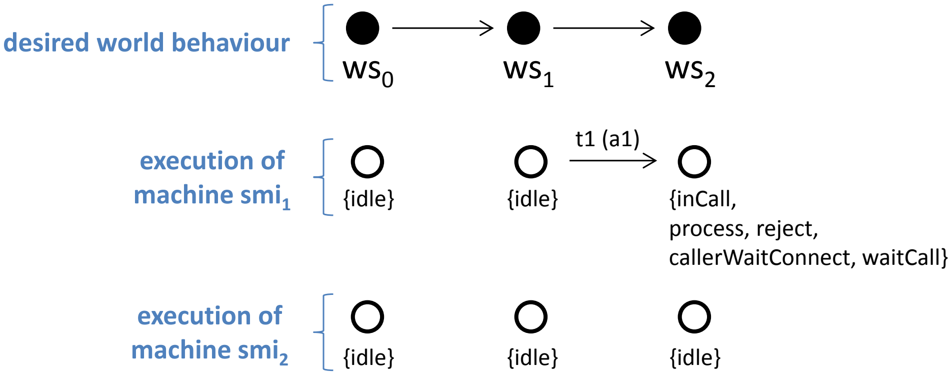

desired world behaviour
ws0
ws1
ws2
execution of machine smi1
{idle}
{idle}
t1 (a1)
{inCall, process, reject, callerWaitConnect, waitCall}
execution of machine smi2
{idle}
{idle}
{idle}

Figure 3.20: Machine executions for the world behaviour shown in Figure 3.18: smi_1 and smi_2 are instances of the BCS state machine *main* (Figure 3.16), corresponding to the *TelSoft* products ts_1 and ts_2 in Figure 3.18, respectively.

- smi_1's reaction is to perform $t1$'s action $a1$, which is enabled by default since the action has no guard. Action $a1$'s WCA

  $$\mathsf{Call} + (\mathsf{caller} = \mathsf{user}, \mathsf{callee} = \mathsf{o.target}, \mathsf{status} = \mathsf{request})$$

  specifies that a new *Call* link is to be added to *ws1*: the *caller* is the *User* object who subscribes to smi_1's product (the macro user expands to myproduct.Subscription.user), the *callee* is the *target* parameter of *sc* (bound to object variable o), and the *status* is request. In our example, the value of world state ws_2 is determined solely by applying action $a1$ to world state ws_1. The world-state transition from ws_1 to ws_2 could have included additional changes (within the world-model constraints) to uncontrolled phenomena, but no such changes occur in this example.

3.4.2.2 Updating a machine's configuration

A machine's configuration is updated in a small step as follows (the description below uses the notions defined over a state-machine's state hierarchy in Definitions 3.4.3 and 3.4.4): for each transition t that is executed in the small step,

1. The source state *src* of t is exited (i.e., removed from the configuration), along with all of *src*'s ancestors and descendants.
2. The destination state *dst* of t is entered (i.e., added to the configuration), along with all of *dst*'s ancestors (except the root) and initial descendants.

3. If the destination state of t is a child state of a region r, and there are regions orthogonal to r that were not in the machine's configuration before executing t, all such regions and their initial descendants are entered[17].

Consider the execution of machine smi_1 in Figure 3.20. The initial configuration is $\{idle\}$. On execution of *t1*, the configuration is updated as follows: (1) The source state *idle* of *t1* is exited. (2) *t1*'s destination state *callerWaitConnect* is entered, along with the state's ancestors (i.e., the region *process* and its superstate *inCall*). (3) The orthogonal regions of the newly entered region *process* and their initial descendants are entered (i.e., the region *reject* and its initial state *waitCall*). Therefore, the configuration of smi_1 after executing *t1* is

$$\{inCall, process, callerWaitConnect, reject, waitCall\}$$

3.4.2.3 Simplifying Assumptions

For simplicity of presentation, the above description of a simple big step makes the following assumptions:

- It is assumed that machines (state-machine instances) in the behaviour model are *deterministic*. In order for a machine to be deterministic, it should not be possible for two or more nonconcurrent transitions (Definition 3.4.7) of the machine to be enabled in the same small step. If multiple nonconcurrent transitions were simultaneously enabled, we would need to select one for execution.

- It is assumed that the enabled actions in a big step are *non-conflicting*: that is, there always exists a valid world-state transition from the current world state that satisfies all of the actions.

- It is assumed that the enabled transitions in a big step have *non-conflicting destination states*: that is, updating the configuration of each machine will always result in a valid configuration of the machine.

We relax the assumption of determinism in Section 4.1, and the assumptions of non-conflicting actions and destination states in Section 4.2.

[17] The process for updating a machine's configuration will fail if it requires computing the initial descendants of a state that has one or more regions that have no initial states. Even though a region may not strictly need an initial state – because none of its ancestor states have incoming transitions (e.g., region *process* in Figure 3.16) – it is good practice to give each region an initial state anyway to accommodate possible extensions of the region's state machine by new features (see Section 3.4.6).

3.4.3 Compound Big Step

There are situations where it is not possible to specify a desired world-state transition in terms of a simple big step. For example, suppose that we want to specify the desired *TelSoft* world behaviour ws_i, ws_{i+1}, ws_{i+2} shown in Figure 3.21 (ignore the world state $ws_{i+1,1}$ for now). This world behaviour represents the case where two users *u1* and *u3* make simultaneous call requests to a third user *u2* (the world-state transition from ws_i to ws_{i+1}), and the desired outcome is for one of these call requests to be connected and for the other to be rejected. The coordination of these two reactions cannot be done concurrently (the calls to be rejected are known only *after* selecting the call to be connected, or vice versa). However, it is often possible to specify the desired world-state transition in terms of a sequence of two or more intermediate world-state transitions that are *unobservable* (i.e., they do not appear in the world behaviour). In the above example, we can first nondeterministically connect one call request(s) (the transition from ws_{i+1} to the intermediate world state $ws_{i+1,1}$) and subsequently reject the remaining call request (the transition from $ws_{i+1,1}$ to ws_{i+2}). To accommodate such cases, a desired world-state transition can be specified as a *compound big step*:

Definition 3.4.10. *A compound big step in the execution of a set of machines comprises a sequence of two or more small steps. Each small step in a compound big step specifies an intermediate world-state transition that (except for the last small step) results in an intermediate world state that is the source state of the next small step. The end of an intermediate small step in a compound big step is marked with one or more unstable states (see Definition 3.4.5).*

Figure 3.22 shows an example of a compound big step.

3.4.3.1 Big step termination

A big step continues (with additional small steps) so long as any of the executing machines are in an unstable configuration (i.e., the configuration contains one or more unstable states); and a big step terminates when all of the machines arrive at a stable configuration. For example, in Figure 3.22, the big step terminates after $m+1$ small steps, because after each of the first m small steps, at least one machine (smi_n) is in an unstable configuration; whereas, after the $m+1^{th}$ small step, all of the machines are in stable configurations. As soon as a machine reaches a stable configuration, it does not execute any more transitions in the current big step. For example, in Figure 3.22, the machine smi_1 does not execute any

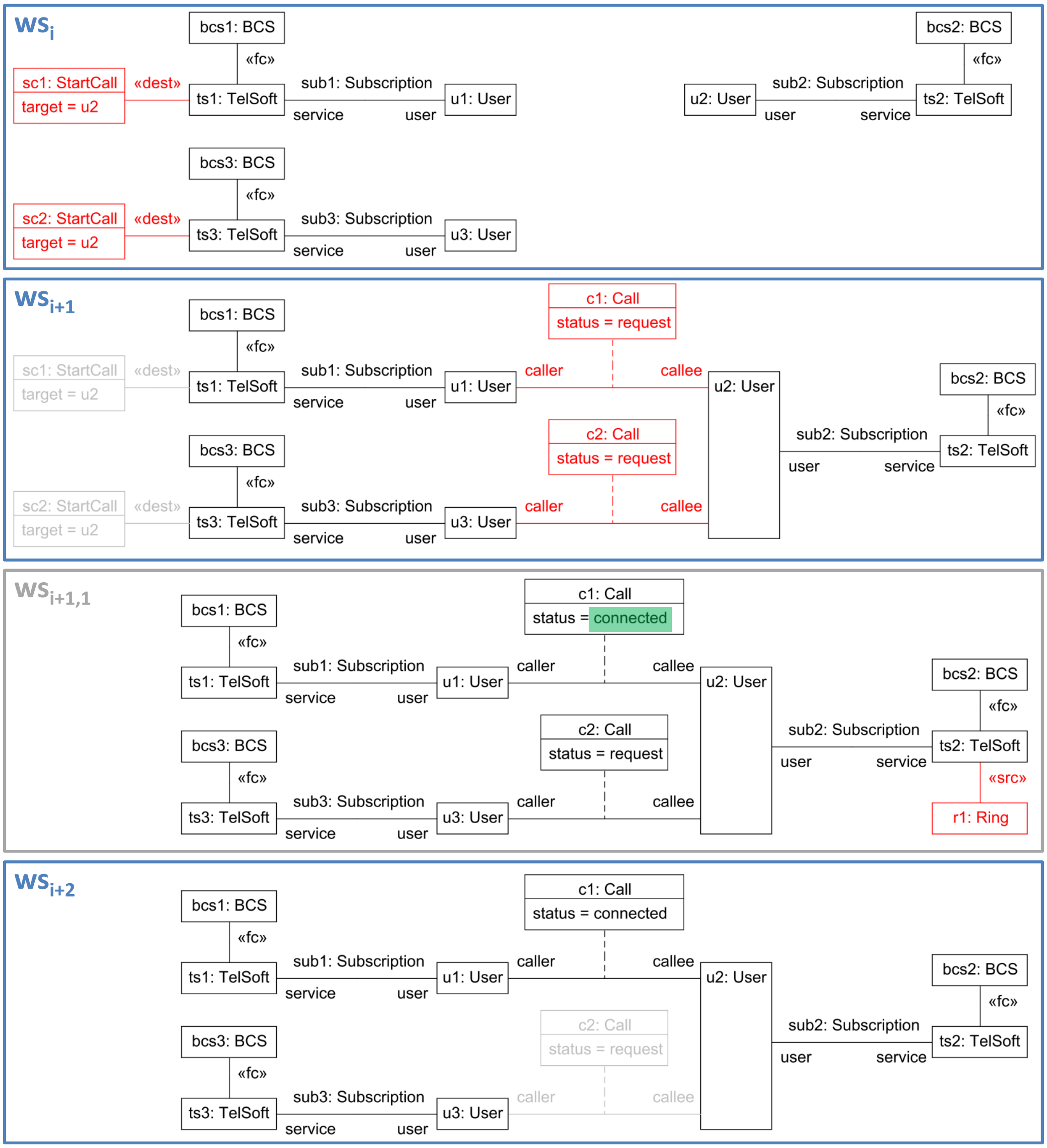


Figure 3.21: A world behaviour of the *TelSoft* world model shown in Figure 3.10: Added elements are coloured red, removed elements are coloured gray, and change attribute values are shaded green.

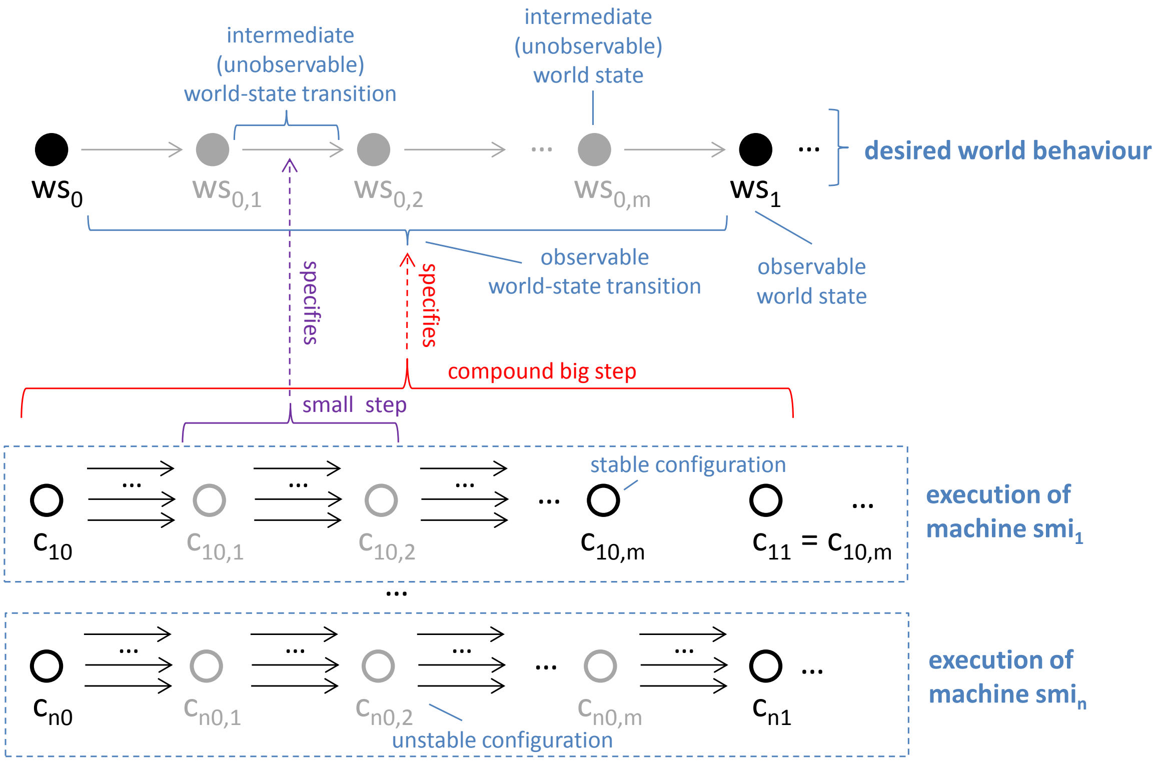


Figure 3.22: A desired world behaviour specified by compound big steps of a set of machines: unstable configurations and unobservable world states are shown in gray.

transitions in the $m + 1^{th}$ small step of the big step, because it is in a stable configuration $c_{10,m}$.

To ensure that a big step always terminates, each state machine *sm* in the behaviour model should satisfy the following well-formedness conditions:

- The *configuration graph* of *sm* should not contain a cycle of unstable configurations. The configuration graph of *sm* is a directed graph, whose nodes are the configurations of *sm*. A directed edge from a configuration c_1 to a configuration c_2 denotes the existence of one or more transitions in *sm* whose execution will change the configuration of an *sm* instance from c_1 to c_2.
- Every unstable state should have an *else transition* that leads to another state. An *else transition* has a label of the form

$$id : else \ / \ al_1, \ \ldots, \ al_n$$

where id is the transition name, and $a1_1 \ldots a1_n$ are action labels as described in Section 3.4.1.3. An else transition t is enabled if the machine's configuration includes t's unstable source state and no other *non-else* transitions that emanate from t's source state are enabled.

The above conditions ensure that a machine will never get stuck in an unstable configuration. Note that the unstable state *issueReject* in Figure 3.16 does not have an outgoing else transition. This is because *issueReject*'s only outgoing transition $t6$ has no trigger or guard, and thus is always enabled in the unstable configuration $\{issueReject\}$; hence, a *main* machine can never get stuck in this unstable configuration.

3.4.3.2 @curri expressions

In a compound big step, expressions in each small step (e.g., trigger expressions, transition guards, actions) are interpreted with respect to a state $\mathbb{s}$ that includes not just the most recent world-state transition $\boldsymbol{(ws_{i-1}, ws_i)}$ $(i \geq 1)$, but also the current intermediate world state $ws_{i,j}$ $(j \geq 1)$. We use the construct @curri for cases when a transition originating from an unstable state has an expression that should be evaluated with respect to the current intermediate world state. As with @pre expressions (Section 3.3.7), the syntax is to affix @curri to an atomic set expression of the form Cs, to a navigation expression, to a parenthesized expression, or to a macro. For example, the expression

Calls@curri[c | c.status = connected]

evaluated in the intermediate world state $ws_{i+1,1}$ in Figure 3.21 returns the $Call$ link (set) $\{c1\}$. However, the expression

Calls[c | c.status = connected]

returns the empty set when evaluated in $ws_{i+1,1}$ because the most recent observable world state ws_{i+1} contains no connected calls (only call requests).

3.4.3.3 Updating the world state

There are two cases for computing the next world state of a small step in a compound big step: the first case applies to small steps that lead to intermediate world states; and the second case applies to the last small step in the big step, leading to the next observable world state.

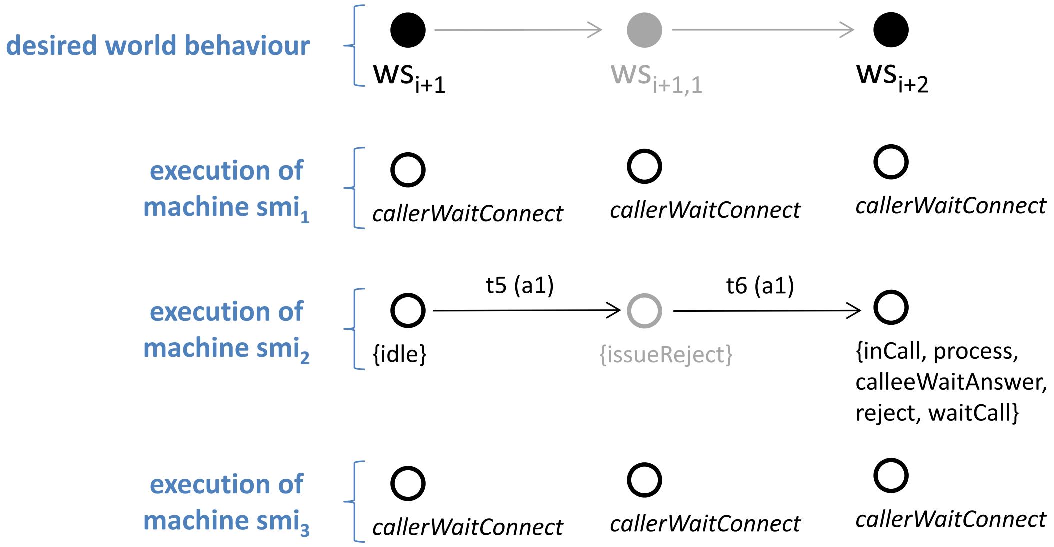


Figure 3.23: Machine executions for the world behaviour shown in Figure 3.21: smi_1, smi_2, and smi_3 are instances of the BCS state machine *main* (Figure 3.16), corresponding to the *TelSoft* products ts_1, ts_2, and ts_3 in Figure 3.18, respectively. *callerWaitConnect* is an abbreviation for the configuration $\{inCall, process, callerWaitConnect, reject, waitCall\}$.

Case 1: An intermediate small step in a compound big step Let ws_c and ws_p be the current and previous observable world states, and let ws_{ci} be the current intermediate world state that is the source of an intermediate small step in a compound big step. At the start of a big step, the current intermediate world state ws_{ci} is equal to the current observable world state ws_c. Let $a_1 \ldots a_n$ be the set of enabled actions that are performed in the small step. The next intermediate world state ws_{ni} is obtained by changing ws_{ci} *exactly* as prescribed by the actions $a_1 \ldots a_n$, and disallowing any further changes. Note that the world state ws_{ni} obtained by this rule is not necessarily valid; however, invalid intermediate world states are tolerated since they are unobservable.

For example, consider the execution in Figure 3.23 of the machines smi_1, smi_2, and smi_3 (instances of the BCS state machine *main* in Figure 3.16) starting in world state ws_{i+1} of Figure 3.21. smi_1 and smi_3 do not react to the change from ws_i to ws_{i+1}; however, smi_2's transition $t5$ is enabled:

- The configuration of smi_2 includes $t5$'s source state *idle*.
- $t5$'s trigger Call+(o) and guard o.callee = user are satisfied by two added *Call* links $c1$

and $c2$ (when bound to object variable o): looking at world state ws_{i+1} in Figure 3.21, the *callee* of both $c1$ and $c2$ is the $User$ object $u2$ (i.e., the $User$ object that the macro $\mathsf{user} = \mathsf{myproduct.Subscription.user}$ evaluates to[18]).

- smi_2's reaction is to perform $t5$'s action $a1$, which is enabled by default since it has no guard. Action $a1$'s WCA

  $$\mathsf{o.status} = \mathsf{connected}$$

  specifies that a $Call$ link ($c1$ or $c2$, chosen nondeterministically and bound to object variable o) is to be connected. In our example, $c1$ is connected.

Transition $t5$ updates smi_2's configuration to the unstable configuration $\{issueReject\}$, which implies that the big step has not yet terminated. Hence, the change prescribed by action $a1$ of $t5$ is the only change applied to the current intermediate world state (equal to ws_{i+1}), resulting in the next intermediate world state $ws_{i+1,1}$.

Case 2: The last small step in a compound big step Let ws_c and ws_p be the current and previous observable world states, and let ws_{ci} be the current intermediate world state. Let $a_1 \dots a_n$ be the set of enabled actions that are performed in the small step. At first, it may appear that the next world state ws_n can be obtained by treating the last small step as a simple big step (see Section 3.4.2) applied to ws_{ci}: that is, by making changes to ws_{ci} that satisfy the actions $a_1 \dots a_n$, including arbitrary changes to the uncontrolled parts of ws_{ci} that are not affected by the actions. The problem with this approach is that the arbitrary changes can counteract changes made by actions in previous small steps of the compound big step. For example, an object added in previous small steps may be removed as an arbitrary change in the last small step. To avoid this problem, the next observable world state ws_{no} is computed using a two-step process:

1. The net effects of all of the small steps in the big step are accumulated in a temporary (and possibly invalid) world state ws_n, obtained by changing the current intermediate world state ws_{ci} exactly as prescribed by the actions $a_1 \dots a_n$.
2. An *implied* set of actions $a'_1 \dots a'_k$ is computed for the big step.

   Definition 3.4.11. *An implied set of actions for a big step is a minimal set of actions that, if applied to the current observable world state ws_c, would result in a temporary world state ws_n that accumulates the net effects of all of the small steps in the big step.*

[18] The variable *myproduct* of smi_2 evaluates to the *TelSoft* object $ts2$, which is related (in the world-behaviour model in Figure 3.21) to the $User$ object $u2$ via a *Subscription* link.

Finally, the next observable world state ws_{no} is obtained as if a simple big step with actions $a'_1 \ldots a'_k$ is applied to ws_c: that is, the changes applied to ws_c satisfy the actions $a'_1 \ldots a'_k$, and include arbitrary changes to the uncontrolled parts of ws_c that are not affected by the actions.

For example, after the first small step in Figure 3.23 (as part of the execution of instances of the BCS state machine *main* in Figure 3.16), smi_1 and smi_3 once again have no reactions; however, smi_2's transition $t6$ is enabled:

- The configuration of smi_2 includes $t6$'s unstable source state *waitReject*. $t6$ has no other enabling conditions, since it has no trigger or guard expressions.
- smi_2's reaction is to perform $t6$'s action $a1$, which is enabled by default since the action has no guard. Action $a1$'s WCA

  $$-\mathsf{Call}(\mathsf{callRequests@curri})$$

  specifies that all remaining *Call* links whose *status* attribute has the value *request* be removed.

Transition $t6$ updates smi_2's configuration to the stable configuration

$$\{inCall, process, calleeWaitAnswer, reject, waitCall\}$$

which together with the fact that smi_1 and smi_3 are also in stable configurations, implies that this is the last small step in the compound big step. Hence, the two-step process is used to compute the next observable world state: (1) The change prescribed by *a1* is applied exactly to $ws_{i+1,1}$ resulting in a temporary world state ws_n. (2) An implied set of actions is computed for the big step that, if applied to ws_{i+1}, would result in ws_n (see Definition 3.4.11):

$$\{\ \mathsf{c1.request} = \mathsf{connected},\ \ -\mathsf{Call}(\mathsf{c2})\ \}$$

The world state ws_{i+2} is determined by applying the implied set of actions plus arbitrary changes to uncontrolled phenomena in ws_{i+1}. In our example in Figure 3.21, no additional arbitrary changes are applied.

The simplifying assumptions about determinism and the absence of conflicts, stated in Section 3.4.2 for simple big steps, apply also to the above description of small steps in a compound big step.

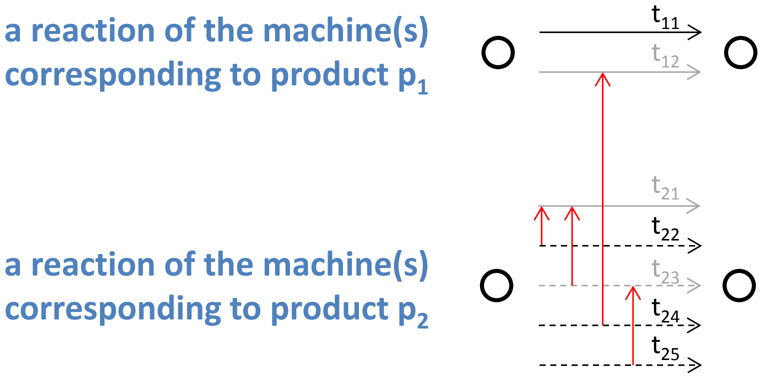

a reaction of the machine(s) corresponding to product p_1
t_{11}
t_{12}
t_{21}
t_{22}
a reaction of the machine(s) corresponding to product p_2
t_{23}
t_{24}
t_{25}

Figure 3.24: A small step involving preemptive transitions: a solid arrow denotes a non-preemptive transition, a dotted arrow denotes a preemptive transition, a red arrow goes from a preemptive transition to its target transition, and a gray arrow denotes a preempted transition.

3.4.4 Preemptive Transitions and Actions

Priorities are often established among the requirements of features, either to indicate that a new requirement overrides an existing requirement or to resolve a conflict between two requirements. For example, call waiting (CW) overrides BCS's busy-treatment requirement, by connecting a second incoming call rather than rejecting it. As an example of using priorities to resolve a requirements conflict, consider the following: BCS requires that when a callee answers a call, the call should have a voice connection; and when a caller ends a call, the call should be removed from the world. However, these two requirements conflict when the caller and callee of the same connected call simultaneously answer and end the call. One possible resolution is to favour the caller's request over the callee's request.

In FORML, requirements priorities are modelled as priorities between pairs of transitions and actions, using *preemptive transitions* and *actions*, respectively. This section focuses on modelling requirements priorities within the same feature (in the same or different products). Section 3.4.6 addresses modelling requirements priorities between different features.

3.4.4.1 Preemptive Transitions

Definition 3.4.12. *A preemptive transition has a higher priority than one or more – implicitly or explicitly specified – other transitions called its target transitions.*

As shown in Figure 3.24, when a preemptive transition (e.g., t_{22}) is enabled together with its target transition (t_{21}) in the same small step, the target transition is *preempted*: that is, it does not execute in the small step. It is possible for one transition to be preempted by multiple preemptive transitions in a small step (e.g., t_{21} is preempted by both t_{22} and t_{23}). It is also possible for a preemptive transition to itself be preempted in a small step; however, its target transitions are still preempted (e.g., even though t_{23} is preempted by t_{25}, it still preempts t_{21}). Finally, the target transition of a preemptive transition may belong to the same product or to different products (e.g., t_{24} in product p_2 preempts transition t_{12} in product p_1). To ensure that big steps always terminate (see Section 3.4.3.1), the target of a preemptive transition cannot be an else transition. There are three basic types of preemptive transitions that differ in how they specify their target transitions or in their enabling conditions.

Super transitions: A *super transition* is a transition whose source state is a superstate. A super transition is an implicit preemptive transition that has priority over all of the transitions whose source states are descendants of t's source state in the state hierarchy. For example, in BCS's feature module (see Figure 3.16), $t10$ is a super transition whose (implicit) target transitions are $t3$, $t4$, $t7$, and $t8$; the priority of $t10$ over $t7$, for example, specifies that if a callee simultaneously answers and ends a connected call, only the end-call request should be processed. Note that the target transitions of a super transition belong to the same machine as the super transition; hence, super transitions specify priorities only among the requirements of a single product.

Priority transitions: A *priority transition* has a label of the form

$$id > list(id_{target}) : te \ [gc] \ / \ al_1, \ \ldots, \ al_n$$

where id is the name of the priority transition, and $list(id_{target})$ is a comma-separated list of references to the target transitions. Using the syntax described in Section 3.4.1.4, id_{target} can refer to a target transition in the same machine, in a different machine of the same product, or in a different machine of a different product. For example, in BCS's feature module (see Figure 3.16), the priority transition $t2$ has a higher priority than transition $t8$ of the same machine (this priority resolves a conflict between the destination states of the concurrent transitions $t2$ and $t8$ when they are both enabled). As another example, given two *TelSoft* products with BCS, one for the caller and one for the callee of the same call, transition $t10$ of the caller's machine (which models the caller ending a call) has priority over transition $t7$ of the callee's machine (which models the callee answering the call). The label on priority transition $t10$ starts with

$$t10 > calleeTelSoft(main).t7$$

where the macro calleeTelSoft evaluates to the callee's *TelSoft* product. The rest of the label of a priority transition, as well as its enabling conditions, are the same as those of a non-preemptive transition.

Override transitions: An *override transition* has a label of the form

$$id : override(id_{target})\ [gc]\ /\ al_1,\ \ldots,\ al_n$$

where id_{target} refers to a unique target transition using the syntax described in Section 3.4.1.4. The override transition id inherits the enabling conditions of its target transition and augments them with an optional guard condition gc. Thus, the id transition of a machine m is enabled in a small step if m's configuration includes id's source state, id's target transition is enabled, and gc is satisfied. Examples of override transitions are given in Section 3.4.6, in the context of modelling the requirements of new features.

Override transitions do not add expressiveness to FORML: it is possible to represent an override transition of the above form as a priority transition that replicates the enabling conditions of the target transition id_{target}:

$$id > id_{target} : te_{target}\ [src_{target}\ \wedge\ gc_{target}\ \wedge\ gc]\ /\ al_1,\ \ldots,\ al_n$$

where te_{target} and gc_{target} are the trigger expression and guard condition of id_{target}, respectively; and src_{target} is an *inState* predicate that checks if id_{target}'s source state is in the corresponding machine's configuration. However, there are two problems with this representation: (1) the *intent* of overriding the target transition is left implicit; and (2) replicating the target transition's enabling conditions hinders modifiability: if the target transition's enabling conditions change, the label of the priority transition would also have to be changed. In contrast, override transitions explicate the intent of overriding the target transition and inherit the target transition's enabling conditions.

Note that the above types of preemptive transitions are not mutually exclusive. For example, a transition whose source state is a superstate and whose label is

$$id > list(id1_{target}) : override(id2_{target})\ [gc]\ /\ al_1,\ \ldots,\ al_n$$

is a super transition, a priority transition, and an override transition, all at the same time.

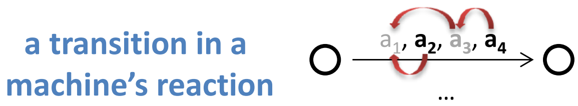


Figure 3.25: A transition involving preemptive actions: non-preemptive actions are not in bold (e.g., a_1), preemptive actions are in bold (a_2, a_3, a_4), a red arrow goes from a preemptive action to its target action, and a preempted action is in gray.

3.4.4.2 Preemptive Actions

Definition 3.4.13. *Analogously, a preemptive action specifies a priority over one or more other actions, called its target actions.*

Preemptive actions specify priorities only among actions of the same transition – hence, they affect the requirements of a single product. As shown in Figure 3.25, when a preemptive action (e.g., a_2) is enabled together with its target action (a_1), the target action is preempted: that is, it does not execute. It is possible for one action to be preempted by multiple preemptive actions (e.g., a_1 is preempted by both a_2 and a_3). It is also possible for a preemptive action to itself be preempted, but its target actions are still preempted (e.g., even though a_3 is preempted by a_4, it still preempts a_1). There are two basic types of preemptive actions that differ in their enabling conditions.

Priority actions: A *priority action* has a label of the form

$$id > list(id_{target}) : [gc]\ a$$

where id is the name of the priority action, and $list(id_{target})$ is a comma-separated list of the names of target actions. The rest of the label of a priority action, as well as its enabling conditions, is the same as that of a non-preemptive action.

Override actions An *override action* has a label of the form

$$id : override(id_{target})\ [gc]\ a$$

where id_{target} refers to a unique target action, and the rest of the label is the same as that of a non-preemptive action. The override action id inherits the enabling conditions of id_{target} and possibly augments them with gc. Thus, the override action id is enabled if action id_{target} is enabled (i.e., id and id_{target}'s (common) transition is executing and id_{target}'s enabling conditions hold) and gc is satisfied.

Override actions do not add expressiveness to FORML and can be equivalently represented by priority actions (in a manner analogous to the representation of override transitions by priority transitions). However, like override transitions, override actions have the benefits of explicating the intent of overriding other actions and avoiding the replication of the target actions' enabling conditions.

Examples of preemptive actions are given in Section 3.4.6, in the context of modelling the requirements of new features. Finally, as with preemptive transitions, the types of preemptive actions are not mutually exclusive. For example, an action whose label is

$$id > list(id1_{target}) : override(id2_{target})\ [gc]\ a$$

is a priority action and an override action at the same time.

3.4.5 Comparison with UML State Machines

FORML state machines adopt from UML state machines the foundational constructs of basic and hierarchical states, concurrent regions, and transitions. Syntactically, FORML extends and adapts UML state machines as follows:

- FORML transitions and actions are named.
- FORML actions can have guard conditions.
- FORML introduces the language of world-change events (WCEs) and WCAs for transition triggers and actions, respectively.
- FORML introduces priority and override transitions and actions.
- FORML uses named unstable states in place of UML pseudo states.

The semantics of FORML state machines is similar to that of UML state machines [84] in that both simple and compound big steps are possible, and the termination criteria for big steps is specified syntactically (via stable states in FORML and non-psuedo states in UML). However, there are several major differences:

- In UML, an arbitrary classifier can have a state machine that is instantiated once per instance (object) of the classifier. In FORML, state machines are associated with a particular classifier, namely, the SPL concept; hence, state machines are instantiated once per product.

- UML state-machine instances operate on the local data of their associated object and synchronize via mechanisms such as signals and operation calls. FORML state-machine instances all operate on global data (the world state) and synchronize via changes to the global data.

- UML semantics includes the notion of an event pool that stores unprocessed signals and operation calls; together with the local data of objects, the event pool is used to determine transition enabledness. FORML semantics does not include an event pool, because transition and action enabledness is determined solely by global data.

- In UML, an execution step processes a single event from the event pool, with the choice of event being a variation point in UML semantics. In FORML, all events (changes to global data) generated in one exection step are processed in the next execution step.

3.4.6 State-Machine Fragments

An SPL is primarily evolved by adding new features. When a new feature is independent of existing features, its requirements can be modelled as one or more new state machines that execute in parallel with the existing features' machines. However, the purpose of a new feature can (in part) be to *enhance* an existing feature's requirements. There are three types of enhancements:

- A new feature can add new requirements in the context of an existing feature's requirements. For example, caller delivery (CD) adds requirements for delivering a caller's identity to the callee, in the context of the call-processing requirements of basic call service (BCS).

- A new feature can modify the requirements of an existing feature. In other words, a new feature can have *intended interactions* with an existing feature. For example, call waiting (CW) modifies the usual busy-treatment of BCS by connecting a second incoming call rather than rejecting it. As another example, a caller's caller delivery blocking (CDB) feature prevents the callee's CD feature from delivering the caller's identity to the callee.

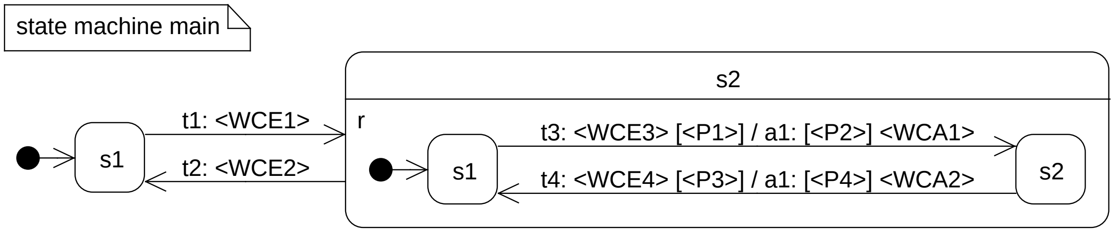

state machine main
s1
t1: <WCE1>
t2: <WCE2>
s2
r
s1
t3: <WCE3> [<P1>] / a1: [<P2>] <WCA1>
t4: <WCE4> [<P3>] / a1: [<P4>] <WCA2>
s2

Figure 3.26: *B* feature module: <*WCE1-4*>, <*WCA1-2*>, and <*P1-4*> are trigger expressions, WCAs, and predicates respectively.

- A new feature F_{new} can add a *retrospective intended interaction* to an existing feature F: specifically, F_{new} can specify that a requirement of F takes priority over a requirement of F_{new} or another existing feature.[19] For example, CW requires BCS to override CW under the following conditions: when CW is active and the voice-connected call is ended by one of the call's parties, CW establishes a voice-connection between the subscriber and the waiting party. However, if the remote user of the waiting call uses BCS to end the call at the same time, BCS overrides CW and the call is ended.

Note that it is possible to have enhancements of enhancements: in the above examples, CDB enhances CD, which itself enhances BCS.

It is natural to model enhancements in terms of *differences* from the enhanced features' requirements. In this approach, the model of the enhanced feature is reused as the context for expressing the new feature. FORML supports modelling a new feature's enhancements of existing features' requirements in terms of *state-machine fragments* (fragments for short) that extend existing feature modules.

Specifying a fragment involves referencing elements in existing feature modules. A fragment necessarily references the state-machine element that it extends, but can also reference existing macros from within expressions; and reference existing state machines, states, transitions, and actions from within *inState* predicates, and preemptive transitions and actions. An element n in the feature module of an existing feature F can be referenced by its *qualified name* `F{n}`; n is the element's *global name* within its feature module. The global name of a state or region is described in Section 3.4.1.1. The global name of a

[19] We say that F has a (retrospective) intended interaction with F_{new} (or another existing feature) because it is F that is modifying the requirements of F_{new} (by taking precedence over F_{new}'s requirements) and not the other way around. The term *retrospective* reflects the fact that the intended interaction is added to the requirements of a feature introduced before F_{new}.

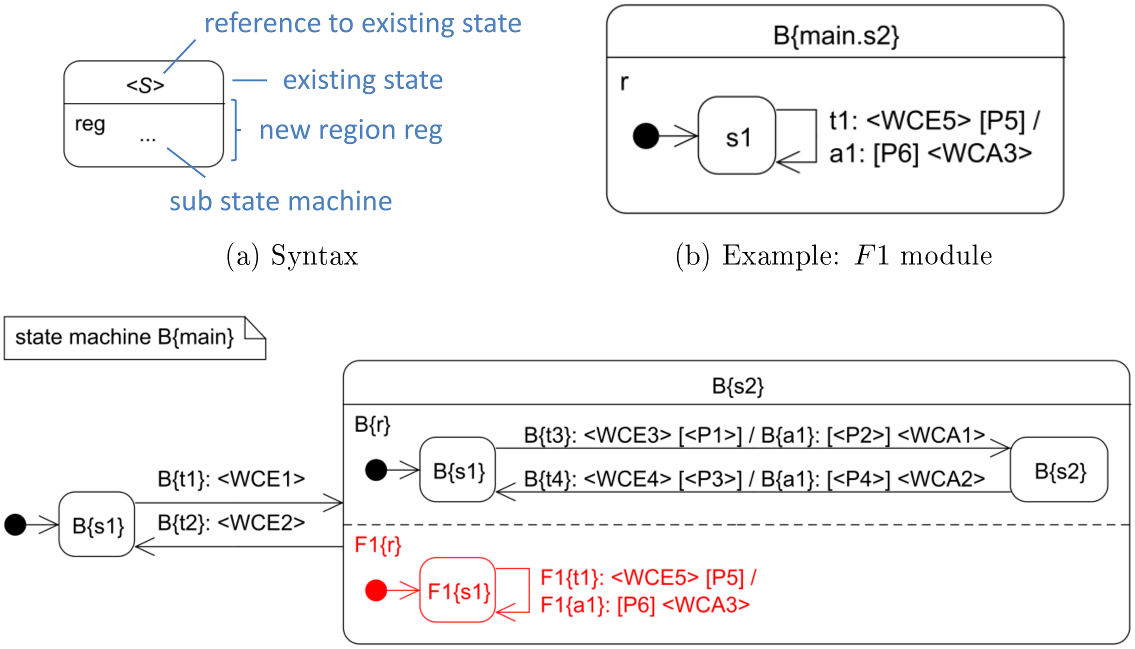
transition t of a state machine sm is $sm.t$, and the global name of an action a of t is $sm.t.a$. The global names of fragments are described in Section 3.4.6.4.

The following subsections give both pedagogical and real examples for the different types of fragments. The pedagogical examples are based on the abstract feature module shown in Figure 3.26, which belongs to a fictitious feature B and comprises a single state machine *main*. As real examples, the models for CW, CD, and CDB are presented, and references are given to additional *TelSoft* and *AutoSoft* models in Appendix A.

3.4.6.1 Modelling the Addition of Requirements to Existing Features

Recall that, fundamentally, an existing feature's requirements are specified (in its feature module) as transition actions that constrain changes to the world state, and as enabling conditions of transitions and actions that constrain when the actions apply. A fragment specifies new behaviours by adding new actions to existing feature modules. There are three types of fragments for adding requirements to existing features.

New regions A *new region* extends a stable state in an existing feature module. Figure 3.27a shows the syntax for specifying a new region. The new region is shown (using the usual notation for regions) in the context of the superstate that is to be extended. The superstate's name is a reference to the state being extended. For example, in Figure 3.27b, a new feature $F1$ extends state $s2$ of state machine *main* in B's feature module (shown in Figure 3.26) with a new region r. The state to be extended is referenced by the expression B{main.s2}. The composition of the $F1$ and B feature modules, shown in Figure 3.27c, illustrates the effect of the extension. Note how the name of each element in the composed model is qualified with the name of the feature that introduced the element; for example, state $s1$ in B's ($F1$'s) feature module is named $B\{s1\}$ ($F1\{s1\}$) in the composed model. There is more to composing feature modules than what appears in this section's examples. Feature-module composition is discussed in detail in Section 3.5.

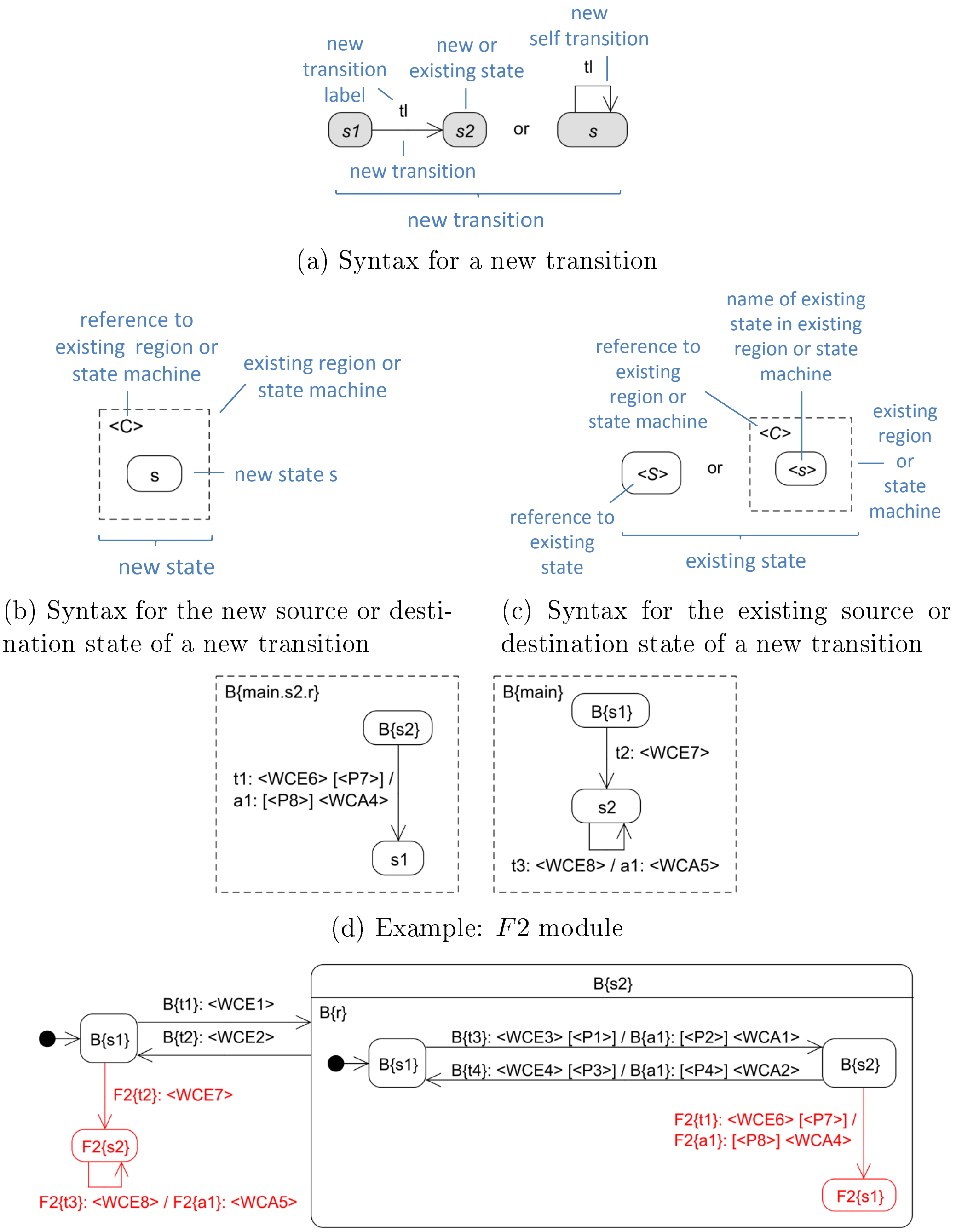


(a) Syntax

(b) Example: $F1$ module

(c) $F1$ composed with B (the new region is coloured in red)

Figure 3.27: State-machine fragment to specify new regions



(a) Syntax for a new transition

(b) Syntax for the new source or destination state of a new transition

(c) Syntax for the existing source or destination state of a new transition

(d) Example: $F2$ module

(e) $F2$ composed with B (the new states and transitions are coloured in red)

Figure 3.28: State-machine fragment to specify new transitions

New non-preemptive transitions A *new transition* extends a state machine in an existing feature module. The source and destination states of a new transition can each be either an existing state or a *new state*. Figure 3.28a shows the syntax for specifying a new transition. The new transition is shown (as usual) as a labelled arrow between its source and destination states.

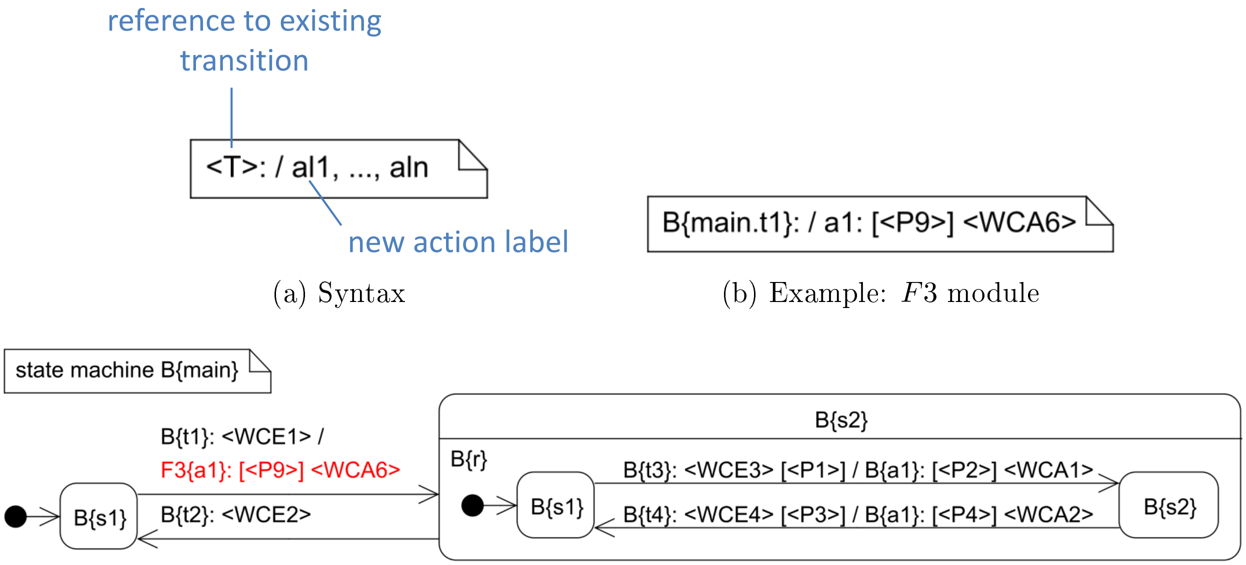
Figure 3.28b shows the syntax for specifying a new (source or destination) state in the context of the region or state machine that is being extended; the latter is represented by a dotted box labelled (at its top-left corner) with a reference to the extended region or state machine[20]. Figure 3.28c shows the syntax for specifying an existing (source or destination) state. In the simplest case, an existing state can be shown as a basic state (regardless as to whether the state is actually a basic state) whose name is a reference to the existing state. When specifying multiple (new or existing) states within the same region or state machine, it is convenient to show the states in the context of a single dotted box representing the shared container.

For example, in Figure 3.28d, a new feature $F2$ extends state machine $main$ in B's feature module (shown in Figure 3.26) with three new transitions: (1) transition $t1$ from state $s2$ in region B{main.s2.r} to a new state $s1$ in the same region, (2) transition $t2$ from the top-level state $s1$ in $main$ to a new top-level state $s2$ in $main$, and (3) a self transition $t3$ from the new top-level state $s2$ in $main$. The composition of the feature modules for $F2$ and B is shown in Figure 3.28e.

New non-preemptive actions A *new action* extends a transition in an existing feature module. Figure 3.29a shows the syntax for specifying a set of new actions that extend the same transition: a UML note that names the extended transition is followed by a comma-separated list of action labels.

For example, in Figure 3.29b, a new feature $F3$ extends transition $t1$ of state machine $main$ in B's feature module (shown in Figure 3.26) with a new action $a1$. $t1$ is referenced by the expression B{main.t1}. The composition of the feature modules for $F3$ and B is shown in Figure 3.29c.

[20] Even though the new state is shown as a basic stable state in Figure 3.28b, the new state could also be a basic unstable state or a superstate.


reference to existing transition
<T>: / al1, ..., aln
new action label
B{main.t1}: / a1: [<P9>] <WCA6>
state machine B{main}
B{s1}
B{t1}: <WCE1> /
F3{a1}: [<P9>] <WCA6>
B{t2}: <WCE2>
B{s2}
B{r}
B{s1}
B{t3}: <WCE3> [<P1>] / B{a1}: [<P2>] <WCA1>
B{t4}: <WCE4> [<P3>] / B{a1}: [<P4>] <WCA2>
B{s2}

(a) Syntax

(b) Example: $F3$ module

(c) $F3$ composed with B (the new action is coloured in red)

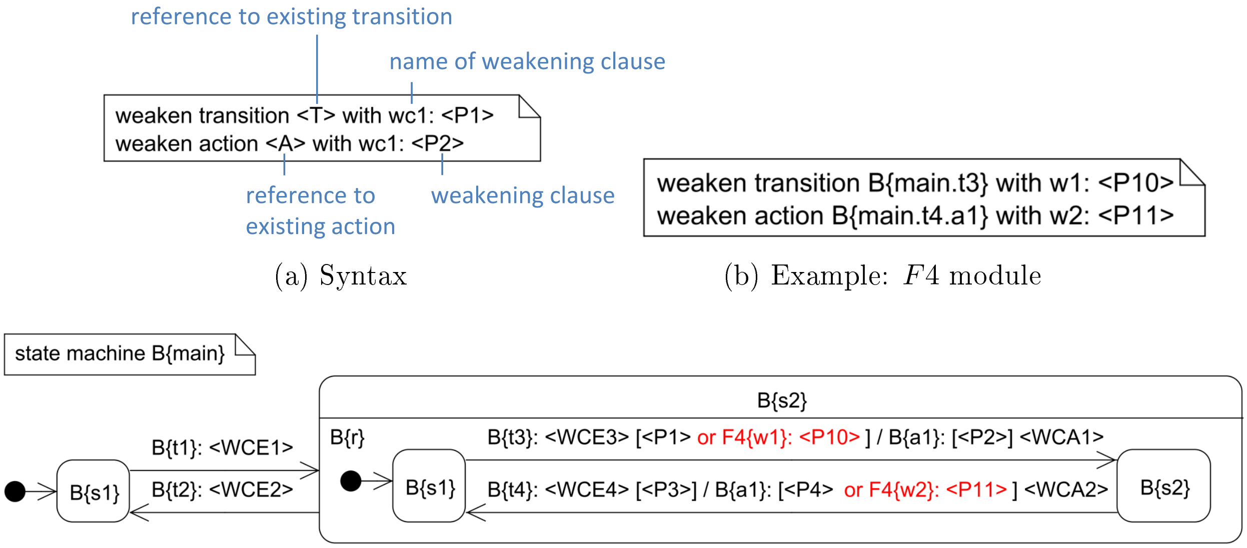
Figure 3.29: State-machine fragment to specify new actions

3.4.6.2 Modelling Intended Interactions

The following fragments can be used to model a new feature's modifications to the requirements of existing features, called the new feature's **intended interactions** with existing features. A fragment specifies an intended interaction by adding new enabling conditions or removing conditions on existing transitions or actions; or by overriding existing transitions (actions) with new transitions (actions).

Weakening and strengthening clauses A *weakening clause* is a predicate that extends, through *disjunction*, the guard condition of a transition or action in an existing feature module. Weakening a guard *adds* a new condition under which the guarded transition or action is enabled for execution. Figure 3.30a shows the syntax for specifying a weakening clause and how the clause is associated with an existing feature's transition or action. A weakening clause is named to allow the clause to be referenced by future feature modules for the purpose of weakening or strengthening the clause itself.

For example, in Figure 3.30b, a new feature $F4$ adds the weakening clause $w1$ to the guard of transition $B\{t3\}$; and adds the weakening clause $w2$ to the guard of action $a1$ in transition $B\{t4\}$. The composition of the feature modules for $F4$ and B is shown


reference to existing transition
name of weakening clause
weaken transition <T> with wc1: <P1>
weaken action <A> with wc1: <P2>
reference to existing action
weakening clause
weaken transition B{main.t3} with w1: <P10>
weaken action B{main.t4.a1} with w2: <P11>
state machine B{main}
B{s2}
B{r}
B{t1}: <WCE1>
B{t2}: <WCE2>
B{s1}
B{s1}
B{t3}: <WCE3> [<P1> or F4{w1}: <P10>] / B{a1}: [<P2>] <WCA1>
B{t4}: <WCE4> [<P3>] / B{a1}: [<P4> or F4{w2}: <P11>] <WCA2>
B{s2}

(a) Syntax

(b) Example: $F4$ module

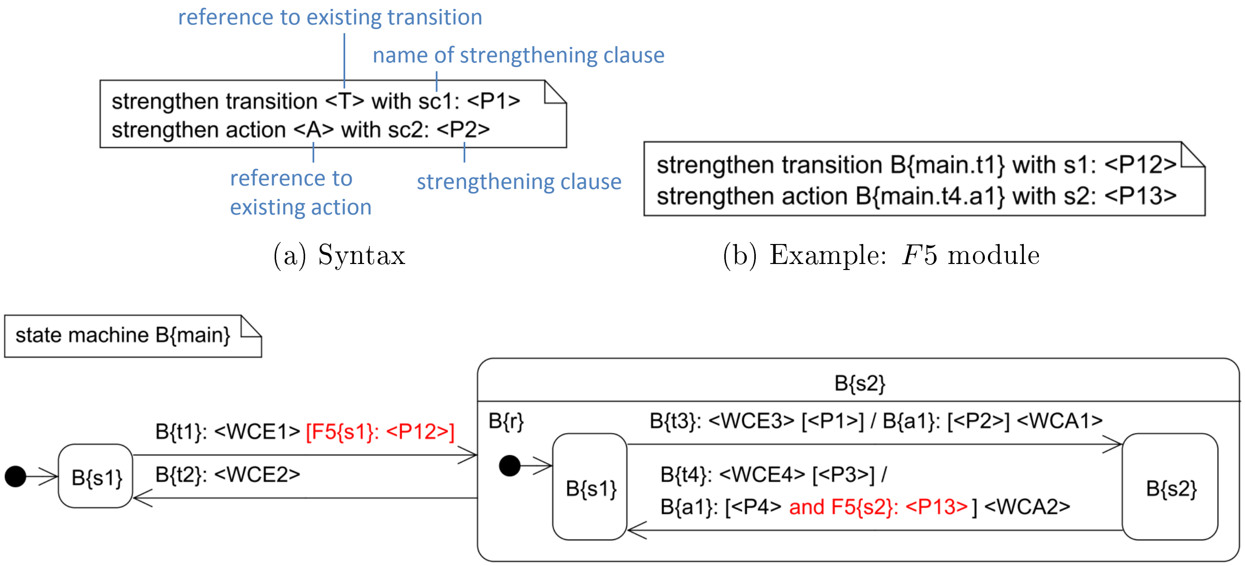
(c) $F4$ composed with B (the weakening clause is coloured in red)

Figure 3.30: State-machine fragment to specify weakening clauses

in Figure 3.30c. The effect of $w1$ (respectively, $w2$) is to extend the guard condition of transition $B\{t3\}$ (respectively, action $a1$ in $B\{t4\}$) with a disjunctive clause, which extend the set of circumstances under which transition $B\{t3\}$ and action $a1$ in transition $B\{t4\}$ execute.

Analogously, a *strengthening clause* is a predicate that extends, through *conjunction*, the guard condition of a transition or action in an existing feature module. Unlike a weakening clause, a strengthening clause can be applied to a transition or action that does not have a guard: in this case, the strengthening clause becomes the guard of the transition or action. Strengthening or adding a guard *removes* a condition under which the extended transition or action is enabled for execution. Figure 3.31a shows the syntax for specifying a strengthening clause, which is analogous to that of a weakening clause.

For example, in Figure 3.31b, a new feature $F5$ adds the strengthening clause $s1$ to transition $B\{t1\}$; and adds the strengthening clause $s2$ to the guard of action $a1$ in transition $B\{t4\}$. The composition of the feature modules for $F5$ and B is shown in Figure 3.31c. The effect of $s1$ is to add a guard condition to transition $B\{t1\}$, and the effect of $s2$ is to add a conjunctive clause to the guard of action $a1$ in transition $B\{t4\}$, which reduce the set of circumstances under which transition $B\{t1\}$ and action $a1$ in transition $B\{t4\}$ execute.


reference to existing transition
name of strengthening clause
strengthen transition <T> with sc1: <P1>
strengthen action <A> with sc2: <P2>
reference to existing action
strengthening clause
strengthen transition B{main.t1} with s1: <P12>
strengthen action B{main.t4.a1} with s2: <P13>
state machine B{main}
B{s2}
B{r}
B{t1}: <WCE1> [F5{s1}: <P12>]
B{t2}: <WCE2>
B{s1}
B{t3}: <WCE3> [<P1>] / B{a1}: [<P2>] <WCA1>
B{t4}: <WCE4> [<P3>] /
B{a1}: [<P4> and F5{s2}: <P13>] <WCA2>
B{s1}
B{s2}

(a) Syntax

(b) Example: $F5$ module

(c) $F5$ composed with B (the strengthening clause is coloured in red)

Figure 3.31: Strengthening clauses

New preemptive transitions and actions A fragment can add a new preemptive transition to a state machine or a new preemptive action to a transition in an existing feature module. A preemptive transition preempts the execution of its target transitions, and a preemptive action preempts the execution of its target actions, when they are enabled in the same small step. A new preemptive transition (action) is specified like a new non-preemptive transition (action), as described above.

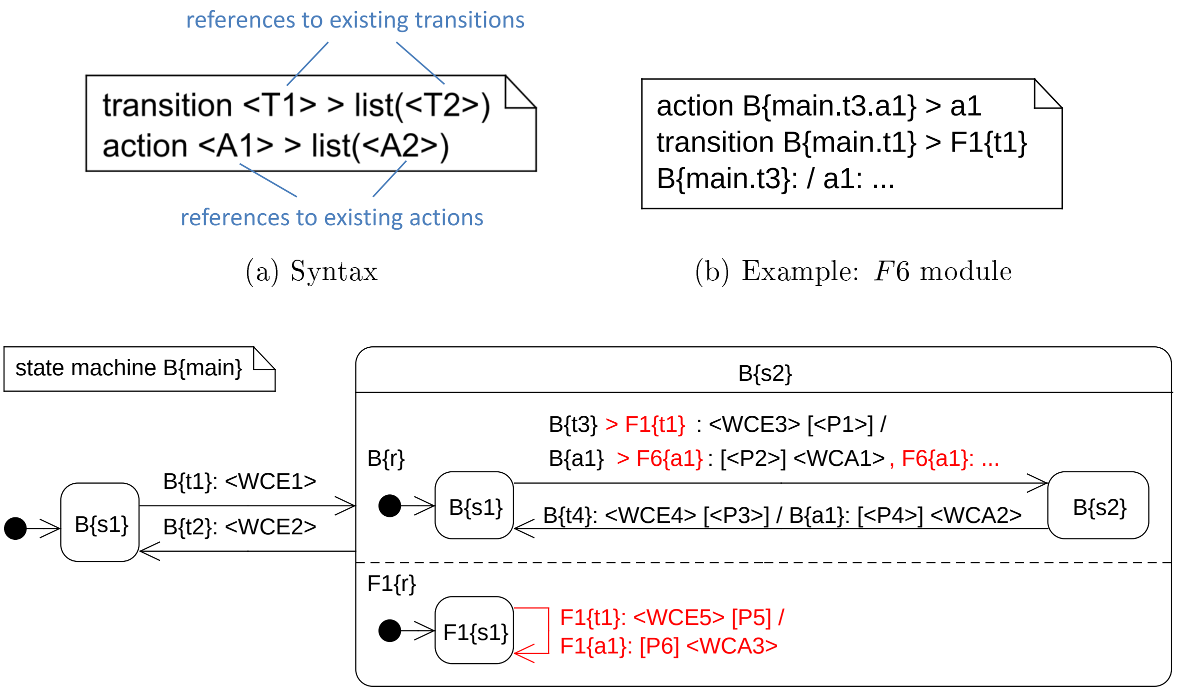
Fragments that trigger or prohibit state change Triggering the entry (exit) of an existing state s of a feature F causes an intended interaction with F: as a result of entering (exiting) s, the transitions and actions of F whose enabledness is affected by s execute under more (fewer) conditions. Examples of such transitions and actions include transitions that are reachable from s (i.e., transitions whose source state can be reached by zero or more transitions from s), override transitions with target transitions that are reachable from s, and transitions and actions that are guarded by *inState* predicates that check whether a machine is in state s. Conversely, prohibiting the entry (exit) of s causes the transitions and actions of F whose enabledness is affected s to execute under fewer (more) conditions.

Definition 3.4.14. *The following fragments intentionally trigger or prohibit the entry or* *exit of an existing state: A new (preemptive or non-preemptive) transition to (from) an existing state intentionally triggers the entry (exit) of the state; however, a state change occurs only if the new transition is not a self-transition. Furthermore, a fragment that triggers or prohibits an existing transition (i.e., a preemptive transition, or a weakening or strengthening clause) intentionally triggers or prohibits the exit (and entry) of the existing transitions' source (and destination) state, respectively; however, the state changes only if the existing transition is not a self-transition.*

3.4.6.3 Modelling Retrospective Intended Interactions

A new feature can add a *retrospective intended interaction* to an existing feature: specifically, a new feature can add priorities to an existing feature's requirements, such that the existing feature F's transition or action has priority over another (new or existing) feature G's transition or action. Such new priorities can be modelled by specifying G transitions as new target transitions of an existing F transition, or specifying G actions as new target actions of an existing F action[21]. Adding targets to a non-preemptive transition (action) x changes x into a priority transition (action). Figure 3.32a shows the syntax for specifying new target transitions and actions for an existing transition and action, respectively. In Figure 3.32b, a new feature $F6$ adds a new action $a1$ to transition $B\{t3\}$ and adds the new action $F6\{a1\}$ as a target action to B's action $a1$ of the same transition. $F6$ also adds $F1$'s transition $t1$ (see Figure 3.27b) as a target transition to B's transition $t1$. The composition of the feature modules for $F6$ and B is shown in Figure 3.32c.

[21] A target action added to an existing action a must belong to the same transition as a.


references to existing transitions
transition <T1> > list(<T2>)
action <A1> > list(<A2>)
references to existing actions
action B{main.t3.a1} > a1
transition B{main.t1} > F1{t1}
B{main.t3}: / a1: ...
state machine B{main}
B{s1}
B{t1}: <WCE1>
B{t2}: <WCE2>
B{s2}
B{r}
B{t3} > F1{t1} : <WCE3> [<P1>] /
B{a1} > F6{a1} : [<P2>] <WCA1> , F6{a1}: ...
B{s1}
B{t4}: <WCE4> [<P3>] / B{a1}: [<P4>] <WCA2>
B{s2}
F1{r}
F1{s1}
F1{t1}: <WCE5> [P5] /
F1{a1}: [P6] <WCA3>

(a) Syntax

(b) Example: $F6$ module

(c) $F6$ composed with B and $F1$ (the new targets and the transitions and actions that they reference are coloured in red)

Figure 3.32: State-machine fragments to specify new target transitions and actions

3.4.6.4 Modelling Enhancements of Enhancements

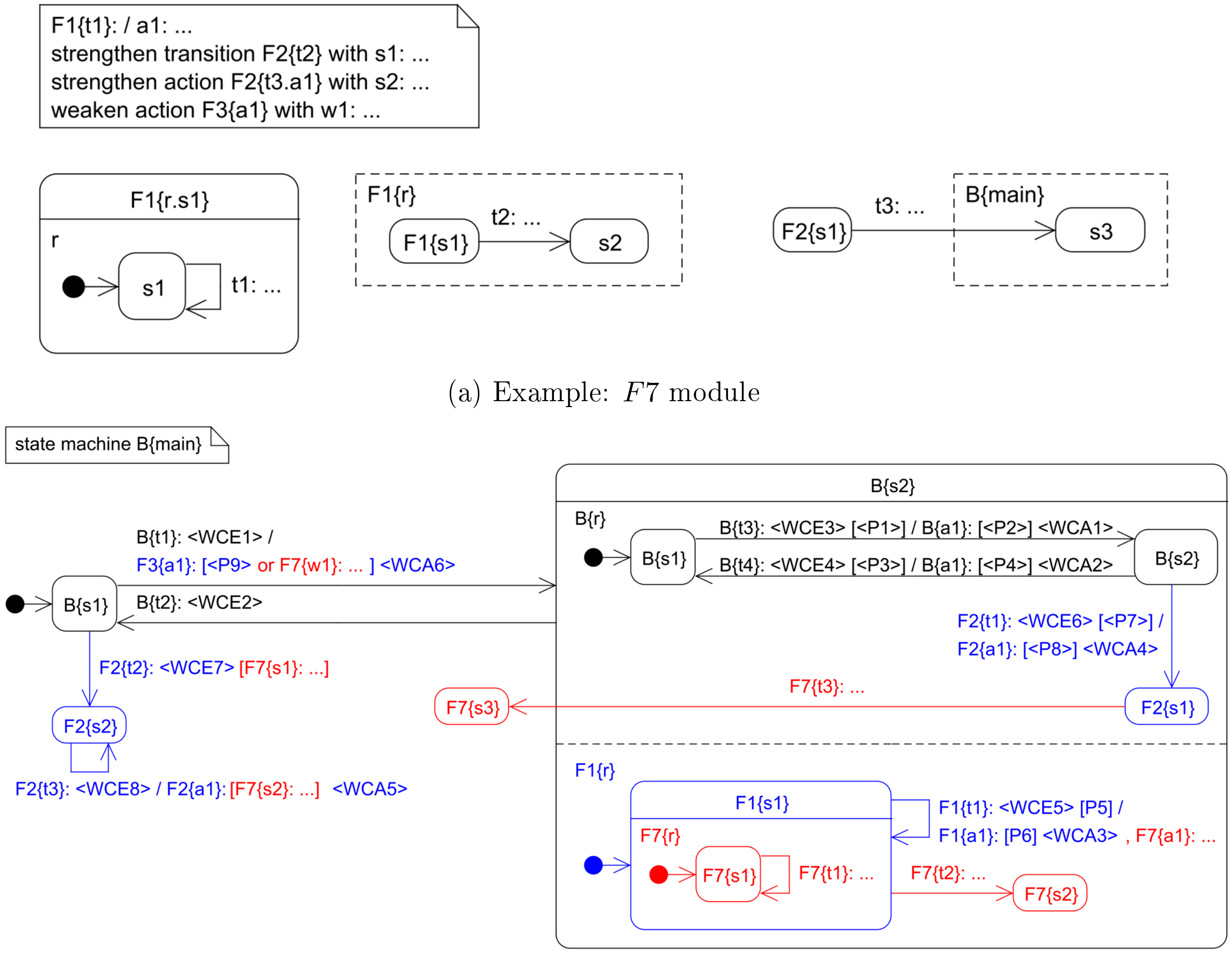
In FORML, an enhancement of an enhancement can be modelled using fragments that extend other fragments in existing feature modules. All of the fragments described above can be applied to existing fragments: for example, a state introduced by one fragment can, in a later fragment, serve as the source or destination of a new transition; a transition introduced by one fragment can later be extended with a new action; and so on. The syntax for fragments that extend existing fragments is the same as for fragments that extend existing state machines. However, the referencing of a model element that was introduced by a fragment is slightly different.

A fragment (e.g., a new region) and its sub-elements (e.g., the states in a new region) are referenced by their global names. The global name of a fragment is simply the fragment's name. The global name of an action a of a new transition t is $t.a$. A new state or region x_{new} introduces a sub state hierarchy rooted at x_{new}. Hence, the global name of a state or region x_{dsc} that is a descendant of x_{new} is the dot-separated list of node names along the path in the state hierarchy from x_{new}'s node to x_{dsc}'s node. To avoid ambiguity in expressions that reference fragment elements, the following uniqueness constraints apply to the names of such elements: the name of a fragment (i.e., a new region, transition, state, action, or clause) should be unique in its feature module, and the names of the sub-elements of fragments should follow the same uniqueness rules as in state machines (see Section 3.4.1.4).

For example, in Figure 3.33a, a new feature $F7$ extends several elements introduced by fragments in the feature modules of features $F1$, $F2$, and $F3$ (shown in Figures 3.27, 3.28, and 3.29, respectively): In feature $F1$, $F7$ extends state `F1{r.s1}` with a new region named r, adds a new transition $t2$ from $F1$'s state $s1$ to a new state $s2$, and extends $F1$'s transition $t1$ with a new action $a1$. In feature $F2$, $F7$ adds a new transition $t3$ from $F2$'s state $s1$ to a new state $s3$, strengthens $F2$'s transition $t2$ with a strengthening clause $s1$, and strengthens action $a1$ in $F2$'s transition $t3$ (referenced by the expression `F2{t3.a1}`) with a strengthening clause $s2$. In feature $F3$, $F7$ weakens $F3$'s action $a1$ with a weakening clause $w1$. The composition of $F7$'s feature module with the feature modules for B, $F1$, $F2$, and $F3$ is shown in Figure 3.33b.

As mentioned above, weakening and strengthening clauses can themselves be weakened or strengthened. Figure 3.34a shows the syntax for specifying a weakening clause that extends an existing strengthening or weakening clause. Such a weakening clause is named (to allow further weakening or strengthening), and the extended clause is referenced accordingly. For example, in Figure 3.34b, a new feature $F8$ weakens the strengthening clause $s2$ in $F5$'s feature module (shown in Figure 3.31b) with the weakening clause $w1$. The extended clause $s2$ is referenced by the expression `F5{s2}`. The composition of $F8$'s feature module with the feature modules for B and $F5$ is shown in Figure 3.34c.

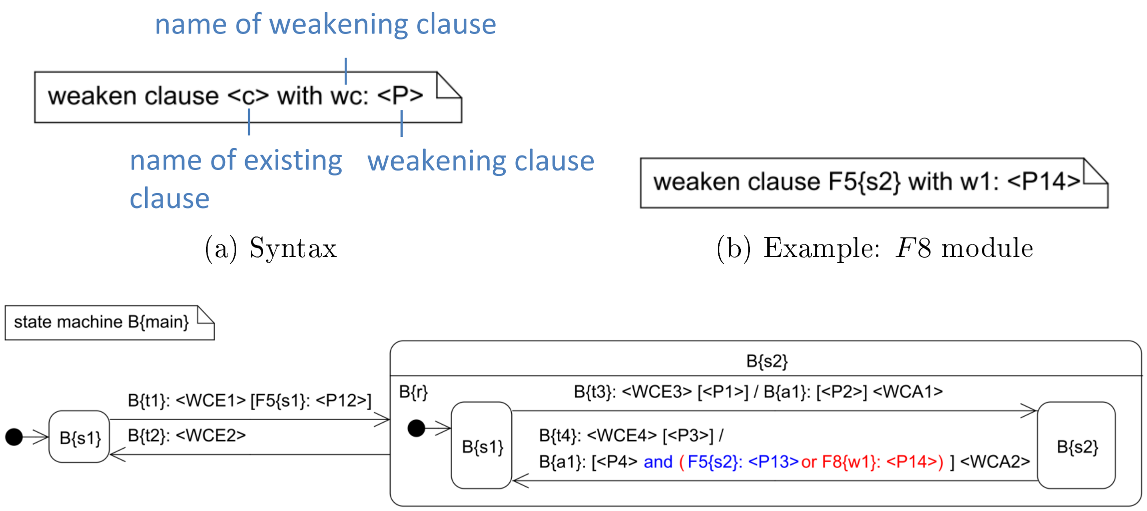
Figure 3.35a shows the syntax for specifying a strengthening clause that extends an existing strengthening or weakening clause; the syntax is analogous to that described above for a weakening clause. For example, in Figure 3.35b, a new feature $F9$ strengthens the weakening clause $w2$ in $F4$'s feature module (shown in Figure 3.30b) with the strengthening clause $s1$. The composition of $F9$'s feature module with the feature modules for B and $F4$ is shown in Figure 3.35c.


F1{t1}: / a1: ...
strengthen transition F2{t2} with s1: ...
strengthen action F2{t3.a1} with s2: ...
weaken action F3{a1} with w1: ...
F1{r.s1}
r
s1
t1: ...
F1{r}
t2: ...
F1{s1}
s2
t3: ...
B{main}
F2{s1}
s3
state machine B{main}
B{s2}
B{r}
B{t3}: <WCE3> [<P1>] / B{a1}: [<P2>] <WCA1>
B{s1}
B{t4}: <WCE4> [<P3>] / B{a1}: [<P4>] <WCA2>
B{s2}
B{t1}: <WCE1> /
F3{a1}: [<P9> or F7{w1}: ...] <WCA6>
B{s1}
B{t2}: <WCE2>
F2{t1}: <WCE6> [<P7>] /
F2{a1}: [<P8>] <WCA4>
F2{t2}: <WCE7> [F7{s1}: ...]
F7{t3}: ...
F7{s3}
F2{s1}
F2{s2}
F2{t3}: <WCE8> / F2{a1}: [F7{s2}: ...] <WCA5>
F1{r}
F1{s1}
F1{t1}: <WCE5> [P5] /
F1{a1}: [P6] <WCA3> , F7{a1}: ...
F7{r}
F7{s1}
F7{t1}: ...
F7{t2}: ...
F7{s2}

(a) Example: *F*7 module

(b) *F*7 composed with *B*, *F*1, *F*2, and *F*3 (fragments introduced by *F*1, *F*2, and *F*3 are coloured in blue and the extensions introduced by *F*7 are coloured in red)

Figure 3.33: New regions, transitions, and (weakening and strengthening) that extend existing fragments

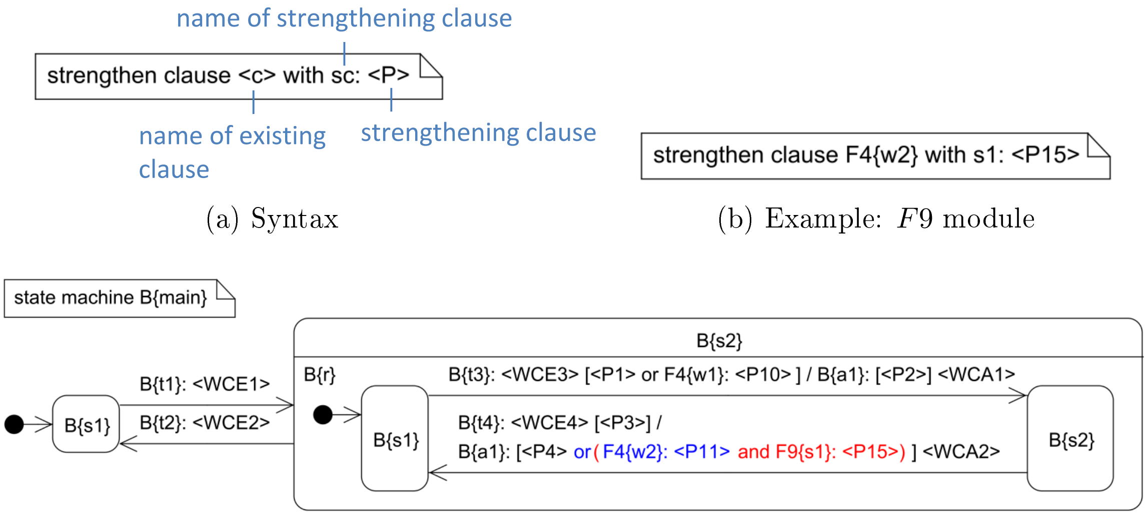

name of weakening clause
weaken clause <c> with wc: <P>
name of existing clause
weakening clause
weaken clause F5{s2} with w1: <P14>
state machine B{main}
B{s2}
B{r}
B{t1}: <WCE1> [F5{s1}: <P12>]
B{t2}: <WCE2>
B{s1}
B{t3}: <WCE3> [<P1>] / B{a1}: [<P2>] <WCA1>
B{t4}: <WCE4> [<P3>] /
B{a1}: [<P4> and (F5{s2}: <P13> or F8{w1}: <P14>)] <WCA2>
B{s1}
B{s2}

(a) Syntax

(b) Example: $F8$ module

(c) $F8$ composed with B and $F5$ (the extended clause is coloured in blue and the new weakening clause introduced by $F8$ is coloured in red)

Figure 3.34: State-machine fragment to specify weakening clauses that extend existing clauses


name of strengthening clause
strengthen clause <c> with sc: <P>
name of existing clause
strengthening clause
strengthen clause F4{w2} with s1: <P15>
state machine B{main}
B{s2}
B{r}
B{t1}: <WCE1>
B{t2}: <WCE2>
B{s1}
B{t3}: <WCE3> [<P1> or F4{w1}: <P10>] / B{a1}: [<P2>] <WCA1>
B{t4}: <WCE4> [<P3>] /
B{a1}: [<P4> or (F4{w2}: <P11> and F9{s1}: <P15>)] <WCA2>
B{s1}
B{s2}

(a) Syntax

(b) Example: $F9$ module

(c) $F9$ composed with B and $F4$ ($F4$'s extended clause is coloured in blue and the new strengthening clause introduced by $F9$ is coloured in red)

Figure 3.35: State-machine fragment to specify strengthening clauses that extend existing clauses

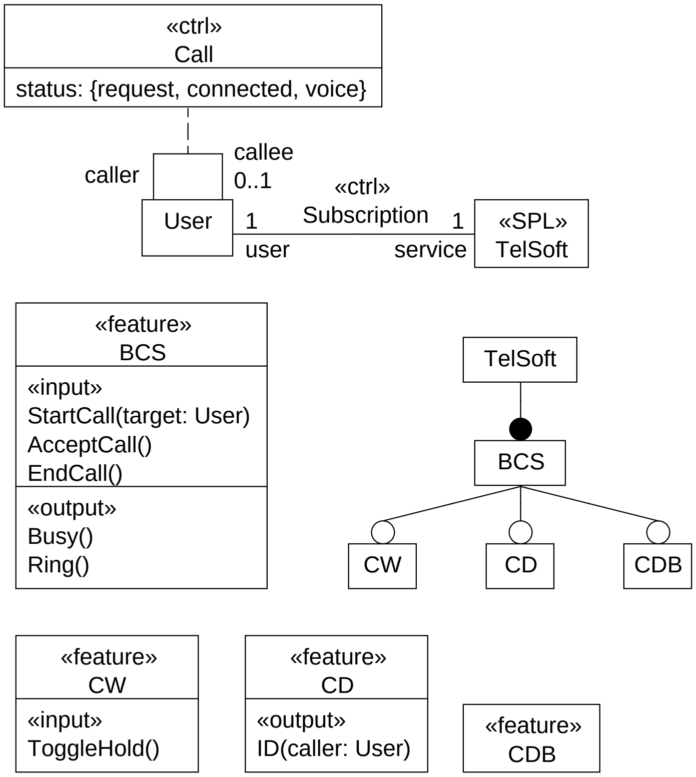


Figure 3.36: Partial *TelSoft* world model

3.4.6.5 Examples

This subsection presents the feature modules for the *TelSoft* features call waiting (CW), caller delivery (CD), and caller delivery blocking (CDB). Features CW and CD enhance BCS and are modelled as fragments that extend BCS's feature module; feature CDB enhances CD and is modelled as a fragment that extends the fragment in CD's feature module. The feature module for BCS (shown in Figure 3.16) is repeated in Figure 3.37 for ease of reference. All of the feature modules are associated with the partial *TelSoft* world model shown in Figure 3.36.

Call waiting: The feature module for CW is shown in Figure 3.38. An instance of CW becomes active only when its subscriber is in a call that has progressed to a voice connection. Thus, we model CW as a set of new behaviours (new transitions and states)

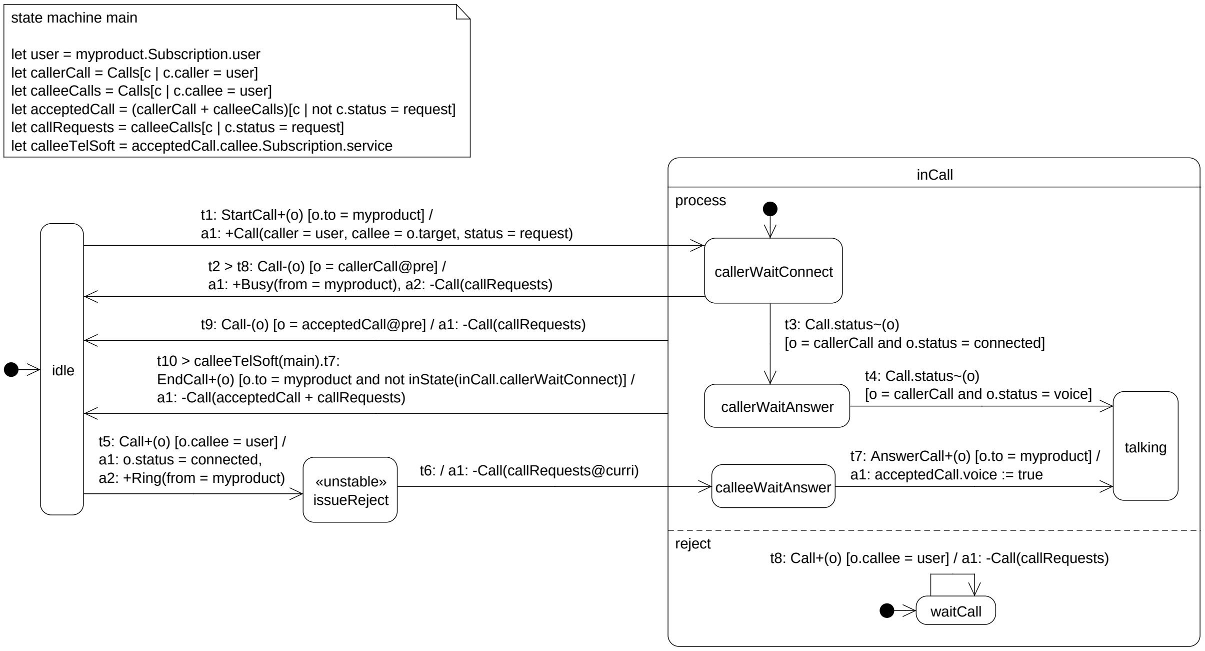

state machine main
let user = myproduct.Subscription.user
let callerCall = Calls[c | c.caller = user]
let calleeCalls = Calls[c | c.callee = user]
let acceptedCall = (callerCall + calleeCalls)[c | not c.status = request]
let callRequests = calleeCalls[c | c.status = request]
let calleeTelSoft = acceptedCall.callee.Subscription.service
inCall
process
idle
t1: StartCall+(o) [o.to = myproduct] /
a1: +Call(caller = user, callee = o.target, status = request)
callerWaitConnect
t2 > t8: Call-(o) [o = callerCall@pre] /
a1: +Busy(from = myproduct), a2: -Call(callRequests)
t9: Call-(o) [o = acceptedCall@pre] / a1: -Call(callRequests)
t3: Call.status~(o)
[o = callerCall and o.status = connected]
t10 > calleeTelSoft(main).t7:
EndCall+(o) [o.to = myproduct and not inState(inCall.callerWaitConnect)] /
a1: -Call(acceptedCall + callRequests)
callerWaitAnswer
t4: Call.status~(o)
[o = callerCall and o.status = voice]
talking
t5: Call+(o) [o.callee = user] /
a1: o.status = connected,
a2: +Ring(from = myproduct)
«unstable»
issueReject
t6: / a1: -Call(callRequests@curri)
calleeWaitAnswer
t7: AnswerCall+(o) [o.to = myproduct] /
a1: acceptedCall.voice := true
reject
t8: Call+(o) [o.callee = user] / a1: -Call(callRequests)
waitCall

Figure 3.37: BCS feature module (same as Figure 3.16)

that emanate from BCS's talking state. The new transitions *t1*, *t2*, *t4*, *t5*, and *t6* specify the conditions for activating and deactivating CW; and the new transition *t3* specifies that when CW is active, the subscriber can toggle the voice connection between the two calls.

CW has the following intended interactions with BCS in the subscriber's *TelSoft* product: (1) CW overrides BCS's requirement of rejecting an incoming call when the subscriber is already in a voice-connected call by connecting one such call (modelled by the override transition *t1*, whose target is BCS's transition *t8*). (2) If the subscriber makes an end-call request while CW is active, CW overrides BCS's requirement of ending all of the subscriber's calls by ending only the voice-connected call and establishing a voice-connection between the subscriber and the waiting party (modelled in CW by the override transition *t6*, whose target is BCS's transition *t10*).

Furthermore, CW specifies a retrospective intended interaction of BCS with CW in the *TelSoft* product of another user: when CW is active and the voice-connected call is ended by one of the call's parties, CW establishes a voice-connection between the subscriber and the waiting party (modelled by transitions $t5$ and $t6$). However, if the remote user of the waiting call ends the call (modelled by BCS's transition $t10$) at the same time, then BCS

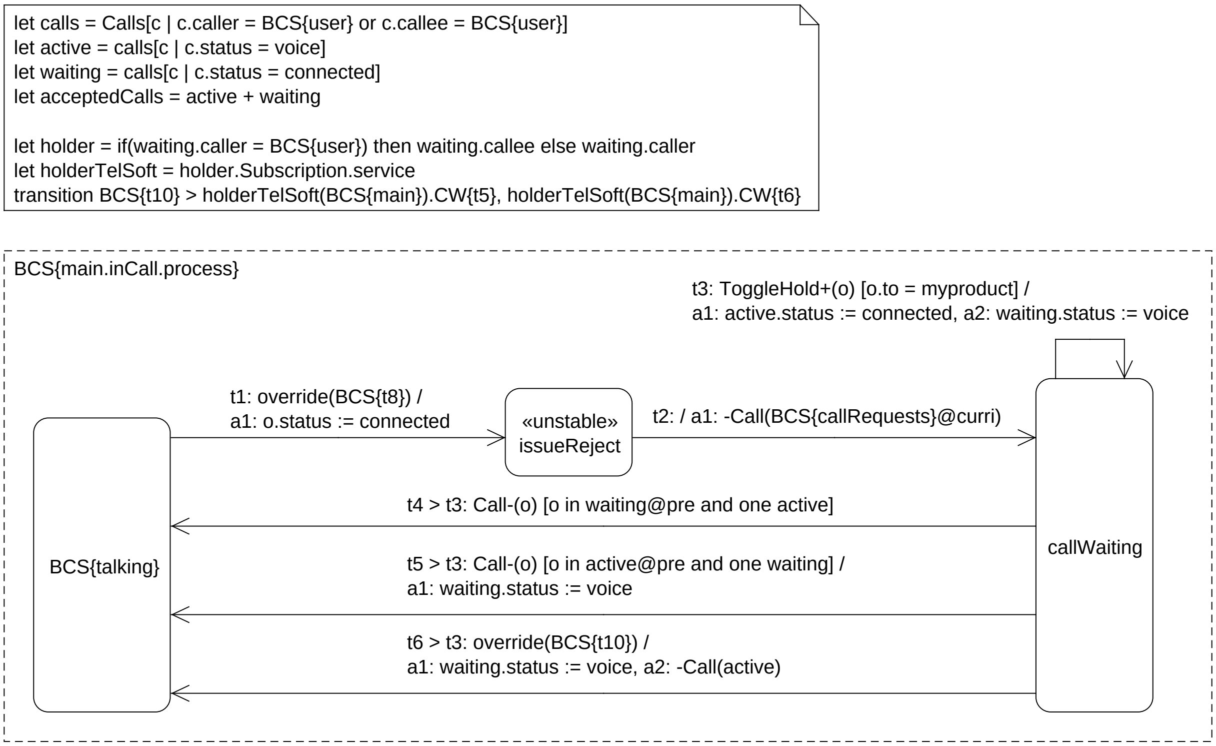

let calls = Calls[c | c.caller = BCS{user} or c.callee = BCS{user}]
let active = calls[c | c.status = voice]
let waiting = calls[c | c.status = connected]
let acceptedCalls = active + waiting
let holder = if(waiting.caller = BCS{user}) then waiting.callee else waiting.caller
let holderTelSoft = holder.Subscription.service
transition BCS{t10} > holderTelSoft(BCS{main}).CW{t5}, holderTelSoft(BCS{main}).CW{t6}
BCS{main.inCall.process}
t3: ToggleHold+(o) [o.to = myproduct] /
a1: active.status := connected, a2: waiting.status := voice
t1: override(BCS{t8}) /
a1: o.status := connected
«unstable»
issueReject
t2: / a1: -Call(BCS{callRequests}@curri)
t4 > t3: Call-(o) [o in waiting@pre and one active]
BCS{talking}
callWaiting
t5 > t3: Call-(o) [o in active@pre and one waiting] /
a1: waiting.status := voice
t6 > t3: override(BCS{t10}) /
a1: waiting.status := voice, a2: -Call(active)

Figure 3.38: CW feature module

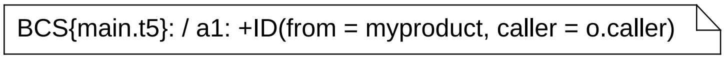
overrides CW and the call is ended (modelled by adding CW's transitions $t5$ and $t6$ as targets of BCS's transition $t10$[22]).

Caller Delivery: The feature module for CD is shown in Figure 3.39. When the subscriber is a callee connected to a new call (modelled by BCS's transition $t5$), CD delivers

[22] Specifically, CW includes the fragment

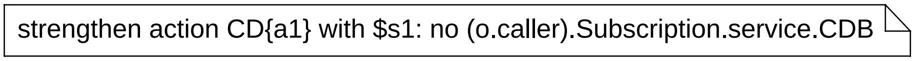
transition BCS{t10} > holderTelSoft(BCS{main}).CW{t5}, holderTelSoft(BCS{main}).CW{t6}

which can be broken down as follows: Let $ts1$ be a *TelSoft* product that has been placed on hold by CW in a remote *TelSoft* product $ts2$ (i.e., $ts1$ is in a connected call with $ts2$). In product $ts1$'s $BCS\{main\}$ machine, the macro holderTelSoft evaluates to $ts2$; hence, the expressions holderTelSoft(BCS{main}).CW{t5} and holderTelSoft(BCS{main}).CW{t6} evaluate to transitions $CW\{t5\}$ and $CW\{t6\}$ in product $ts2$'s $BCS\{main\}$ machine, respectively. As a result, the fragment states that transition $BCS\{t10\}$ in $ts1$'s $BCS\{main\}$ machine takes priority over transitions $CW\{t5\}$ and $CW\{t6\}$ in $ts2$'s $BCS\{main\}$ machine.



```
BCS{main.t5}: / a1: +ID(from = myproduct, caller = o.caller)
```

Figure 3.39: CD feature module



```
strengthen action CD{a1} with $s1: no (o.caller).Subscription.service.CDB
```

Figure 3.40: CDB feature module

the caller's identification to the subscriber. This added requirement is modelled by a new action $a1$ added to BCS's transition $t5$.

Caller Delivery Blocking: The feature module for CDB is shown in Figure 3.40. When the subscriber is a caller initiating a new call, CDB prevents the subscriber's identification from being delivered to a callee who subscribes to CD. This intended interaction of CDB with CD is modelled by strengthening the guard of CD's action $a1$ with a condition stating that CD's behaviour applies only if the caller is not subscribed to CDB. Note that the strengthening clause $s1$ in Figure 3.40 has a $ prefix. The prefix indicates that the clause in not intended to prohibit behaviours in the subscriber's product, but rather to prohibit behaviours in other related products: Specifically, $s1$ does not prohibit action $CD\{a1\}$ in the *TelSoft* product of the CDB subscriber, but in the *TelSoft* product of a remote CD subscriber. The semantics of feature-module composition is affected by the $ prefix, as explained in Section 3.5.

Other examples Additional examples of *TelSoft* and *AutoSoft* features that are modelled as fragments can be found in Appendix A. In particular, for an example of a new region, see the model of the *AutoSoft* feature cruise control (CC); for an example of a weakening condition, see the model of the *AutoSoft* feature lane change control (LXC); and for an example of a preemptive action, see the model of the *TelSoft* feature reverse charging (RC).

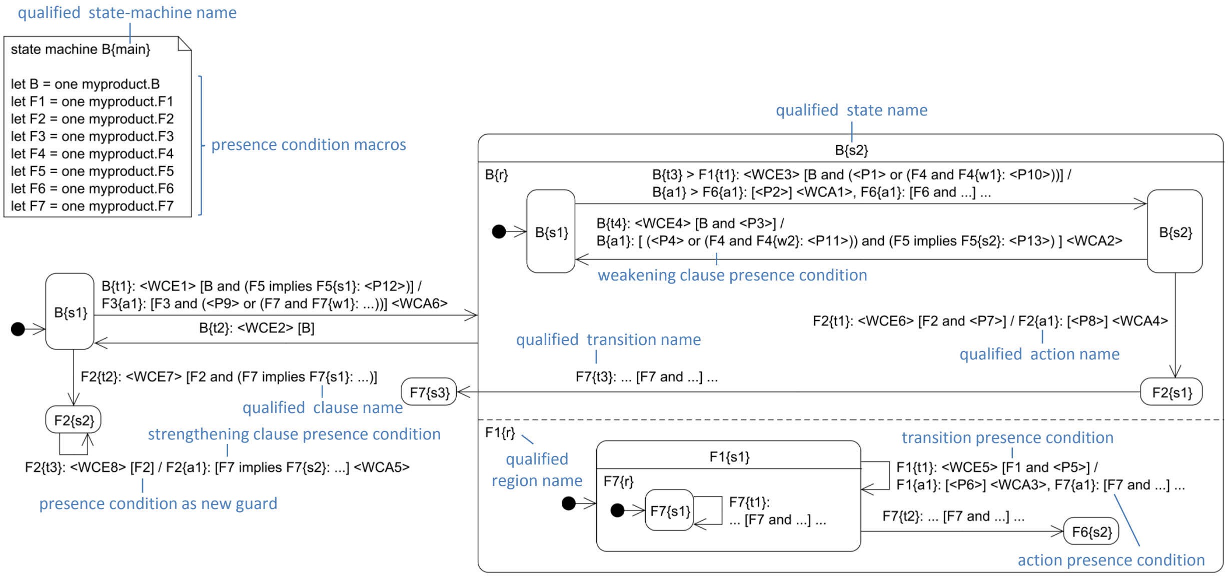

qualified state-machine name
state machine B{main}
let B = one myproduct.B
let F1 = one myproduct.F1
let F2 = one myproduct.F2
let F3 = one myproduct.F3
let F4 = one myproduct.F4
let F5 = one myproduct.F5
let F6 = one myproduct.F6
let F7 = one myproduct.F7
presence condition macros
B{s1}
B{t1}: <WCE1> [B and (F5 implies F5{s1}: <P12>)] /
F3{a1}: [F3 and (<P9> or (F7 and F7{w1}: ...))] <WCA6>
B{t2}: <WCE2> [B]
F2{t2}: <WCE7> [F2 and (F7 implies F7{s1}: ...)]
qualified clause name
F7{s3}
F2{s2}
strengthening clause presence condition
F2{t3}: <WCE8> [F2] / F2{a1}: [F7 implies F7{s2}: ...] <WCA5>
presence condition as new guard
qualified state name
B{s2}
B{r}
B{t3} > F1{t1}: <WCE3> [B and (<P1> or (F4 and F4{w1}: <P10>))] /
B{a1} > F6{a1}: [<P2>] <WCA1>, F6{a1}: [F6 and ...] ...
B{s1}
B{t4}: <WCE4> [B and <P3>] /
B{a1}: [(<P4> or (F4 and F4{w2}: <P11>)) and (F5 implies F5{s2}: <P13>)] <WCA2>
B{s2}
weakening clause presence condition
F2{t1}: <WCE6> [F2 and <P7>] / F2{a1}: [<P8>] <WCA4>
qualified transition name
qualified action name
F7{t3}: ... [F7 and ...] ...
F2{s1}
F1{r}
qualified
region name
F1{s1}
F7{r}
F7{s1}
F7{t1}:
... [F7 and ...] ...
transition presence condition
F1{t1}: <WCE5> [F1 and <P5>] /
F1{a1}: [<P6>] <WCA3>, F7{a1}: [F7 and ...] ...
F7{t2}: ... [F7 and ...] ...
F6{s2}
action presence condition

Figure 3.41: The composition of the feature modules of the features *B* (Figure 3.26) and *F1* to *F7* (Figures 3.27 to 3.33)

3.5 Feature Composition

Although features are modelled as separate feature modules, the modeller will eventually want to visualize and (manually or automatically) analyze feature combinations corresponding to products of the SPL. To do so, the modeller must derive models of feature combinations by composing feature modules using one of two approaches: (1) composing a set of feature modules that correspond to a particular feature configuration to obtain a model of the requirements of a single product; or (2) composing all of the feature modules to obtain a model of the whole SPL and all of its derivable products. This thesis focuses on the second approach for the following reason: It has been shown that a model of the whole SPL enables more efficient analyses of the SPL's set of products, because the analysis can exploit the commonalities among different products [7, 24, 41, 86, 8]. In contrast, given the first approach to feature composition, the requirements analyst would have to derive and analyze each individual product.

The result of composing a set of feature modules is the feature modules' set of parallel state machines that have been extended with the feature modules' set of fragments. For example, Figure 3.41 shows the result of composing the feature modules of the feature *B* (see Figure 3.26) and features *F*1 to *F*7 (see Figures 3.27 to 3.33) from Section 3.4.6. The

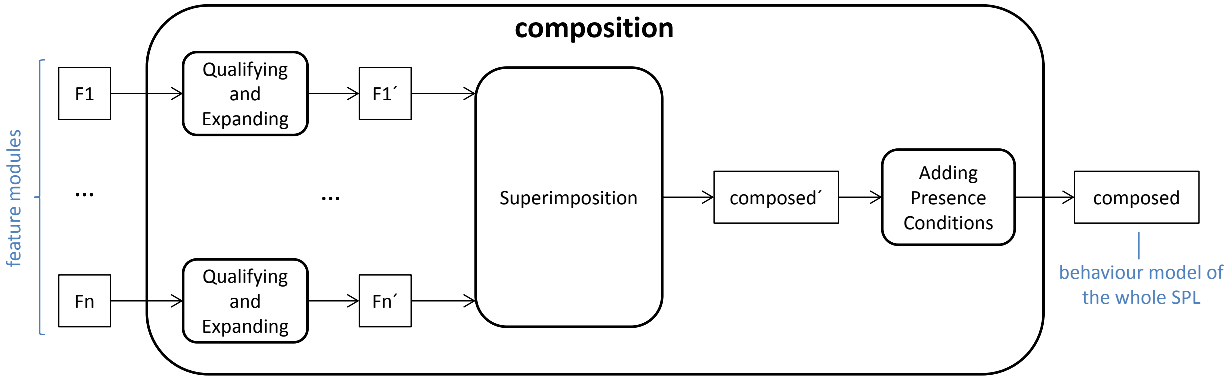


Figure 3.42: The process composing feature modules into an SPL model (boxes with square corners represent models and boxes with rounded corners represent processes)

composition process comprises the following phases (depicted graphically in Figure 3.42):

Qualifying and expanding: This phase preprocesses the feature modules for integration by *qualifying element names* and *expanding fragment references* in each feature module. Many model elements have names, including names of state machines, states, transitions, actions, weakening or strengthening clauses, and macros. To avoid name clashes between elements that are introduced by different feature modules (e.g., features B and F both introduce a region r in B's state $s2$), the name n of an element introduced by a feature F is qualified with the feature's name, resulting in the name $F\{n\}$. For example, in Figure 3.41, the state machine $main$ and state $s2$ introduced by B are named $B\{main\}$ and $B\{s2\}$, respectively; $F1$'s region r is named $F1\{r\}$; $F2$'s action $a1$ is named $F2\{a1\}$; and $F6$'s strengthening clause $s1$ is named $F6\{s1\}$[23].

If an *inState* predicate, preemptive transition or action, or macro expression references a machine element introduced by a state-machine fragment (the syntax for referencing machine elements is given in Section 3.4.1.4), the reference is expanded into the element's name in the composed model, with each component of the name qualified by the feature that introduced the component. For example, suppose that feature $F1$ extends B's state $s2$ with a region r, and suppose that an *inState* predicate inState(r.s1) refers to state $s1$

[23] Recall that a fragment's reference to a state-machine element in another feature module also takes the form $F\{n\}$ (see Section 3.4.6). However, in such a reference, n is the element's *global* name (e.g., $B\{main.s2\}$ refers to state $s2$ in state machine $main$ in B's feature module). In contrast, in the qualified name of an element, n is simply the element's name (e.g., the qualified name of B's state $s2$ is simply $B\{s2\}$).

in the new region; the predicate is expanded to $\mathsf{inState(B\{s2\}.F1\{r.s1\})}$ in the composed model[24].

Example: In preprocessing CW's feature module (see Figure 3.38), the priority transition $t4$'s name is qualified to become $CW\{t4\}$, and $t4$'s reference to its target transition $t3$ is expanded to $t3$'s qualified name $CW\{t3\}$.

Superimposition: This phase integrates the state machines and fragments of the feature modules being composed: the features' state machines are composed in parallel and are extended with the features' fragments. Extending a state machine with fragments is straightforward, with the following exception: if a set of weakening clauses $w_1 \cdots w_n$ and a set of strengthening clauses $s_1 \cdots s_m$ extend the same condition c, the result of composing these extensions in the composed model is the condition

$$(c \text{ or } w_1 \text{ or } \cdots \text{ or } w_n) \text{ and } s_1 \text{ and } \cdots \text{ and } s_m$$

If the weakening and strengthening clauses are applied to a transition or action that has no guard condition, the resulting guard condition is

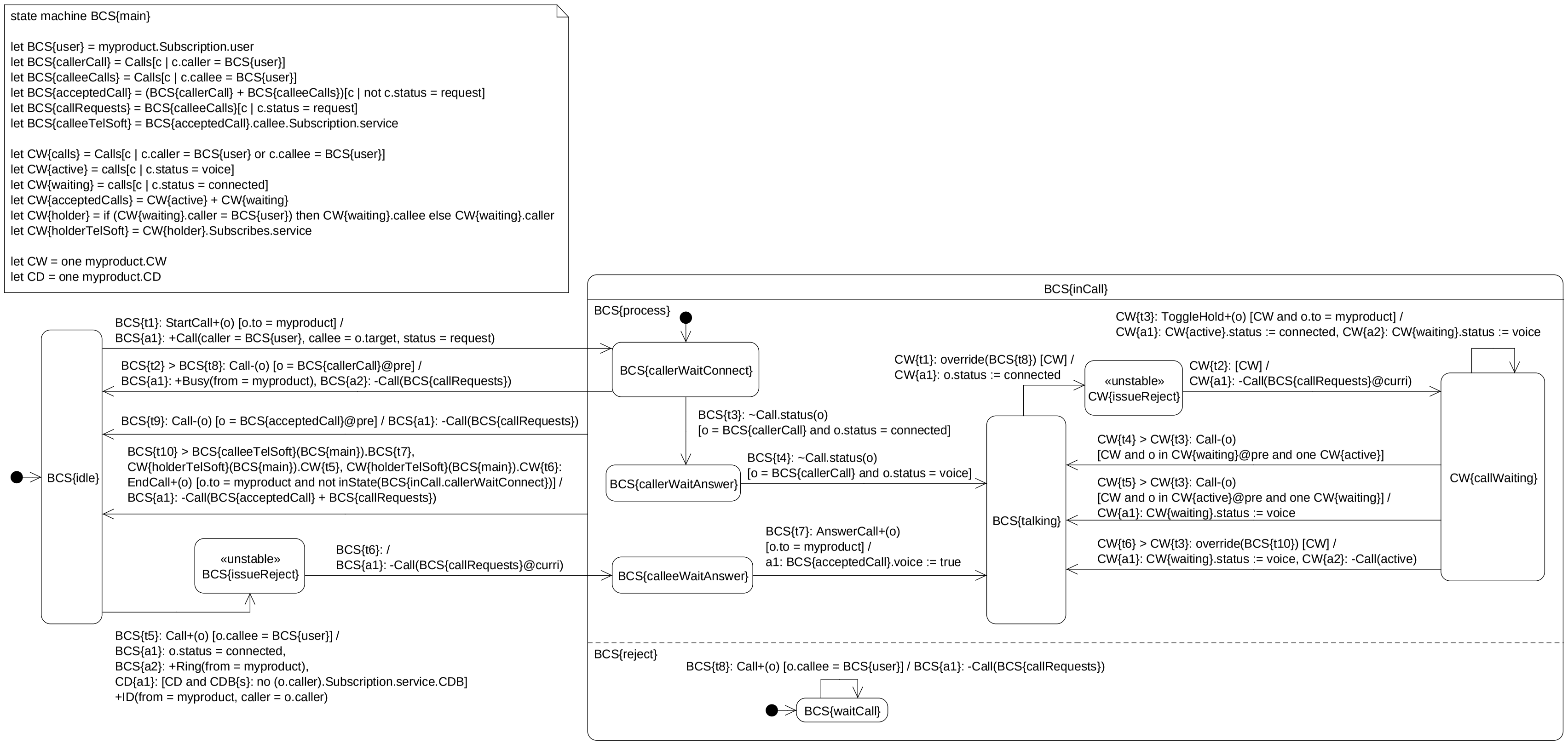
$$(w_1 \text{ or } \cdots \text{ or } w_n) \text{ and } s_1 \text{ and } \cdots \text{ and } s_m$$

For example, $F5$'s strengthening clause $s2$: <*P13*> and $F4$'s weakening clause $w2$: <*P11*> both extend the guard condition <*P4*> of B's action $a1$ in transition $t4$; the result is the following new guard condition:

$$(<\!P4\!> \text{ or } \mathsf{F4\{w2\}} : <\!P11\!>) \text{ and } \mathsf{F5\{s2\}} : <\!P13\!>$$

The rule for composing weakening and strengthening clauses actualizes a canonical guard expression that does not depend on the order in which features are created, evolved, or composed in the SPL. Moreover, the rule gives strengthening clauses priority over weakening clauses: strengthening clauses often model resolutions to undesired feature interactions such as conflicts, whereas weakening clauses liberalize the enabling conditions of behaviours in other features. We view resolutions to undesired interactions as being more critical than liberalizing the enabling conditions of behaviours. As is explained in Section 3.5.1, the proposed canonical composition rule for weakening and strengthening clauses helps to ensure the commutativity and associativity of superimposition. The superimposition phase is described more formally in Section 3.5.1.

[24] Recall from Section 3.4.1.2 that in a predicate *inState*(n), n is a state's local name within its state machine.



```
state machine BCS{main}

let BCS{user} = myproduct.Subscription.user
let BCS{callerCall} = Calls[c | c.caller = BCS{user}]
let BCS{calleeCalls} = Calls[c | c.callee = BCS{user}]
let BCS{acceptedCall} = (BCS{callerCall} + BCS{calleeCalls})[c | not c.status = request]
let BCS{callRequests} = BCS{calleeCalls}[c | c.status = request]
let BCS{calleeTelSoft} = BCS{acceptedCall}.callee.Subscription.service

let CW{calls} = Calls[c | c.caller = BCS{user} or c.callee = BCS{user}]
let CW{active} = calls[c | c.status = voice]
let CW{waiting} = calls[c | c.status = connected]
let CW{acceptedCalls} = CW{active} + CW{waiting}
let CW{holder} = if (CW{waiting}.caller = BCS{user}) then CW{waiting}.callee else CW{waiting}.caller
let CW{holderTelSoft} = CW{holder}.Subscribes.service

let CW = one myproduct.CW
let CD = one myproduct.CD
```

BCS{idle}
BCS{t1}: StartCall+(o) [o.to = myproduct] /
BCS{a1}: +Call(caller = BCS{user}, callee = o.target, status = request)
BCS{t2} > BCS{t8}: Call-(o) [o = BCS{callerCall}@pre] /
BCS{a1}: +Busy(from = myproduct), BCS{a2}: -Call(BCS{callRequests})
BCS{t9}: Call-(o) [o = BCS{acceptedCall}@pre] / BCS{a1}: -Call(BCS{callRequests})
BCS{t10} > BCS{calleeTelSoft}(BCS{main}).BCS{t7},
CW{holderTelSoft}(BCS{main}).CW{t5}, CW{holderTelSoft}(BCS{main}).CW{t6}:
EndCall+(o) [o.to = myproduct and not inState(BCS{inCall.callerWaitConnect})] /
BCS{a1}: -Call(BCS{acceptedCall} + BCS{callRequests})
«unstable»
BCS{issueReject}
BCS{t6}: /
BCS{a1}: -Call(BCS{callRequests}@curri)
BCS{t5}: Call+(o) [o.callee = BCS{user}] /
BCS{a1}: o.status = connected,
BCS{a2}: +Ring(from = myproduct),
CD{a1}: [CD and CDB{s}: no (o.caller).Subscription.service.CDB]
+ID(from = myproduct, caller = o.caller)
BCS{inCall}
BCS{process}
BCS{callerWaitConnect}
BCS{t3}: ~Call.status(o)
[o = BCS{callerCall} and o.status = connected]
BCS{callerWaitAnswer}
BCS{t4}: ~Call.status(o)
[o = BCS{callerCall} and o.status = voice]
BCS{calleeWaitAnswer}
BCS{t7}: AnswerCall+(o)
[o.to = myproduct] /
a1: BCS{acceptedCall}.voice := true
BCS{talking}
CW{t1}: override(BCS{t8}) [CW] /
CW{a1}: o.status := connected
«unstable»
CW{issueReject}
CW{t2}: [CW] /
CW{a1}: -Call(BCS{callRequests}@curri)
CW{t3}: ToggleHold+(o) [CW and o.to = myproduct] /
CW{a1}: CW{active}.status := connected, CW{a2}: CW{waiting}.status := voice
CW{callWaiting}
CW{t4} > CW{t3}: Call-(o)
[CW and o in CW{waiting}@pre and one CW{active}]
CW{t5} > CW{t3}: Call-(o)
[CW and o in CW{active}@pre and one CW{waiting}] /
CW{a1}: CW{waiting}.status := voice
CW{t6} > CW{t3}: override(BCS{t10}) [CW] /
CW{a1}: CW{waiting}.status := voice, CW{a2}: -Call(active)
BCS{reject}
BCS{t8}: Call+(o) [o.callee = BCS{user}] / BCS{a1}: -Call(BCS{callRequests})
BCS{waitCall}

Figure 3.43: The composition of the feature modules of the *TelSoft* features BCS, CW, CD, and CDB

Example: The composition of the feature modules of the *TelSoft* features BCS (see Figure 3.37), CW (see Figure 3.38), CD (see Figure 3.39), and CDB (see Figure 3.40) is shown in Figure 3.43. In this case, the result of the superimposition phase is a single machine (from BCS) extended by fragments (from CW, CD, and CDB).

Adding presence conditions: Recall that a feature module is a schema to be instantiated for each product, to reflect the feature's contribution to that product's requirements. After composing the feature modules, the resulting SPL model is a schema for a product with all features – even if the feature model in the world model excludes such a product. To enable specializing the SPL model for a particular product, we use *presence conditions* that make each requirement conditional on whether the feature that introduced the requirement is present in the product. A presence condition is introduced for each optional feature.

Definition 3.5.1. *Each presence condition is a boolean variable that is named after its corresponding feature.*

The transitions, actions, and weakening and strengthening clauses that are introduced by an optional feature are strengthened with the feature's presence condition.

- To augment a transition with a presence condition F, the presence condition is added in conjunction to the transition's guard condition gc, resulting in a new guard F and gc; if the transition has no guard condition, the presence condition becomes the transition's new guard condition.
- An action is augmented with a presence condition if it is introduced as a fragment that extends another feature's transition. An action that is introduced as part of a transition is already strengthened with the transition's presence condition.

The augmentation of clauses with presence conditions is less obvious:

- A presence condition F augments a weakening clause w (a disjunct) by conjunction, resulting in a new clause F and w; the new clause is false when F is false and is equal to w when F is true.
- Analogously, F augments a strengthening clause s (a conjunct) by being added as an antecedent to s, resulting in a new clause F implies s; the new clause is true when F is false and is equal to s when F is true.

A unique presence condition F is instantiated for each F object of each product object in the world; the values of the presence conditions associated with each product's feature configuration should conform to the feature model.

Consider the composed behaviour model in Figure 3.41. All of the features are assumed to be optional features in the SPL, so all have presence conditions. For example, macro $\mathsf{B} =$ $\mathsf{one\ myproduct.B}$ is the presence condition for feature B. The macro states that a machine's product (referred to by the object variable *myproduct*) has exactly one B feature[25]. The transitions, actions, and clauses of each feature are augmented with the feature's presence condition; for example

- $F1$'s presence condition strengthens the guard condition of $F1$'s transition $t1$.
- $F1$'s transition $t1$ includes an action $F6\{a1\}$ that was introduced by feature $F6$; this action is strengthened by $F6$'s presence condition.
- $F4$'s weakening clause $w2$ in the guard condition of action $B\{main.t4.a1\}$ is strengthened by $F4$'s presence condition.
- $F7$'s strengthening clause $s2$ in the guard condition of action $B\{main\}.F2\{t3.a1\}$ is strengthened by $F7$'s presence condition.

There are cases where an element (transition, action, or clause) introduced by an optional feature constrains a *different* product in the world. For example, the CDB feature in a caller's *TelSoft* product overrides the effects of the CD feature in the callee's *TelSoft* product. In this case, augmenting the CDB element with the feature's presence condition will erroneously constrain the feature's product and not to the intended product. In FORML, an element's name can be prefaced with \$ to indicate that the element should not be augmented with a presence condition upon composition. For example, CDB's strengthening clause $s1$ (see Figure 3.40) is intended to override CD's action $a1$ (see Figure 3.39) in the callee's *TelSoft* product. However, applying CDB's presence condition to $s1$ will cause $s1$ to apply in the caller's *TelSoft* product. To avoid this error, $s1$ is prefaced with \$, so that no CDB presence condition is introduced. Instead, $s1$ realizes a unique form of presence condition for CDB:

$$\mathsf{no(o.caller).Subscription.service.CDB}$$

which relies on CD's presence condition to identify products that have CD, and states that in such products, CD's action $a1$ should not be performed if the caller has CDB.

[25] The quantifier one is needed, because $\mathsf{myproduct.B}$ evaluates to a set that is empty if *myproduct* has no B feature and contains a single B feature object otherwise.

Weakening and strengthening clauses have a lower precedence than presence conditions. Hence, if a presence condition F is applied to the same condition c as a set of weakening clauses $w_1 \cdots w_n$ and a set of strengthening clauses $s_1 \cdots s_m$, the resulting conditions are

$$F \textsf{ and } ((c \textsf{ or } w_1 \textsf{ or } \cdots \textsf{ or } w_n) \textsf{ and } s_1 \textsf{ and } \cdots \textsf{ and } s_m)$$

or

$$F \textsf{ and } ((w_1 \textsf{ or } \cdots \textsf{ or } w_n) \textsf{ and } s_1 \textsf{ and } \cdots \textsf{ and } s_m)$$

if c is a guard condition or weakening clause and

$$F \textsf{ implies } ((c \textsf{ or } w_1 \textsf{ or } \cdots \textsf{ or } w_n) \textsf{ and } s_1 \textsf{ and } \cdots \textsf{ and } s_m)$$

if c is a strengthening clause.

Example: In the composed *TelSoft* state machine shown in Figure 3.43, CW's transitions (e.g., $CW\{t3\}$) and CD's action $a1$ in transition $BCS\{t5\}$ are augmented with the presence conditions of CW and CD, respectively. However, note how CDB's strengthening clause $s1$, applied to CD's action $a1$, is not augmented with CDB's presence condition, since $s1$ is prefaced with $ in CDB's module.

3.5.1 Superimposition

This section gives a precise semantics for the main operation in composing a set of feature modules: the *superimposition* of the feature modules' state machines and fragments, resulting in an integrated set of parallel state machines that have been extended with fragments. The superimposition of a set of (qualified and expanded) feature modules $F'_1 \ \ldots \ F'_n$ is denoted by the expression

$$F'_1 \bullet \ldots \bullet F'_n$$

where $\bullet$ denotes the *superimposition operator*, which takes two feature modules as operands, and returns a composed feature module as a result. As is shown later in this section, the superimposition operator is *commutative*, which means that the order in which feature modules are superimposed does not affect the result; superimposition is also *associative*, which means that all parenthesizations of the above expression have the same result.

Superimposition takes as input an abstract-syntax-tree representation of the feature modules, called feature structure trees (FSTs) [6]. The following describes the FSTs of _behaviour-model_ ::= _state-machine_+ __macro__*

state-machine ::= _state_* __initial-state__? _transition_*

state ::= _region_*

region ::= _state_+ __initial-state__?

transition ::= __source__? __destination__? (__trigger__ | __override-spec__)? __targets__* _condition_? _action_*

action ::= __override-spec__? __targets__* _condition_? __WCA__?

condition ::= __predicate__? _clause_*

clause ::= __predicate__? __clause-type__? _clause_*

Figure 3.44: Abstract syntax of feature-module FSTs: *nonterminal* and **terminal** symbols are denoted with different fonts. + denotes one or more, * denotes zero or more, and ? denotes zero or one repetitions of the preceding syntactic element.

FORML feature modules and presents an implementation of the superimposition operator $\bullet$ as a *merge operation* over FST nodes. Our merge operation is a simplified version of Apel et al.'s generic merge operation [6]. The simplifications that we made to the merge operation ensure that the merge operation is commutative and associative.

3.5.1.1 FSTs of FORML Feature Modules

The FST of a FORML feature module is an abstract syntax tree of the feature module, based on the grammar shown in Figure 3.44. In the grammar, *nonterminal* and **terminal** symbols are denoted with different fonts and symbol multiplicities are specified as follows: + denotes one or more, $*$ denotes zero or more, and ? denotes zero or one repetitions of the preceding syntactic element. For example, a *behaviour-model* nonterminal expands to one or more *state-machine* terminals and zero or more *macro* terminals; and a *region* nonterminal expands to one or more *state* terminals and zero or one *initial-state* terminal[26]. Each node in an FST specifies the type of the element that it represents (which corresponds to a terminal or nonterminal symbol in the grammar), the element's qualified name (if the element is named), and its content. For example, Figures 3.45a, 3.45b, 3.46, and 3.47 show the feature modules and corresponding FSTs of the features B, $F1$, $F4$, and $F8$ from Section 3.4.6, respectively. To illustrate FST nodes for element types not present in these features (e.g., macros, override specifications), Figure 3.48 gives another pedagogical

[26] Note that some mandatory state-machine elements are represented by optional FST nodes (e.g. the mandatory source state of a transition is represented by an optional *source* node). This is because FSTs represent not only complete state machines, but also fragments of state machines. For example, feature $F4$'s FST shown in Figure 3.46 contains a *transition* node $B\{t4\}$: *tran* without a *source* node, which represents the transition extended by a weakening clause in $F4$'s feature module.

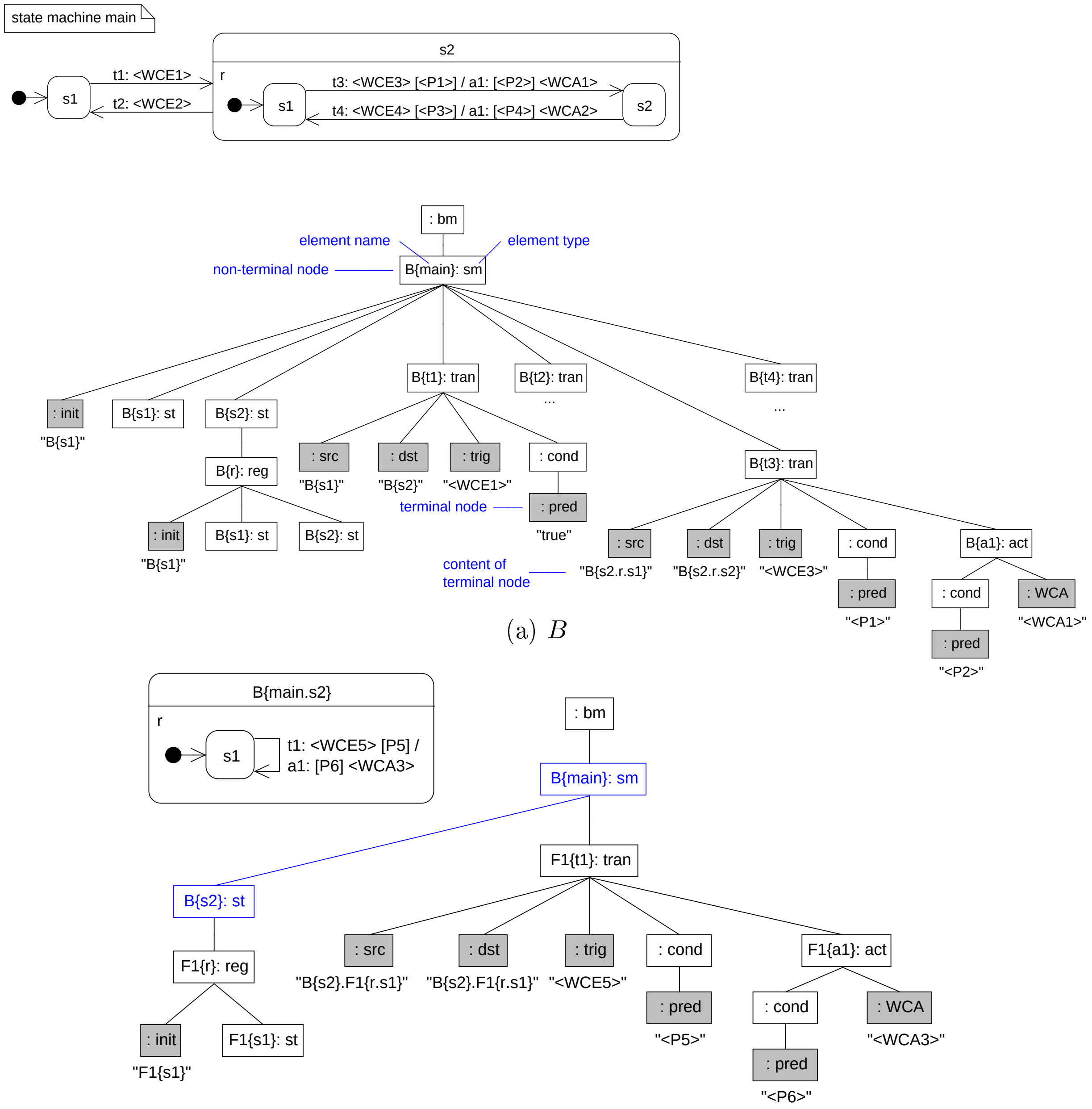
Abbreviation	Node Type
bm	*behaviour-model*
sm	*state-machine*
init	*initial-state*
st	*state*
tran	*transition*
src	*source*
dst	*destination*
trig	*trigger*
over	*override-spec*
cond	*condition*
pred	*predicate*
act	*action*

Table 3.21: Abbreviations for the FST node types used in Figures 3.45 to 3.51

example of a feature F. Table 3.21 gives abbreviations for node types that are used in the figures and in the remainder of this thesis.

In an FST, a terminal node represents a model element that cannot be extended by feature fragments (e.g., the initial-state designation of a state machine, a macro). The content of a terminal node is a string literal that appears directly below the node. For example, in Figure 3.45a, node $B\{main\}\!:\!sm$ has a terminal child node *init* whose content designates $B\{s1\}$ as the initial state of B's state machine $main$. A nonterminal node represents either a compound element (e.g., a state machine, region, transition) or an atomic element that can be extended (e.g., a basic state, which could be extended by a future feature to include regions and sub machines). The content of a nonterminal node comprises the node's FST subtrees. For example, in Figure 3.45a, the content of $B\{main\}\!:\!sm$ are the FST subtrees representing the state hierarchy and transitions of B's state machine $main$.

All feature modules have the same root node: an unnamed *bm* node representing the behaviour model. The subtrees of a *bm* node represent the state machines, fragments (represented as partial state machines), and macros in the corresponding feature module. The representations of state machines and fragments are described below.



(a) B

(b) $F1$ (the path to the extended element is coloured blue)

Figure 3.45: The feature modules and partial FSTs of features B and $F1$ (terminal nodes are shaded in gray)

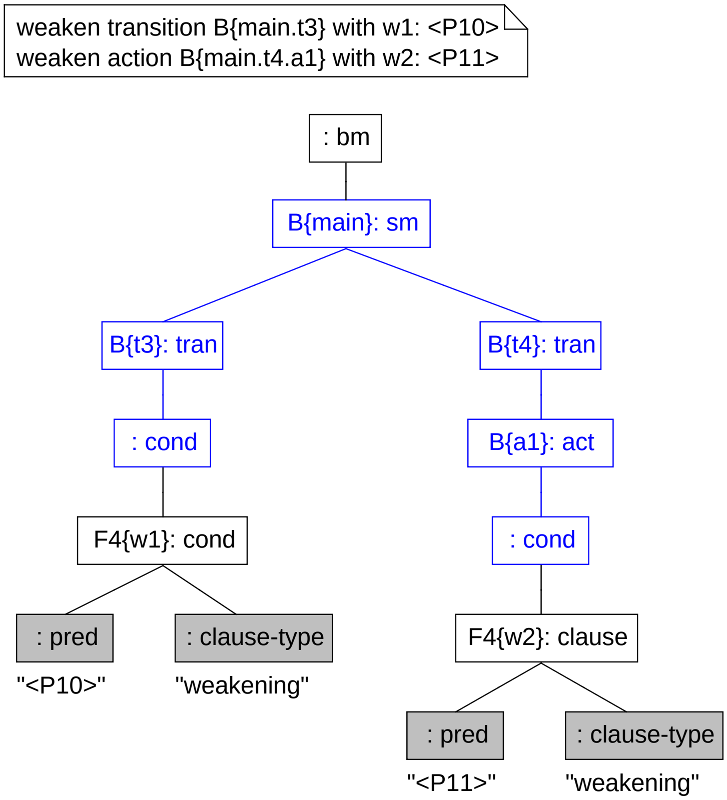


Figure 3.46: The feature module and FST of feature $F4$ (terminal nodes are shaded in gray, and the paths to the extended element is coloured blue)

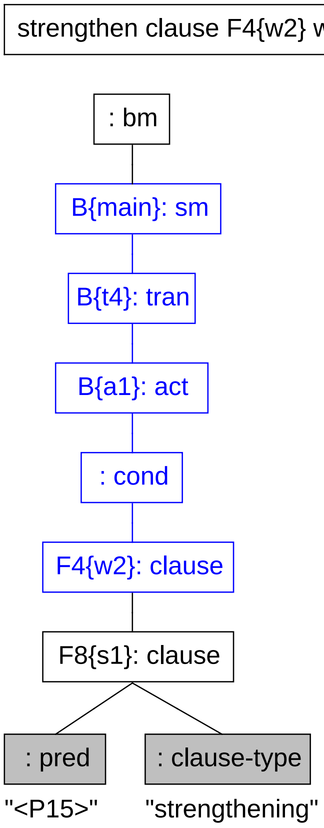


Figure 3.47: The feature module and FST of feature $F8$ (terminal nodes are shaded in gray, and the path to the extended element is coloured blue)

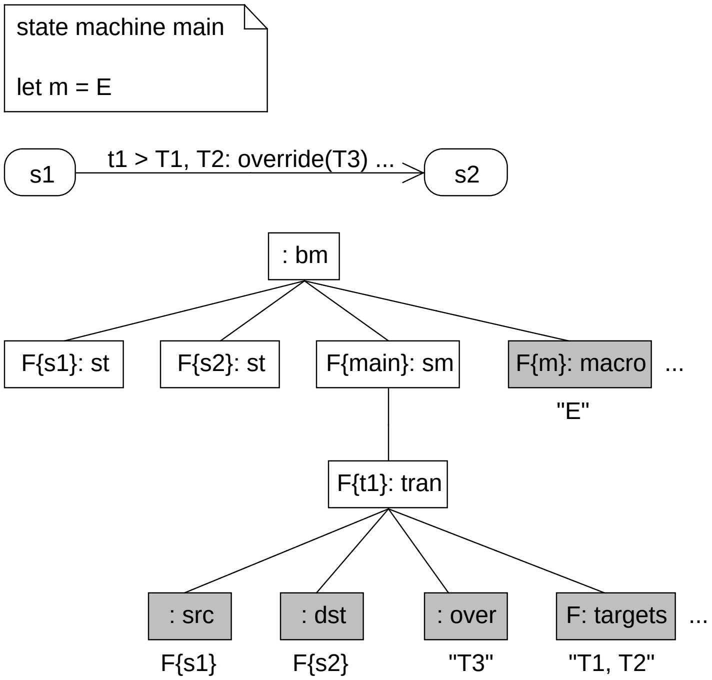


Figure 3.48: The partial state machine and FST of a feature *F*

Representing state machines in FSTs There is a straight-forward correspondence between the nodes in an *sm* subtree and the elements in the corresponding state machine: In Figure 3.45a, each state of B's state machine *main* is represented by a subtree of $B\{main\} : sm$ that is rooted at an *st* node. The descendants of an *st* node represent a state's regions and their sub machines. Similarly, each transition of B's state machine *main* is represented by a subtree of $B\{main\} : sm$ that is rooted at a *tran* node. The subtrees of a *tran* node correspond to a transition's components; for example, the subtrees $:cond$ and $B\{a1\}:act$ of $B\{t3\}:tran$ represents the guard condition and action $a1$ of B's transition $t3$, respectively. The actual predicate that defines a transition or action's guard condition is represented (atomically) by a terminal child node $:pred$ of a $:cond$ node[27]; for example, the predicate *<P1>* of the guard condition of B's transition $t3$ is represented by the $:pred$ child node of the $:cond$ child node of $B\{t3\}:tran$. If a transition or action does not have a guard condition, the FST transition or action has an implicit guard of `true` (e.g., see the transition subtree for $B\{t1\}$).

Representing fragments in FSTs A fragment is represented in the FST of the feature that defines the fragment; the feature being extended is represented by a separate FST. The context of a fragment is mirrored in the fragment feature's FST as the path from the FST root to the root of the fragment. For example, in Figure 3.45b, the context of $F1$'s region $r1$ (i.e., $B\{main.s2\}$) is represented by the nodes $B\{main\} : sm$ and $B\{s2\} : st$ along the path from the FST's root to $r1$'s root $F1\{r\}:reg$.

The representations of weakening and strengthening clauses require special attention, since unlike other fragments (i.e., new states, transitions), weakening and strengthening clauses are not native elements of state machines. A (weakening or strengthening) clause is represented by a *clause* subtree of the *cond* node (*clause* node) of the guard condition (clause) that it extends. A *clause* node has a *pred* child node, and a *clause-type* child node that represents the type of the clause: weakening or strengthening. For example, in Figure 3.46, $F4$'s weakening clause $w2$ is specified by the node $F4\{w2\} : clause$, which has a (weakening) *clause-type* child node. Because $w2$ weakens the guard condition of action $a1$ in B's transition $t4$, the node $F4\{w2\} : clause$ is a child of the *cond* node representing the action's guard condition. As another example, in Figure 3.47, $F8$'s strengthening clause $s1$ is specified by the node $F8\{s1\} : clause$, which has a (strengthening) *clause-type* child node. Because $s1$ strengthens the weakening clause $w2$ of $F4$, $F8\{s1\} : clause$ is a child of the node $F4\{w2\} : clause$.

[27] Since a guard condition's predicate is represented atomically, the FST grammar in Figure 3.44 does not include predicate-logic operations.

3.5.1.2 Merge Operation

Our merge operation is adapted from the generic merge operation defined in the FOSD algebra [6]). The operands of the merge operation are *unordered* FSTs, meaning that the children of each nonterminal node form a set (not a sequence).

Proposition 3.5.1. *The FST of a FORML feature module is unordered.*

Proof: *The children of a nonterminal node represent a set of sub-elements of a composite element:*

- *The children of a bm node represent the union of the set of parallel state machines, set of fragments, and set of macros in a feature module.*
- *The children of an sm node represent the union of the set of top-level states, set of transitions, and the initial-state designation in a state machine.*
- *The children of a reg node represent the union of the set of child states and the initial-state designation in a region.*
- *The children of a tran node represent the union of a transition's source, destination, trigger or (in the case of an override transition) override specification, guard condition, set of actions, and (in the case of a priority transition) set of target transitions.*
- *The children of an act node represent the union of an action's guard condition, WCA, (in the case of an override action) override specification, and (in the case of a priority action) set of target actions.*
- *The children of a cond or clause node represent the union of a (guard condition's or clause's) predicate, and the set of weakening and strengthening clauses that extend the guard condition or clause, respectively. The components of an extended guard condition or clause form a set because the rule for combining them is insensitive to their order[28].*

Note that although some feature-module elements are ordered in the feature module's graphical representation (e.g., the trigger, guard, and actions of a transition), this order is arbitrary and does reflect a semantic order among the elements. Also, although preemptive

[28] Recall that extending a guard condition or clause with a set of weakening and strengthening clauses results in the conjunction of the strengthening clauses, with an additional conjunct that is the disjunction of weakening clauses and the original guard condition or clause.

```
1: procedure MERGE(Node $n_1$, Node $n_2$)
2:     $result \leftarrow$ new Node(name = $n_1$.name, type = $n_1$.type, children = $\emptyset$)
3:     $Mergable \leftarrow \{\ (c_1, c_2) \in n_1.\text{Children} \times n_2.\text{Children}\ |$
                 $c_1.\text{name} = c_2.\text{name}\ \wedge\ c_1.\text{type} = c_2.\text{type}\ \}$
4:     $result.\text{Children} \leftarrow result.\text{Children}\ \cup\ \{\ \text{MERGE}(c_1, c_2)\ |\ (c_1, c_2) \in Mergable\ \}$
5:     $Unmergable \leftarrow \{\ c_1 \in n_1.\text{Children}\ |\ \nexists c_2 \in n_2.\text{Children} : (c_1, c_2) \in Mergable\ \}\ \cup$
                 $\{\ c_2 \in n_2.\text{Children}\ |\ \nexists c_1 \in n_1.\text{Children} : (c_1, c_2) \in Mergable\ \}$
6:     $result.\text{Children} \leftarrow result.\text{Children}\ \cup\ Unmergable$
7:     return $result$
8: end procedure
```

Figure 3.49: Merge operation for unordered FSTs

transitions (actions) impose a priority order on transitions (actions), this priority ordering is specified in the transitions' (actions') labels and not by the ordering of the corresponding tran (act) FST nodes. □

The merge operation is shown in Figure 3.49. An unordered FST is represented by a linked structure of *Node* objects. *Node* is a recursive structure that represents an FST node including its *name* (*name*'s value is *null* if the element is unnamed), FST node *type*, *Children* nodes, and the *content* of terminal nodes. The merge operation superimposes its operand FSTs (arguments n_1 and n_2 of the MERGE operation in Figure 3.49) by recursively merging their common nodes, starting with their root nodes. Two nodes are merged if they have the same name and type. The root nodes of the operand FSTs are always merged because they are of the same type (*bm*) and are both unnamed. When two nodes are merged, so are their child nodes, where possible (lines 2 to 4 in Figure 3.49). The merged and unmerged child nodes (including terminal child nodes) are all added as children of the merged parent node (lines 3-7).

For example, Figure 3.50 shows the partial result of merging feature B's FST (see Figure 3.45a) and feature $F1$'s FST (see Figure 3.45b). As another example, Figure 3.51 shows the partial result of merging feature $F4$'s FST (see Figure 3.46) and feature $F8$'s FST (see Figure 3.47).

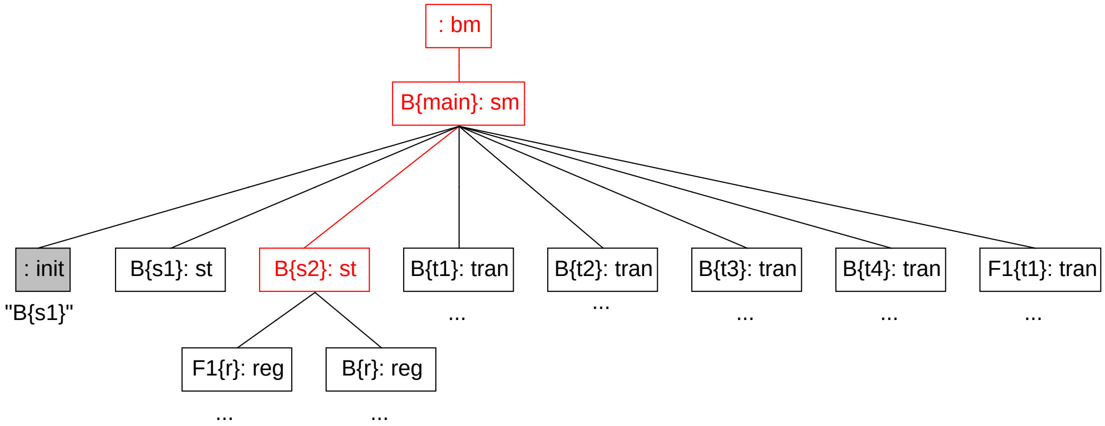


Figure 3.50: Partial result of merging feature B's FST (see Figure 3.45a) and feature $F1$'s FST (see Figure 3.45b): the merged nodes are coloured red.

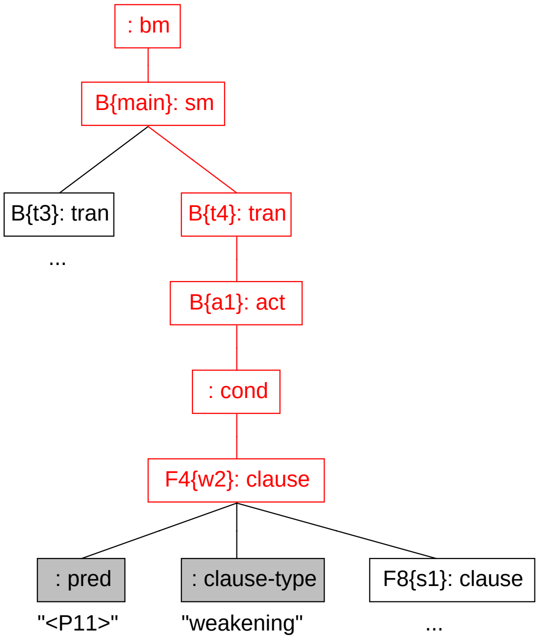


Figure 3.51: Partial result of merging feature $F4$'s FST (see Figure 3.46) and feature $F8$'s FST (see Figure 3.47): the merged nodes are coloured red.

The merge operation described above simplifies the generic merge operation defined in the FOSD algebra by excluding the case of merging common terminal nodes.

Proposition 3.5.2. *Well-formed FORML feature modules do not have common terminal child nodes of common nonterminal nodes.*

Proof:

- *macro child nodes of the bm node in different feature modules have different qualified names.*
- *To avoid common target nodes in different feature modules specifying targets for the same priority transition or action, each feature's targets nodes are named after the feature*[29]*. A feature has at most one targets node per transition, which specifies either the (priority) transition's original targets or the targets added to the transition by a retrospective intended interaction*[30]*. Hence, the targets child nodes of a tran node have different names in different feature modules.*
- *The init child node of an sm or region node is present only in the feature module that introduces the corresponding state machine or region. (A fragment cannot add an initial state to an existing state machine or region.)*
- *The src, dst, trig, override-spec, and priority-list child nodes of a tran node are present only in the feature module that introduces the corresponding transition. (A fragment cannot add a source state, destination state, trigger, or override specification to an existing transition.)*
- *The override-spec, priority-list, and WCA child nodes of an act node are present only in the feature module that introduces the corresponding action. (A fragment cannot add an override specification, target action, or WCAs to an existing action.)*
- *The pred and clause-type child nodes of a cond or clause node are present only in the feature module that introduces the corresponding guard condition or clause. (A fragment can add weakening and strengthening clauses only to an existing guard condition or clause (which corresponds to adding cond child nodes to a cond node), and cannot change the original predicate or clause type of the extended guard condition or clause.)*

[29] For example, in Figure 3.48, the set of target transitions $\{T1, T2\}$ of feature F's priority transition $t1$ is represented in F's FST by the child $F\!:\!\mathit{targets}$ of the node $F\{t1\}\!:\!\mathit{tran}$.

[30] If a feature has multiple fragments that add targets to the same transition, the feature's *targets* node specifies the union of the fragments' added targets.

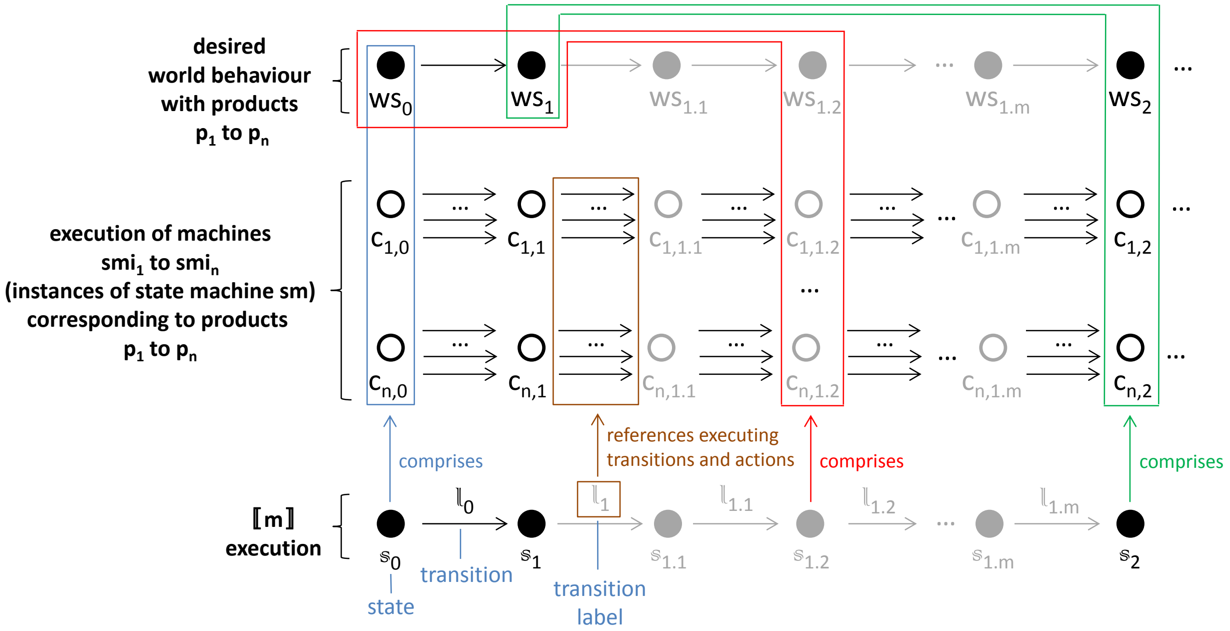


Figure 3.52: An execution of $[\![m]\!]$

□

Proposition 3.5.3. *The merge operation in Figure 3.49 is commutative and associative.*

Proof: *The merge operation essentially takes the union of the merged and unmerged children of the argument nodes. Set union is associative and commutative, and hence, so is the merge operation.* □

Proposition 3.5.3, in conjunction with Proposition 3.5.1 above, imply that the superimposition operator is commutative and associative. Assuming syntactically correct feature modules, the result of superimposing all of the feature modules of an SPL is an FST representing a set of parallel state machines; that is, the FST does not include any fragments.

3.6 Simple Behaviour Model Semantics

Let m be a FORML model of an SPL's requirements, comprising a world model wm and a composed behaviour model (CBM) *cbm* obtained from composing all of the feature modules into a model of the whole SPL. The semantics of *cbm* is the set $\boldsymbol{REQ} \subseteq \boldsymbol{WB}$ of *desired* world behaviours described by *cbm*, where $\boldsymbol{WB}$ is the set of possible *valid* world behaviours described by *wm* (see Definition 3.2.9). $\boldsymbol{REQ}$ is formally defined in terms of the executions of a state-transition system $[\![m]\!]$ derived from *cbm*. $[\![m]\!]$ is described in detail in Sections 3.6.1 to 3.6.4.

As an overview, $[\![m]\!]$ describes the set of desired world behaviours for every possible product configuration. Figure 3.52 shows that an execution of $[\![m]\!]$ represents the concurrent execution of a set of machines (instances of the state machines in *cbm*) for a particular product configuration, and the effects of the machines' transitions on the world behaviour. The particular product configuration is determined by the set of products, each with a particular feature configuration, in m's world model. Thus, Figure 3.52 shows an execution of $[\![m]\!]$ involving n instances of the same state machine *sm*, corresponding to n products. In general, $[\![m]\!]$ comprises multiple (possibly diverse) products, each modelled as multiple parallel machines. Each transition of $[\![m]\!]$ represents a small step in the machines' execution, and is labelled with references to the transitions and actions of the machines that execute in the small step. Each state of $[\![m]\!]$ encapsulates the information needed to compute the next small step: that is, the current configurations of the machines (e.g., in Figure 3.52, $\mathbbm{s}_0$ includes $\boldsymbol{c_{1,0}}$ to $\boldsymbol{c_{n,0}}$), the current observable world state (e.g., $\mathbbm{s}_{1.2}$ includes $\boldsymbol{ws_1}$), and two other world states that are explained below.

3.6.1 Semantic State-Transition System

We now formally define the semantic state-transition system $[\![m]\!]$ derived from a CBM *cbm*.

Definition 3.6.1. *$[\![m]\!]$ is a tuple* $(\mathbb{S}, \mathbb{S}_0, \mathbb{L}, \mathbb{T})$ *where*

- $\mathbb{S}$ *is a set of states.*
- $\mathbb{S}_0 \subseteq \mathbb{S}$ *is a set of initial states.*
- $\mathbb{L} \subseteq 2^{\boldsymbol{T_{univ}}} \times 2^{\boldsymbol{A_{univ}}}$ *is a set of labels, where* $\boldsymbol{T_{univ}}$ *and* $\boldsymbol{A_{univ}}$ *are the unions of the sets of transitions and actions, respectively, of all instances of the state machines in cbm. A label* $\mathbbm{l} \in \mathbb{L}$ *is a tuple of the form* $(\boldsymbol{T}, \boldsymbol{A})$*, where* $\boldsymbol{T}$ *and* $\boldsymbol{A}$ *are references to the transitions and actions of a set of executing machines, respectively, and each action in* $\boldsymbol{A}$ *belongs to some transition in* $\boldsymbol{T}$*.*
- $\mathbb{T} \subseteq \mathbb{S} \times \mathbb{L} \times \mathbb{S}$ *is a transition relation. A member* $(\mathbbm{s}_{src}, \mathbbm{l}, \mathbbm{s}_{dst})$ *of* $\mathbb{T}$ *is a transition from state* $\mathbbm{s}_{src}$ *to state* $\mathbbm{s}_{dst}$ *that is labelled* $\mathbbm{l}$*, and can be graphically shown as* $\mathbbm{s}_{src} \xrightarrow{\mathbbm{l}} \mathbbm{s}_{dst}$*.*

Definition 3.6.2. *Let $\llbracket m \rrbracket$ be a semantic state machine derived from a CBM cbm. An execution of $\llbracket m \rrbracket$ is an infinite sequence of transitions of $\llbracket m \rrbracket$ of the form*

$$(\mathbb{s}_0, \mathbb{l}_0, \mathbb{s}_1), (\mathbb{s}_1, \mathbb{l}_1, \mathbb{s}_2), (\mathbb{s}_2, \mathbb{l}_2, \mathbb{s}_3), \ldots$$

where the source state $\mathbb{s}_0$ of the first transition is an initial state of $\llbracket m \rrbracket$, and the source state of every other transition is the destination state of the previous transition. The execution above can be graphically shown as

$$\mathbb{s}_0 \xrightarrow{\mathbb{l}_0} \mathbb{s}_1 \xrightarrow{\mathbb{l}_1} \mathbb{s}_2 \xrightarrow{\mathbb{l}_2} \mathbb{s}_3 \ldots$$

The elements of $\llbracket m \rrbracket$ are described in detail below.

NOTE: To avoid confusing the states and transitions of $\llbracket m \rrbracket$ with those of its *cbm*, in the following, the unqualified terms "state" and "transition" are used refer to the states and transitions of the *cbm*, respectively; and the terms are qualified when referring to the states and transitions of $\llbracket m \rrbracket$ (e.g., states of $\llbracket m \rrbracket$, an $\llbracket m \rrbracket$ transition).

3.6.2 Semantic States

Definition 3.6.3. *In a given execution of $\llbracket m \rrbracket$, a state $\mathbb{s}$ of $\llbracket m \rrbracket$ is a tuple*

$$\mathbb{s} = (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$$

where

- ***mode** represents the set of state-machine instances (machines) that are executing and their current configurations. More precisely, **mode** assigns values to the following sets and functions:*
    - *For each state machine sm in cmb, $\boldsymbol{I_{sm}}$ is the set of instances of sm in **mode**.*
    - *Let $\boldsymbol{P}$ be the static set of product objects in the execution[31]. For each state machine sm in cbm, a bijective function*

      $$\boldsymbol{sm.product} : \boldsymbol{I_{sm}} \mapsto \boldsymbol{P}$$

      *states the product object in $\boldsymbol{P}$ that corresponds to each sm machine in **mode**.*

[31] The set of products in the initial world state remains static throughout an execution.

- *For each state machine sm in cbm, the function*

$$\boldsymbol{sm.config} : \boldsymbol{I_{sm}} \mapsto Config_{sm}$$

states the configuration of each sm machine in **mode**, *where* $Config_{sm}$ *denotes sm's set of possible configurations (Definition 3.4.6).* $\mathbb{s}$ *is unstable if any machine in* **mode** *is in an unstable configuration and is stable otherwise.*

- $\boldsymbol{ws_c} \in \boldsymbol{WS}$ *represents the current observable world state, where* $\boldsymbol{WS}$ *is the set of valid world states specified by the world model (see Definition 3.2.7).*

- $\boldsymbol{ws_p} \in \boldsymbol{WS}$ *represents the previous observable world state (except for initial states of* $[\![m]\!]$*, as described below).*

- *If* $\mathbb{s}$ *is unstable,* $\boldsymbol{ws_{ci}} \in \boldsymbol{WS}$ *represents the current intermediate world state.*

In the remainder of this thesis, $\boldsymbol{mode}$'s value assignments to the above sets and functions are referenced using the prefix $\boldsymbol{mode::}$ (e.g., $\boldsymbol{mode::sm.product}$).

For example, in Figure 3.52, the unstable $[\![m]\!]$ state $\mathbb{s}_{1.2}$ is defined by the following elements:

$$\begin{aligned}
\boldsymbol{mode::I_{sm}} &= \{\boldsymbol{smi_1}, \ldots, \boldsymbol{smi_n}\}\\
\boldsymbol{mode::sm.product} &= \{(\boldsymbol{smi_1}, \boldsymbol{p_1}), \ldots, (\boldsymbol{smi_n}, \boldsymbol{p_n})\}\\
\boldsymbol{mode::sm.config} &= \{(\boldsymbol{smi_1}, \boldsymbol{c_{1,1.2}}), \ldots, (\boldsymbol{smi_n}, \boldsymbol{c_{n,1.2}})\}\\
\boldsymbol{ws_c} &= \boldsymbol{ws_1}\\
\boldsymbol{ws_p} &= \boldsymbol{ws_0}\\
\boldsymbol{ws_{ci}} &= \boldsymbol{ws_{1.2}}
\end{aligned}$$

and the stable $[\![m]\!]$ state $\mathbb{s}_2$ is defined by the following elements:

$$\begin{aligned}
\boldsymbol{mode::I_{sm}} &= \{\boldsymbol{smi_1}, \ldots, \boldsymbol{smi_n}\}\\
\boldsymbol{mode::sm.product} &= \{(\boldsymbol{smi_1}, \boldsymbol{p_1}), \ldots, (\boldsymbol{smi_n}, \boldsymbol{p_n})\}\\
\boldsymbol{mode::sm.config} &= \{(\boldsymbol{smi_1}, \boldsymbol{c_{1,2}}), \ldots, (\boldsymbol{smi_n}, \boldsymbol{c_{n,2}})\}\\
\boldsymbol{ws_c} &= \boldsymbol{ws_2}\\
\boldsymbol{ws_p} &= \boldsymbol{ws_1}\\
\boldsymbol{ws_{ci}} &= \boldsymbol{ws_2}
\end{aligned}$$

In a stable $[\![m]\!]$ state, it is always the case that $\boldsymbol{ws_{ci}} = \boldsymbol{ws_c}$.

Context	**Scope**
Selection expressions: v in $<O>$ [v | $<P>$]	$<P>$
Quantified predicates: e.g., v in no v: $<S>$ | $<P>$	$<S>$ and $<P>$
Trigger expressions: e.g., o in C+(o)	The whole transition, and the whole of override transitions that target the transition
WCAs that add objects: e.g., var in var = +E(list(a = $<val>$))	Other WCAs in the same transition

Table 3.22: The context and scope of object variables in the behaviour-model

3.6.2.1 Initial Semantic States

Definition 3.6.4. *For each valid world state* $\boldsymbol{ws_0} \in \boldsymbol{WS}$*,* $[\![m]\!]$ *has exactly one initial state* $\$_0 \in \mathbf{\$}_0$*:* $\$_0 :: \boldsymbol{ws_c} = \$_0 :: \boldsymbol{ws_p} = \$_0 :: \boldsymbol{ws_{ci}} = \boldsymbol{ws_0}$*. All machines in* $\$_0 :: \boldsymbol{mode}$ *are in their initial configurations.*

$[\![m]\!]$ has a unique initial state $\$_0$ for each world state $\boldsymbol{ws_0}$ for the following reasons: The current world state $\$_0 :: \boldsymbol{ws_c}$ is defined to be $\boldsymbol{ws_0}$, the first world state of a world behaviour. Furthermore, $\$_0 :: \boldsymbol{ws_p} = \$_0 :: \boldsymbol{ws_c}$, because there is no previous observable world state at the start of a world behaviour; and $\$_0 :: \boldsymbol{ws_{ci}} = \$_0 :: \boldsymbol{ws_c}$, because the initial configurations of the machines in $\$_0 :: \boldsymbol{mode}$ are stable. Finally, a machine has a unique initial configuration; hence, $\$_0 :: \boldsymbol{mode}$ is unique.

3.6.3 Semantics of Behaviour-Model Expressions

This section presents the formal semantics of expressions that are used in the behaviour model, including trigger expressions, guard conditions, and expressions that appear in actions. We revisit the expression types described in Section 3.3 (e.g., set expressions, predicates) and interpret them with respect to a semantic state of $[\![m]\!]$. We also discuss four expression types that are specific to the behaviour model: *inState* predicates, @*curri* expressions, trigger expressions, and WCAs.

$$
\begin{aligned}
\llbracket \textsf{inState(st)} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \boldsymbol{st} \in \boldsymbol{mode}\,\boldsymbol{::}\,\boldsymbol{sm_{this}.config(this)}\\
\llbracket \textsf{sm.inState(st)} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \boldsymbol{st} \in \boldsymbol{mode}\,\boldsymbol{::}\,\boldsymbol{sm.config(}\ \boldsymbol{mode}\,\boldsymbol{::}\,\boldsymbol{sm.product^{-1}(p_{this})}\ \boldsymbol{)}\\
\llbracket {<}p{>}\textsf{(sm).inState(st)} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \boldsymbol{st} \in \boldsymbol{mode}\,\boldsymbol{::}\,\boldsymbol{sm.config(}\ \boldsymbol{mode}\,\boldsymbol{::}\,\boldsymbol{sm.product^{-1}(}\llbracket {<}p{>} \rrbracket\boldsymbol{(\mathbb{s}, B))}\ \boldsymbol{)}
\end{aligned}
$$

Figure 3.53: The formal semantics of *inState* predicates: $\boldsymbol{this}$ denotes the machine in which the predicate appears, $\boldsymbol{sm_{this}}$ denotes the type of $\boldsymbol{this}$, $\boldsymbol{p_{this} = mode :: sm_{this}.product(this)}$ stands for the product of $\boldsymbol{this}$, and $\boldsymbol{mode :: sm.product^{-1}}$ is the inverse of $\boldsymbol{mode\,::\,sm.product}$.

$$
\begin{aligned}
\llbracket \textsf{Cs@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \boldsymbol{ws_{ci}\,::\,O_C}\\
\llbracket {<}O{>}\textsf{.a@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,C.a(o)} \mid \boldsymbol{o} \in \llbracket {<}O{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}\\
\llbracket {<}O{>}\textsf{.A-r@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{a} \in \boldsymbol{ws_{ci}\,::\,O_A} \mid \exists\ \boldsymbol{o} \in \llbracket {<}O{>} \rrbracket(\mathbb{s}, \boldsymbol{B}) : \boldsymbol{ws_{ci}\,::\,A.r(a) = o}\}\\
\llbracket {<}O{>}\textsf{.A@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \llbracket {<}O{>}\textsf{.A-r} \rrbracket(\mathbb{s}, \boldsymbol{B})^{1}\\
\llbracket {<}L{>}\textsf{.r@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,A.r(a)} \mid \boldsymbol{a} \in \llbracket {<}L{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}\\
\llbracket {<}P{>}\textsf{.F@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,SPL.F(p)} \mid \boldsymbol{p} \in \llbracket {<}P{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}\\
\llbracket {<}M{>}\textsf{.to@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,M.to(m)} \mid \boldsymbol{m} \in \llbracket {<}M{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}\\
\llbracket {<}M{>}\textsf{.from@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,M.from(m)} \mid \boldsymbol{m} \in \llbracket {<}M{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}\\
\llbracket {<}M{>}\textsf{.p@curri} \rrbracket(\mathbb{s}, \boldsymbol{B}) &\equiv \{\boldsymbol{ws_{ci}\,::\,M.p(m)} \mid \boldsymbol{m} \in \llbracket {<}M{>} \rrbracket(\mathbb{s}, \boldsymbol{B})\}
\end{aligned}
$$

[1] This expression is well-formed only if the objects $\llbracket {<}O{>} \rrbracket(\boldsymbol{B})$ participate only in role r of A. Hence, the role r for which the navigation expression $\llbracket {<}O{>}\textsf{.A@curri} \rrbracket$ should be evaluated can be inferred.

Figure 3.54: The formal semantics of @curri expressions: ${<}O{>}$, ${<}L{>}$, ${<}P{>}$, and ${<}M{>}$ are set expressions.

3.6.3.1 General Expressions

This section gives the formal semantics for *general expressions* that appear within guard conditions and actions: there are the expression types in Section 3.3, *inState* predicates, and @*curri* expressions. The formal semantics of a general expression e is given as a mathematical expression $\llbracket e \rrbracket(\mathbb{s},\ \boldsymbol{B})$ that is evaluated with respect to a state $\mathbb{s} = \boldsymbol{(mode, ws_c, ws_p, ws_{ci})}$ of $\llbracket m \rrbracket$ and a set $\boldsymbol{B}$ of value bindings for object variables. The object variables in $\boldsymbol{B}$ are introduced by expressions and have limited scopes. Table 3.22 summarizes the different types and scopes of object variables that can be introduced by behaviour-model expressions. For example, a selection expression (Section 3.3.1) introduces an object variable whose scope is limited to the selection predicate. The formal semantics of set, predicate, integer, and @*pre* expressions in the behaviour model are as described in Section 3.3, with a minor adaptation: In Section 3.3, the state $\mathbb{s}$ is assumed to be a pair of consecutive world states $\boldsymbol{(ws_{i-1}, ws_i)}$, where $\boldsymbol{ws_i}$ and $\boldsymbol{ws_{i-1}}$ are the current and previous world states, respectively. In this section, $\boldsymbol{ws_i}$ and $\boldsymbol{ws_{i-1}}$ are referred to as $\boldsymbol{ws_c}$ and $\boldsymbol{ws_p}$, respectively; furthermore, $\mathbb{s}$ is expanded with the additional elements $\boldsymbol{mode}$ and $\boldsymbol{ws_{ci}}$, which are used to evaluate *inState* predicates and @*curri* expressions, respectively.

The *inState* predicate is used to check whether a machine's configuration includes a particular state *st* (see the semantic definition in Figure 3.53). The three versions of the predicate apply to the configuration of *this* machine (i.e., the machine in which the predicate appears), a different machine in the same product as *this*, and a machine in a different product, respectively.

An @curri expression is used in cases when a transition originating from an unstable state has an expression that should be evaluated with respect to the current intermediate world state (see the semantic definition in Figure 3.54). For example, the semantics of $<O>$.a@curri is the set of values of attribute a in $\boldsymbol{ws_{ci}}$ for the objects $<O>$.

3.6.3.2 Trigger expressions

Recall from Section 3.4.1.2 that a transition's trigger expression *te* is evaluated together with the transition's guard condition *gc*: *te* evaluates to true if its WCE occurs *and* *gc* evaluates to true with respect to the (added, removed, changed) object that raised the WCE. To simplify the presentation of the formal semantics of *te*, *gc* is assumed to be the predicate true if the transition does not have a guard condition. The formal semantics of *te* has two components (the subscripts P and B stand for *predicate* and *binding*, respectively):

- A predicate $[\![te,\, gc]\!]_P(\mathbb{s})$ corresponding to whether *te* together with *gc* evaluates to true in $[\![m]\!]$ state $\mathbb{s}$. The predicate is true if a WCE of the type specified by *te* occurs in the world-state transition from $\boldsymbol{ws_p}$ to $\boldsymbol{ws_c}$, and *gc* evaluates to true with respect to $\mathbb{s}$ and the WCE's object. There may be more than one instance of the WCE that satisfies both *te* and *gc*. For example, suppose that multiple C objects with value v for attribute a are concurrently added in the world-state transition from $\boldsymbol{ws_p}$ to $\boldsymbol{ws_c}$. In this case, $[\![\mathsf{C{+}(o),\ o.a = v}]\!]_P(\mathbb{s})$ evaluates to true for all such objects.

$$
\begin{aligned}
[\![\mathsf{C+(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_P(\mathbb{s}) &\equiv O_{C+}(\mathbb{s}, {<}gc{>}) \neq \emptyset \text{ and} \\
[\![\mathsf{C+(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_B(\mathbb{s}) &\equiv \{\ (o, \mathit{some}\ O_{C+}(\mathbb{s}, {<}gc{>}))\ \} \text{ where} \\
&\quad O_{C+}(\mathbb{s}, {<}gc{>}) = \{\ \boldsymbol{c} \in \boldsymbol{ws_c}\,\boldsymbol{::}\,\boldsymbol{O_C} \mid \boldsymbol{c} \notin \boldsymbol{ws_p}\,\boldsymbol{::}\,\boldsymbol{O_C}\ \wedge\ [\![{<}gc{>}]\!](\mathbb{s}, \{(o,c)\})\ \} \\
[\![\mathsf{C\text{-}(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_P(\mathbb{s}) &\equiv O_{C-}(\mathbb{s}, {<}gc{>}) \neq \emptyset \text{ and} \\
[\![\mathsf{C\text{-}(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_B(\mathbb{s}) &\equiv \{\ (o, \mathit{some}\ O_{C-}(\mathbb{s}, {<}gc{>}))\ \} \text{ where} \\
&\quad O_{C-}(\mathbb{s}, {<}gc{>}) = \{\ \boldsymbol{c} \in \boldsymbol{ws_p}\,\boldsymbol{::}\,\boldsymbol{O_C} \mid \boldsymbol{c} \notin \boldsymbol{ws_c}\,\boldsymbol{::}\,\boldsymbol{O_C}\ \wedge\ [\![{<}gc{>}]\!](\mathbb{s}, \{(o,c)\})\ \} \\
[\![\mathsf{C.a{\sim}(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_P(\mathbb{s}) &\equiv O_{C\sim}(\mathbb{s}, {<}gc{>}) \neq \emptyset \text{ and} \\
[\![\mathsf{C.a{\sim}(o)},\ \mathsf{<}\mathit{gc}\mathsf{>}]\!]_B(\mathbb{s}) &\equiv \{\ (o, \mathit{some}\ O_{C\sim}(\mathbb{s}, {<}gc{>}))\ \} \text{ where} \\
&\quad O_{C\sim}(\mathbb{s}, {<}gc{>}) = \{\ \boldsymbol{c} \in \boldsymbol{ws_c}\,\boldsymbol{::}\,\boldsymbol{O_C}\ \cap\ \boldsymbol{ws_p}\,\boldsymbol{::}\,\boldsymbol{O_C} \mid \\
&\qquad\qquad \boldsymbol{ws_c}\,\boldsymbol{::}\,\boldsymbol{C.a(c)} \neq \boldsymbol{ws_p}\,\boldsymbol{::}\,\boldsymbol{C.a(c)}\ \wedge\ [\![{<}gc{>}]\!](\mathbb{s}, \{(o,c)\})\ \}
\end{aligned}
$$

Figure 3.55: The formal semantics of trigger expressions: *some* S denotes a nondeterministically chosen member of the set S.

- A singleton set $[\![te,\ gc]\!]_B(\mathbb{s})$ that binds te's object variable to a particular object that satisfies the predicate $[\![te,\ gc]\!]_P(\mathbb{s})$. If multiple objects satisfy the predicate, one such object (chosen nondeterministically) is bound to te's object variable.

Figure 3.55 gives the formal semantics for the different types of trigger expressions.

3.6.3.3 WCAs

The formal semantics of a WCA wca has two components:

- A predicate $[\![wca]\!]_P(\mathbb{s}, \boldsymbol{B}, \boldsymbol{ws'})$ that specifies the effect of wca as a (partial[32]) constraint on how the current (observable or intermediate) world state should change, where $\mathbb{s}$ is an $[\![m]\!]$ state, $\boldsymbol{B}$ is a set of value bindings for object variables, and $\boldsymbol{ws'}$ is the next (observable or intermediate) world state

- A set $[\![wca]\!]_B$ that, if wca adds an object to the world and defines an object variable var to refer to the new object, binds variable var to the added object; otherwise, the set is empty.

[32] Recall from Section 3.4 that, in some small steps, the uncontrolled parts of the current world state that are not affected by the small step's actions can change arbitrarily within the constraints of the world model.

$$
\begin{aligned}
&\llbracket \textsf{var} = \ +\textsf{E}(\textsf{list}(\textsf{a} = \ <\!val\!>))\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \\
&\qquad \boldsymbol{new_E} \in \boldsymbol{ws'::O_E} \ \wedge \bigwedge_{list(a=<val>)} \boldsymbol{ws'::E.a(new_E)} = \llbracket <\!val\!> \rrbracket(\$, \boldsymbol{B}) \\
&\llbracket \textsf{var} = \ +\textsf{E}(\textsf{list}(\textsf{a} = \ <\!val\!>))\rrbracket_B \equiv \{\ (var, \boldsymbol{new_E})\ \} \\
&\llbracket \textsf{var} = \ +\textsf{A}(\textsf{list}(\textsf{r} = \ <\!o\!>)\ ,\ \textsf{list}(\textsf{a} = \ <\!val\!>))\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \\
&\qquad \boldsymbol{new_A} \in \boldsymbol{ws'::O_A} \ \wedge \\
&\qquad \bigwedge_{list(r=<o>)} \boldsymbol{ws'::A.r(new_A)} = \llbracket <\!o\!> \rrbracket(\$, \boldsymbol{B}) \ \wedge \bigwedge_{list(a=<val>)} \boldsymbol{ws'::A.a(new_A)} = \llbracket <\!val\!> \rrbracket(\$, \boldsymbol{B}) \\
&\llbracket \textsf{var} = \ +\textsf{A}(\textsf{list}(\textsf{r} = \ <\!o\!>)\ ,\ \textsf{list}(\textsf{a} = \ <\!val\!>))\rrbracket_B \equiv \{\ (var, \boldsymbol{new_A})\ \} \\
&\llbracket +\textsf{MO}(\textsf{from} = <\!p\!>\ ,\ \textsf{list}(\textsf{pr} = \ <\!e\!>))\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \\
&\qquad \boldsymbol{new_{MO}} \in \boldsymbol{ws'::O_{MO}} \ \wedge \ \boldsymbol{ws'::MO.from(new_{MO})} = \llbracket <\!p\!> \rrbracket(\$, \boldsymbol{B}) \ \wedge \\
&\qquad \bigwedge_{list(pr=<e>)} \boldsymbol{ws'::MO.pr(new_{MO})} = \llbracket <\!e\!> \rrbracket(\$, \boldsymbol{B}) \\
&\llbracket +\textsf{MIO}(\textsf{to} = <\!p1\!>,\ \textsf{from} = <\!p2\!>\ ,\ \textsf{list}(\textsf{pr} = \ <\!e\!>))\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \\
&\qquad \boldsymbol{new_{MIO}} \in \boldsymbol{ws'::O_{MIO}} \ \wedge \\
&\qquad \boldsymbol{ws'::MIO.to(new_{MIO})} = \llbracket <\!p1\!> \rrbracket(\$, \boldsymbol{B}) \ \wedge \ \boldsymbol{ws'::MIO.from(new_{MIO})} = \llbracket <\!p2\!> \rrbracket(\$, \boldsymbol{B}) \ \wedge \\
&\qquad \bigwedge_{list(pr=<e>)} \boldsymbol{ws'::MIO.pr(new_{MIO})} = \llbracket <\!e\!> \rrbracket(\$, \boldsymbol{B}) \\
&\llbracket \textsf{-C}(<\!O\!>)\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \llbracket <\!O\!> \rrbracket(\$, \boldsymbol{B}) \ \cap \ \boldsymbol{ws'::O_C} = \emptyset \\
&\llbracket <\!o\!>\textsf{.a} := \ <\!v\!> \rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'}) \equiv \llbracket <\!o\!> \rrbracket(\$, \boldsymbol{B}) \in \boldsymbol{ws'::O_C} \ \Rightarrow \ \boldsymbol{ws'::C.a}(<\!o\!>) = \llbracket <\!v\!> \rrbracket(\$, \boldsymbol{B}) \\
&\qquad \text{where } \llbracket <\!o\!> \rrbracket(\$, \boldsymbol{B}) \text{ evaluates to a } C \text{ object.}
\end{aligned}
$$

Figure 3.56: The formal semantics of WCA expressions: E is an entity, A is an association, MO is an output message, MIO is an IO message, and C is a general concept. $\boldsymbol{new_C}$ denotes a new object of type C that has never appeared in past world states in the current world behaviour.

Figure 3.56 gives the above components for the different types of WCAs. For example, the semantics of the WCA $\textsf{var} = \ +\textsf{E}(\textsf{list}(\textsf{a} = \ <\!val\!>))$ has a predicate

$$\llbracket \textsf{var} = \ +\textsf{E}(\textsf{list}(\textsf{a} = \ <\!val\!>))\rrbracket_P(\$, \boldsymbol{B}, \boldsymbol{ws'})$$

which specifies that a new E object $\boldsymbol{new_E}$ should appear in the next world state ws'

$$\boldsymbol{new_E} \in \boldsymbol{ws'::O_E}$$

and that each attribute a of the new object should be initialized to a corresponding value $<\!val\!>$

$$\bigwedge_{list(a=<val>)} \boldsymbol{ws' :: E.a(new_E)} = [\![<\!val\!>]\!](\mathbb{s}, \boldsymbol{B})$$

The set

$$[\![\mathsf{var} = \ +\mathsf{E}(\mathrm{list}(\mathsf{a} = \ <\!val\!>))]\!]_B$$

binds the new object to the object variable *var*. The variable *var* can then be referenced in other WCAs of the same transition, for example, to link the new object to another object.

3.6.4 Semantic Transitions

An $[\![m]\!]$ transition represents a small step in the concurrent execution of a set of machines.

Definition 3.6.5. *A transition of* $[\![m]\!]$ *is a tuple* $(\mathbb{s}, \mathbb{l}, \mathbb{s}')$ *where*

- $\mathbb{s} = (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$ *is the source state of the* $[\![m]\!]$ *transition.*
- $\mathbb{l} = (\boldsymbol{T}, \boldsymbol{A})$ *is the label of the* $[\![m]\!]$ *transition, which references the transitions* $\boldsymbol{T}$ *and actions* $\boldsymbol{A}$ *of the machines in* $\boldsymbol{mode}$ *that are enabled with respect to* $\mathbb{s}$ *and are not preempted by other enabled transitions and actions. The execution of these transitions and actions constitutes a small step in the machines' execution.*
- $\mathbb{s}' = (\boldsymbol{mode'}, \boldsymbol{ws'_c}, \boldsymbol{ws'_p}, \boldsymbol{ws'_{ci}})$ *is the destination state of the* $[\![m]\!]$ *transition, which results from executing the transitions and actions in* $\mathbb{l}$*, starting from* $\mathbb{s}$*.*

Caveat: For simplicity of presentation, the above definition assumes that the CBM is *deterministic*. In order for the CBM to be deterministic, it must not be possible for two or more unpreempted nonconcurrent transitions (Definition 3.4.7) to be enabled in the same $[\![m]\!]$ state (i.e., $\boldsymbol{T}$ is unique in $\mathbb{s}$). We relax the assumption of determinism in Section 4.1.

For example, in Figure 3.52, the $[\![m]\!]$ transition $(\mathbb{s}_1, \mathbb{l}_1, \mathbb{s}_{1.1})$ represents a small step in the execution of the machines $\boldsymbol{smi_1} \ldots \boldsymbol{smi_n}$: the transitions and actions in $\mathbb{l}_1$ update the machines' configurations from $\boldsymbol{c_{1,1}} \ldots \boldsymbol{c_{n,1}}$ in $\mathbb{s}_1$ to $\boldsymbol{c_{1,1.1}} \ldots \boldsymbol{c_{n,1.1}}$ in $\mathbb{s}_{1.1}$; and update the current observable, previous observable, and current intermediate world states from $\boldsymbol{ws_1}$, $\boldsymbol{ws_0}$, and $\boldsymbol{ws_1}$ in $\boldsymbol{s_1}$ to $\boldsymbol{ws_1}$, $\boldsymbol{ws_0}$, and $\boldsymbol{ws_{1.1}}$ in $\mathbb{s}_{1.1}$, respectively.

The following describes how the label of an $[\![m]\!]$ transition is computed from its source state, and how its destination state is computed from its label and source state.

3.6.4.1 Computing the Label

Let $\mathbb{s} = (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$ be the source state of an $[\![m]\!]$ transition. The label $\mathbb{l}$ of the $[\![m]\!]$ transition is the tuple $(T_{exe}(\mathbb{s}), A_{exe}(\mathbb{s}))$, where $T_{exe}(\mathbb{s})$ and $A_{exe}(\mathbb{s})$ – described in detail below – are the sets of enabled transitions and actions, respectively, of the machines in $\boldsymbol{mode}$ that are not preempted by other enabled transitions and actions.

Computing $T_{exe}(\mathbb{s})$: Let $T(\mathbb{s})$ be the set of transitions of the machines in $\boldsymbol{mode}$. Let $T_{en}(\mathbb{s}) \subseteq T(\mathbb{s})$ be the subset of transitions in $T(\mathbb{s})$ that are enabled with respect to $\mathbb{s}$. $T_{en}(\mathbb{s})$ is computed in four steps: (1) The set of enabled non-override and non-else transitions is computed; in the following, such transitions are called *independent transitions*, since their enabling conditions do not depend on the enabledness of other transitions. (2) The set of enabled override transitions is computed based on the set of enabled independent transitions. (3) The set of enabled else transitions is computed based on the sets of enabled independent and override transitions. (4) Finally, $T_{en}(\mathbb{s})$ is computed as the union of the sets of enabled independent, override, and else transitions.

(1) Recall from Section 3.4 that an independent transition is enabled when its source state is in the corresponding machine's configuration, and its trigger expression (if any) and guard condition (if any) are satisfied. Also, recall from Section 3.4.3.1 that as soon as a machine reaches a stable configuration, it does not execute any more transitions in the current big step. More precisely, except at the start of a big step where $\mathbb{s}$ is stable, a transition with a stable source state cannot be enabled. Hence, the set $T_{en-ind}(\mathbb{s})$ of enabled independent transitions can be defined more precisely as follows:

Definition 3.6.6.

$$\begin{aligned} T_{en-ind}(\mathbb{s}) = \{\ \boldsymbol{t} \in T_{ind}(\mathbb{s}) \mid\ & \boldsymbol{src(t)} \in \boldsymbol{mode}::\boldsymbol{sm(t).config(smi(t))}\ \wedge \\ & [\![\boldsymbol{te(t)},\ \boldsymbol{gc(t)}]\!]_P(\mathbb{s})\ \wedge \\ & (stable(\mathbb{s})\ \vee\ unstable(\boldsymbol{src(t)}))\ \} \end{aligned}$$

where $T_{ind}(\mathbb{s}) \subseteq T(\mathbb{s})$ is the subset of independent transitions in $T(\mathbb{s})$; for a machine transition $\boldsymbol{t}$, $\boldsymbol{smi(t)}$, $\boldsymbol{sm(t)}$, $\boldsymbol{src(t)}$, $\boldsymbol{dst(t)}$, $\boldsymbol{te(t)}$, and $\boldsymbol{gc(t)}$, represent $\boldsymbol{t}$'s machine, machine type (i.e., the state machine in the CBM of which $\boldsymbol{t}$'s machine is an instance), source state, destination state, triggering event, and guard condition; $stable(\mathbb{s})$ means that $\mathbb{s}$ is a stable semantic state; and $unstable(\boldsymbol{s})$ means that $\boldsymbol{s}$ is an unstable machine state.

If $\boldsymbol{gc(t)}$ does not exist, it is assumed to be true. *If $\boldsymbol{te(t)}$ does not exist,*

$$\begin{aligned} T_{en-ind}(\$) = \{\ \boldsymbol{t} \in T_{ind}(\$) \mid\ & \boldsymbol{src(t)} \in \boldsymbol{mode}::\boldsymbol{sm(t).config(smi(t))}\ \wedge \\ & [\![\boldsymbol{gc(t)}]\!](\$, \emptyset)\ \wedge \\ & (stable(\$)\ \vee\ unstable(\boldsymbol{src(t)}))\ \} \end{aligned}$$

(2) Recall from Section 3.4, that an override transition is enabled when its source state is in the corresponding machine's configuration, its guard condition (if any) is satisfied, and its target transition is enabled. Also, recall from Section 3.4.3.1 that, except at the start of a big step where $\$$ is stable, a transition with a stable source state cannot be enabled. Hence, the set $T_{en-ov}(\$)$ of enabled override transitions can be defined more precisely as the set of enabled transitions that override enabled independent transitions, the enabled transitions that override those transitions, and so on. To do so, we first define the set of enabled override transitions that override a specific set of enabled transitions T_{targ}:

Definition 3.6.7. *Let $T_{ov}(\$) \subseteq T(\$)$ be the subset of override transitions in $T(\$)$ and let T_{targ} be a set of enabled transitions in $T(\$)$.*

$$\begin{aligned} T_{en-ov}(\$, T_{targ}) = \{\ \boldsymbol{t} \in T_{ov}(\$) \mid\ & \boldsymbol{src(t)} \in \boldsymbol{mode}::\boldsymbol{sm(t).config(smi(t))}\ \wedge \\ & [\![\boldsymbol{gc(t)}]\!](\$, \boldsymbol{B_{targ}})\ \wedge \\ & (stable(\$)\ \vee\ unstable(\boldsymbol{src(t)}))\ \wedge \\ & \exists\ \boldsymbol{t_{targ}} \in T_{targ} : \boldsymbol{t}\ overrides\ \boldsymbol{t_{targ}}\ \} \end{aligned}$$

where $\boldsymbol{B_{targ}}$ is the set of object-variable bindings introduced by $\boldsymbol{t}$'s target transition $\boldsymbol{t_{targ}}$; that is,

$$\boldsymbol{B_{targ}} = [\![\boldsymbol{te(t_{targ})},\ \boldsymbol{gc(t_{targ})}]\!]_B(\$)$$

if $\boldsymbol{te(t_{targ})}$ exists (if $\boldsymbol{gc(t_{targ})}$ does not exist, it is assumed to be true*); otherwise, $\boldsymbol{B_{targ}}$ is the empty set.*

$T_{ov}(\$)$ can now be defined as follows:

Definition 3.6.8. *$T_{en-ov}(\$)$ is the least fixpoint of the function*

$$\lambda\ \$, T_{targ} : T_{en-ov}(\$, T_{en-ind}(\$))\ \cup\ T_{en-ov}(\$, T_{targ})$$

That is, $T_{en-ov}(\$)$ is the limit of the sequence $T^0_{targ}, T^1_{targ}, T^2_{targ}, \ldots$ where

$$\begin{aligned} T^0_{targ} &= T_{en-ov}(\$, T_{en-ind}(\$)),\ and \\ T^{i+1}_{targ} &= T^i_{targ}\ \cup\ T_{en-ov}(\$, T^i_{targ}) \end{aligned}$$

(3) Recall from Section 3.4 that an else transition is enabled when its source state is in the corresponding machine's configuration, and no non-else transitions from its source state are enabled. Hence, the set $T_{en-els}(\$)$ of enabled else transitions can be defined more precisely as follows:

Definition 3.6.9.

$$\begin{aligned} T_{en-els}(\$) = \{\ \boldsymbol{t} \in T_{else}(\$) \mid \boldsymbol{src(t)} \in \boldsymbol{mode::sm(t).config(smi(t))}\ \wedge \\ \nexists\ \boldsymbol{t_{nel}} \in T_{en-ind}(\$)\ \cup\ T_{en-ov}(\$) : \boldsymbol{src(t_{nel})} = \boldsymbol{src(t)}\ \} \end{aligned}$$

where $T_{else}(\$) \subseteq T(\$)$ is the subset of else transitions in $T(\$)$[33].

(4) Finally, $T_{en}(\$)$ can be defined as follows:

Definition 3.6.10.

$$T_{en}(\$) = T_{en-ind}(\$)\ \cup\ T_{en-ov}(\$)\ \cup\ T_{en-els}(\$)$$

Given the above definition for $T_{en}(\$)$, $T_{exe}(\$)$ is the subset of transitions in $T_{en}(\$)$ that are not preempted by other transitions in $T_{en}(\$)$[34]:

Definition 3.6.11.

$$T_{exe}(\$) = \{\ \boldsymbol{t_1} \in T_{en}(\$) \mid \nexists\ \boldsymbol{t_2} \in T_{en}(\$) : \boldsymbol{(t_2, t_1)} \in Preempts_{tran}(\$)\ \}$$

where $Preempts_{tran}(\$)$ (defined below) is the relation between preemptive transitions and their target transitions.

Definition 3.6.12.

$$Preempts_{tran}(\$) = \{\ \boldsymbol{(t_{pre}, t)} \in T(\$) \times T(\$) \mid \boldsymbol{t}\ \textit{is a target transition of}\ \boldsymbol{t_{pre}}\ \}$$

[33] Note that preempting the enabled non-else transitions from an unstable state does not enable the state's else transition. This semantic decision helps to ensure that preemptive transitions are side-effect free (so as to avoid unintended side effects).

[34] Note that a preemptive transition t's target is preempted, even if t is itself preempted. This helps to ensure that preemptive transitions are side-effect free.

Computing $A_{exe}(\mathbb{s})$**:** Let $A(\mathbb{s})$ be the set of actions of the machines in ***mode***, and $A(T_{exe}(\mathbb{s})) \subseteq A(\mathbb{s})$ be the set of actions of the transitions in $T_{exe}(\mathbb{s})$. Let $A_{en}(\mathbb{s}) \subseteq A(T_{exe}(\mathbb{s}))$ be the subset of actions in $A(T_{exe}(\mathbb{s}))$ that are enabled with respect to $\mathbb{s}$. $A_{en}(\mathbb{s})$ is computed in three steps: (1) The set of enabled non-override actions is computed. (2) The set of enabled override actions is computed based on the set of enabled non-override actions. (3) Finally, $A_{en}(\mathbb{s})$ is computed as the union of the sets of enabled non-override and override actions.

(1) Recall from Section 3.4 that a non-override action is enabled if its guard condition (if any) is satisfied. Hence, the set $A_{en-nov}(\mathbb{s})$ of enabled non-override actions can be defined more precisely as follows:

Definition 3.6.13.

$$A_{en-nov}(\mathbb{s}) = \{\ \boldsymbol{a} \in A_{nov}(\mathbb{s}) \mid [\![\boldsymbol{gc(a)}]\!](\mathbb{s}, B(\mathbb{s}, \boldsymbol{a}))\ \}$$

where $A_{nov}(\mathbb{s}) \subseteq A(T_{exe}(\mathbb{s}))$ *is the subset of non-override actions in* $A(T_{exe}(\mathbb{s}))$, $\boldsymbol{gc(a)}$ *represents the guard condition of action* $\boldsymbol{a}$*, and* $B(\mathbb{s}, \boldsymbol{a})$ *(defined below) is the set of value bindings for the object variables whose scope includes* $\boldsymbol{a}$*.*

Definition 3.6.14. *Let* $\boldsymbol{a}$ *be an action of a machine transition* $\boldsymbol{t}$*. The object-variable bindings that affect* $\boldsymbol{a}$ *(i.e., affect expressions in* $\boldsymbol{a}$*'s guard condition and WCA) come from two sources: (1)* $\boldsymbol{t}$*'s trigger expression, if any; (2) if* t *is an override transition, the trigger expression, if any, of* t*'s target transition* $\boldsymbol{t_{targ}}$*; and (3) the other actions of* $\boldsymbol{t}$ *that add objects, if any (see Table 3.22). Let* $\boldsymbol{A_{other}}$ *be the actions of* $\boldsymbol{t}$ *other than* $\boldsymbol{a}$*. If* $\boldsymbol{t}$ *has a trigger expression* $te(\boldsymbol{t})$ *and a guard condition* $gc(\boldsymbol{t})$ *(assume* $gc(\boldsymbol{t}) = \mathsf{true}$ *if* $\boldsymbol{t}$ *does not have a guard condition)*

$$B(\mathbb{s}, \boldsymbol{a}) = [\![te(\boldsymbol{t}),\ gc(\boldsymbol{t})]\!]_B(\mathbb{s}) \ \cup\ \boldsymbol{B_{targ}} \ \cup \bigcup_{\boldsymbol{a_{other}} \,\in\, \boldsymbol{A_{other}}} [\![\boldsymbol{a_{other}}]\!]_B$$

where $\boldsymbol{B_{targ}}$ *is the set of object-variable bindings introduced by* $\boldsymbol{t_{targ}}$ *(see Definition 3.6.16). If* $\boldsymbol{t}$ *does not have a trigger*

$$B(\mathbb{s}, \boldsymbol{a}) = \boldsymbol{B_{targ}} \ \cup \bigcup_{\boldsymbol{a_{other}} \,\in\, \boldsymbol{A_{other}}} [\![\boldsymbol{a_{other}}]\!]_B$$

A problem arises if $\boldsymbol{a}$ *refers to an object added by an action in* $\boldsymbol{A_{other}}$ *that is not enabled. To avoid this problem, we require that actions that add objects have no guard conditions so that they are always enabled.*

(2) Recall from Section 3.4 that an override action is enabled when its guard condition (if any) is satisfied and its target action is enabled. Hence, the set $A_{en-ov}(\$)$ of enabled override actions can be defined more precisely as the set of enabled actions that override enabled non-override actions, the enabled actions that override those actions, and so on. To do so, we first define the set of enabled override actions that override a specific set of enabled actions A_{targ}:

Definition 3.6.15. *Let $A_{ov}(\$) \subseteq A(T_{exe}(\$))$ be the subset of override actions in $A(T_{exe}(\$))$ and let A_{targ} be a set of enabled actions in $A(T_{exe}(\$))$.*

$$A_{en-ov}(\$, A_{targ}) = \{\ \boldsymbol{a} \in A_{ov}(\$) \mid [\![\boldsymbol{gc(a)}]\!](\$, B(\$, \boldsymbol{a}))\ \wedge \\ \exists\ \boldsymbol{a_{targ}} \in A_{targ} : \boldsymbol{a}\ \textit{overrides}\ \boldsymbol{a_{targ}}\ \}$$

$A_{en-ov}(\$)$ can now be defined as follows:

Definition 3.6.16. *$A_{en-ov}(\$)$ is the least fixpoint of the function*

$$\lambda\ \$, A_{targ} : A_{en-ov}(\$, A_{en-nov}(\$))\ \cup\ A_{en-ov}(\$, A_{targ})$$

That is, $A_{en-ov}(\$)$ is the limit of the sequence $A^0_{targ}, A^1_{targ}, A^2_{targ}, \ldots$ where

$$A^0_{targ} = A_{en-ov}(\$, A_{en-nov}(\$)),\ \textit{and} \\ A^{i+1}_{targ} = A^i_{targ}\ \cup\ A_{en-ov}(\$, A^i_{targ})$$

(3) Finally, $A_{en}(\$)$ can be defined as

Definition 3.6.17.

$$A_{en}(\$) = A_{en-nov}(\$)\ \cup\ A_{en-ov}(\$)$$

Given the above definition for $A_{en}(\$)$, $A_{exe}(\$)$ is the subset of actions in $A_{en}(\$)$ that are not preempted by other actions in $A_{en}(\$)$:

Definition 3.6.18.

$$A_{exe}(\$) = \{\ \boldsymbol{a_1} \in A_{en}(\$) \mid \nexists\ \boldsymbol{a_2} \in A_{en}(\$) : \boldsymbol{(a_2, a_1)} \in \textit{Preempts}_{act}(\$)\ \}$$

where $\textit{Preempts}_{act}(\$)$ (defined below) is the relation between preemptive actions and their target actions.

Definition 3.6.19.

$$\textit{Preempts}_{act}(\$) = \{\ \boldsymbol{(a_{pre}, a)} \in A(\$) \times A(\$) \mid \boldsymbol{a}\ \textit{is a target action of}\ \boldsymbol{a_{pre}}\ \}$$

3.6.4.2 Computing the Destination State

Let $\mathbb{s} = (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$ and $\mathbb{l} = (\boldsymbol{T}, \boldsymbol{A})$ be the source state and label of an $[\![m]\!]$ transition, respectively. The destination state $\mathbb{s}'$ of the $[\![m]\!]$ transition is the tuple $(\boldsymbol{mode'}, \boldsymbol{ws'_c}, \boldsymbol{ws'_p}, \boldsymbol{ws'_{ci}})$ that results from executing the transitions $\boldsymbol{T}$ and actions $\boldsymbol{A}$ in $\mathbb{l}$, starting from $\mathbb{s}$. The components of $\mathbb{s}'$ are computed from $\mathbb{l}$ and $\mathbb{s}$ as described below.

Computing $\boldsymbol{mode'}$: $\boldsymbol{mode'}$ is the result of updating the configurations of the machines in $\boldsymbol{mode}$ based on the destination states of the transitions $\boldsymbol{T}$. Let $\boldsymbol{smi}$ be a machine of type sm in $\boldsymbol{mode}$.

Definition 3.6.20. *The function for computing the new configuration of $\boldsymbol{smi}$, an sm machine in $\mathbf{mode}$, upon executing the transitions $\boldsymbol{T}$ is defined as*

$$nextConfig(\mathbf{mode}, \boldsymbol{T}, \boldsymbol{smi}) =$$
$$(\boldsymbol{mode}\mathbin{::}\boldsymbol{sm.config(smi)} - \bigcup_{\boldsymbol{t}\,\in\,\boldsymbol{T(smi)}} Exited(\boldsymbol{t})) \;\cup\; \bigcup_{\boldsymbol{t}\,\in\,\boldsymbol{T(smi)}} Entered(\boldsymbol{t}, \mathbf{mode}, \boldsymbol{smi})$$

where

- $\boldsymbol{T(smi)}$ *denotes* $\boldsymbol{smi}$*'s transitions in* $\boldsymbol{T}$.
- $Exited(\boldsymbol{t})$ *returns the states and regions that are exited (i.e., removed from* $\boldsymbol{smi}$*'s configuration) upon executing a transition* $\boldsymbol{t} \in \boldsymbol{T(smi)}$*:*

  $$Exited(\boldsymbol{t}) = \{\boldsymbol{src(t)}\} \;\cup\; \boldsymbol{Anc(src(t))} \;\cup\; \boldsymbol{Des(src(t))}$$

  where $\boldsymbol{Anc(s)}$ *and* $\boldsymbol{Des(s)}$ *denote the ancestors and descendants of state* $\boldsymbol{s}$ *of* $\boldsymbol{smi}$*.*
- $Entered(\boldsymbol{t}, \mathbf{mode}, \boldsymbol{smi})$ *returns the states and regions that are entered (i.e., added to* $\boldsymbol{smi}$*'s configuration) upon executing a transition* $\boldsymbol{t} \in \boldsymbol{T(smi)}$*:*

  $$\begin{aligned} &Entered(\boldsymbol{t}, \mathbf{mode}, \boldsymbol{smi}) = \\ &\quad \{\boldsymbol{dst(t)}\} \;\cup\; (\boldsymbol{Anc(dst(t))} - \{\ \boldsymbol{root}\ \}) \;\cup\; \boldsymbol{initDes(dst(t))} \cup \\ &\quad \boldsymbol{orthogonal(t, mode, smi)} \end{aligned}$$

  In addition to $\boldsymbol{dst(t)}$ *and its ancestors* $\boldsymbol{Anc(dst(t))}$ *(except for the* $\boldsymbol{root}$*), the following states should be entered:*

- *If* $\boldsymbol{dst(t)}$ *is a superstate, all sub machines within* $\boldsymbol{dst(t)}$ *should be initialized. More precisely, the initial descendants of* $\boldsymbol{dst(t)}$ *in* $\boldsymbol{sm}$*'s state hierarchy should be entered.* $\boldsymbol{initDes(s)}$ *denotes the set of initial descendants of state* $\boldsymbol{s}$ *of* $\boldsymbol{smi}$*.* $\boldsymbol{initDes(s)}$ *is empty if* $\boldsymbol{s}$ *is basic state and is otherwise given by*

  $$\boldsymbol{children(s)} \cup \bigcup_{\boldsymbol{r \in children(s)}} \left(\{\boldsymbol{init(r)}\} \cup \boldsymbol{initDes(init(r))} \right)$$

  where $\boldsymbol{children(s)}$ *denotes* $\boldsymbol{s}$*'s children (set of regions), and* $\boldsymbol{init(r)}$ *denotes the initial state of a region* $\boldsymbol{r}$*.*

- *If* $\boldsymbol{dst(t)}$ *is the child state of a region* $\boldsymbol{r}$*, and there are regions orthogonal to* $\boldsymbol{r}$ *(not necessarily siblings of* $\boldsymbol{r}$ *as explained below and in Definition 3.4.3) that were not in* $\boldsymbol{smi}$*'s configuration before executing* $\boldsymbol{t}$*, the sub machines in those regions should be entered and their sub machines initialized. More precisely, let* $\boldsymbol{orthReg}$ *be the regions of* $\boldsymbol{smi}$ *that are orthogonal to* $\boldsymbol{r}$ *and that were not in* $\boldsymbol{smi}$*'s configuration before executing* $\boldsymbol{t}$*:*

  $$\begin{aligned}\boldsymbol{orthReg} = \{\, & \boldsymbol{reg} \in (\boldsymbol{Regions(smi)} - \boldsymbol{mode::sm.config(smi)}) \mid \\ & \boldsymbol{reg} \neq \boldsymbol{r} \;\wedge\; \boldsymbol{isOrth(reg, r)} \,\}\end{aligned}$$

  where $\boldsymbol{Regions(smi)}$ *denotes the regions of* $\boldsymbol{smi}$*. Two regions* $\boldsymbol{r1}$ *and* $\boldsymbol{r2}$ *are orthogonal, denoted* $\boldsymbol{isOrth(r1, r2)}$*, if neither is an ancestor of the other, and their nearest common ancestor (i.e., the common ancestor with the maximum rank) is a state:*

  $$\begin{aligned}\boldsymbol{isOrth(r1, r2)} = & \boldsymbol{r1} \notin \boldsymbol{Anc(r2)} \;\wedge\; \boldsymbol{r2} \notin \boldsymbol{Anc(r1)} \;\wedge \\ & \boldsymbol{maxRank(Anc(r1)} \;\cap\; \boldsymbol{Anc(r2))} \in \boldsymbol{States(smi)}\end{aligned}$$

  where $\boldsymbol{States(smi)}$ *denotes the states of* $\boldsymbol{smi}$ *and* $\boldsymbol{maxRank(S)}$ *denotes the state in set* $\boldsymbol{S}$ *with the highest rank (Definition 3.4.3).* $\boldsymbol{orthogonal(t, mode, smi)}$ *returns the set of states and regions that are entered upon entering the regions in* $\boldsymbol{orthReg}$ *and initializing their sub machines:*

  $$\bigcup_{\boldsymbol{reg} \,\in\, \boldsymbol{orthReg}} (\{\boldsymbol{reg}\} \cup \{\boldsymbol{init(reg)}\} \cup \boldsymbol{initDes(init(reg))})$$

  If $\boldsymbol{orthReg}$ *is empty, then* $\boldsymbol{orthogonal(t, mode, smi)}$ *is the empty set.*

We can now specify $\boldsymbol{mode'}$ more formally as the result of updating the configurations of the machines in $\boldsymbol{mode}$. For each state machine sm in the CBM

$$\begin{aligned}
&\boldsymbol{mode'} :: \boldsymbol{I_{sm}} = \boldsymbol{mode} :: \boldsymbol{I_{sm}} \wedge \\
&\boldsymbol{mode'} :: \boldsymbol{sm.product} = \boldsymbol{mode} :: \boldsymbol{sm.product} \wedge \\
&\boldsymbol{mode'} :: \boldsymbol{sm.config} = \{(\boldsymbol{smi}, nextConfig(\boldsymbol{mode}, \boldsymbol{T}, \boldsymbol{smi})) \mid \boldsymbol{smi} \in \boldsymbol{mode} :: \boldsymbol{I_{sm}}\}
\end{aligned}$$

Caveat: For simplicity of presentation, the above definition of $nextConfig(\boldsymbol{mode}, \boldsymbol{T}, \boldsymbol{smi})$ assumes that the transitions in $\boldsymbol{T(smi)}$ have *non-conflicting destination states*, and thus their execution results in a valid configuration of sm:

$$nextConfig(\boldsymbol{mode}, \boldsymbol{T}, \boldsymbol{smi}) \in Config_{sm}$$

Conflicting destination states cannot be realized simultaneously in a valid configuration (e.g., destination states that are children of the same region). We relax this assumption in Section 4.2.

Computing $\boldsymbol{ws'_c}$, $\boldsymbol{ws'_p}$, and $\boldsymbol{ws'_{ci}}$: The world states $\boldsymbol{ws'_c}$, $\boldsymbol{ws'_p}$, and $\boldsymbol{ws'_{ci}}$ result from applying changes to $\boldsymbol{ws_c}$ or $\boldsymbol{ws_{ci}}$, based on the actions $\boldsymbol{A}$ in the $[\![m]\!]$ transition's label. As explained in Section 3.4, the changes and the world state that they are applied to depend on whether the $[\![m]\!]$ transition represents (1) a simple big step, (2) an intermediate small step in a compound step, or (3) the last small step in a compound big step. The following describes the computation of $\boldsymbol{ws'_c}$, $\boldsymbol{ws'_p}$, and $\boldsymbol{ws'_{ci}}$, for each of these three cases.

Case 1: The $[\![m]\!]$ transition represents a simple big step if both its source state $\mathbb{s}$ and destination state $\mathbb{s}'$ are stable (e.g., transition $\mathbb{s}_0 \xrightarrow{\mathbb{l}_0} \mathbb{s}_1$ in Figure 3.52). In this case, the next observable world state $\boldsymbol{ws'_c}$ is obtained by applying changes to the current observable world state $\boldsymbol{ws_c}$, with the following conditions:

- The changes applied to $\boldsymbol{ws_c}$ are a combination of the actions $\boldsymbol{A}$ in the $[\![m]\!]$ transitions' label and arbitrary changes to uncontrolled phenomena (within the constraints imposed by the world model). Hence, the changes do not normally result in a unique next world state (due to the arbitrary changes), but rather a set of possible next world states denoted by $nextWS_{obs}(\mathbb{s}, \boldsymbol{A})$:

  Definition 3.6.21.

$$\begin{aligned}
&nextWS_{obs}(\mathbb{s}, \boldsymbol{A}) = \\
&\quad \{\ \boldsymbol{ws_{next}} \in \boldsymbol{WS} \mid \boldsymbol{(ws_c, ws_{next})} \in \boldsymbol{WST}\ \wedge \\
&\qquad \forall\ \boldsymbol{a} \in \boldsymbol{A} \mid [\![\boldsymbol{a}]\!]_P(\mathbb{s}, B(\mathbb{s}, \boldsymbol{a}), \boldsymbol{ws_{next}})\ \}
\end{aligned}$$

where $\boldsymbol{WS}$ and $\boldsymbol{WST}$ denote the set of valid world states (see Definition 3.2.7) and valid world-state transitions (see Definition 3.2.8), respectively; and $B(\mathbb{s}, \boldsymbol{a})$ denotes the set of object-variable bindings whose scope includes action $\boldsymbol{a}$ (see Definition 3.6.14).

- The only changes to the controlled phenomena (see Section 3.2.2) in $\boldsymbol{ws_c}$[35] are those specified by the actions $\boldsymbol{A}$. Definition 3.6.21 ensures that a possible next world state includes the changes to controlled phenomena specified by $\boldsymbol{A}$; however, the definition also allows additional arbitrary changes to controlled phenomena (within the constraints imposed by the world model). To omit the additional changes, we limit the set of possible next world states to those world states that have a *minimal* difference from $\boldsymbol{ws_c}$ with respect to controlled phenomena.

  Definition 3.6.22. *Let $D_1 \dots D_n$ be the sets and functions declared in the world model (e.g., object sets, attribute-value functions – see Section 3.2.4) that correspond to controlled phenomena. The difference between two world states $\boldsymbol{ws_1}$ and $\boldsymbol{ws_2}$ with respect to controlled phenomena is the difference in their values of $D_1 \dots D_n$:*

  $$\sum_{1 \le i \le n} (|\boldsymbol{ws_1} :: D_i - \boldsymbol{ws_2} :: D_i| + |\boldsymbol{ws_2} :: D_i - \boldsymbol{ws_1} :: D_i|)$$

  Thus, the (limited) set of possible next world states, denoted $nextWS_{obs-ctrl}(\mathbb{s}, \boldsymbol{A})$, is defined as follows:

  Definition 3.6.23. *$nextWS_{obs-ctrl}(\mathbb{s}, \boldsymbol{A}) \subseteq nextWS_{obs}(\mathbb{s}, \boldsymbol{A})$ is the set of possible next world states that have a minimal difference from $\boldsymbol{ws_c}$ with respect to controlled phenomena (see Definition 3.6.22).*

Hence, $\boldsymbol{ws'_c}$ is some nondeterministically chosen world state in $nextWS_{obs-ctrl}(\mathbb{s}, \boldsymbol{A})$[36], $\boldsymbol{ws'_p}$ records the previously current observable world state ($\boldsymbol{ws_c}$), and since $\mathbb{s}'$ is stable, $\boldsymbol{ws'_{ci}}$ is set to $\boldsymbol{ws'_c}$:

$$\begin{aligned} \boldsymbol{ws'_c} &= some\ nextWS_{obs-ctrl}(\mathbb{s}, \boldsymbol{A}) \\ \boldsymbol{ws'_p} &= \boldsymbol{ws_c} \\ \boldsymbol{ws'_{ci}} &= \boldsymbol{ws'_c} \end{aligned}$$

[35] A change to controlled phenomena in $\boldsymbol{ws_c}$ is either the addition or removal of a controlled object to or from $\boldsymbol{ws_c}$, respectively – except for the removal of transient output and IO message objects; or a change to the value of a controlled attribute, from its value in $\boldsymbol{ws_c}$.

[36] Other behaviours of $[\![m]\!]$ in which an $[\![m]\!]$ transition with actions $\boldsymbol{A}$ executes in $[\![m]\!]$ state $\mathbb{s}$ will result in other nondeterministically chosen next world states $\boldsymbol{ws'_c} \in nextWS_{obs-ctrl}(\mathbb{s}, \boldsymbol{A})$.

where *some* S denotes a nondeterministically chosen member of the set S.

Case 2: The $[\![m]\!]$ transition represents an intermediate step in a compound big step if the transition's destination state $\mathbb{s}'$ is unstable (e.g., transition $\mathbb{s}_1 \xrightarrow{\mathbb{l}_1} \mathbb{s}_{1.1}$ in Figure 3.52). Recall from Section 3.4.3 that, in this case, the next intermediate world state $\boldsymbol{ws'_{ci}}$ is obtained by applying to the current intermediate world state $\boldsymbol{ws_{ci}}$ the changes specified by the actions $\boldsymbol{A}$; the result need not satisfy the world-model constraints. $\boldsymbol{ws'_{ci}}$ can be specified more precisely in two steps. First, let $nextWS_{int}(\mathbb{s}, \boldsymbol{A})$ be the set of possible next (intermediate) world states that result from applying changes to $\boldsymbol{ws_{curri}}$ that include those required by the actions $\boldsymbol{A}$:

Definition 3.6.24.

$$\begin{aligned}
&nextWS_{int}(\mathbb{s}, \boldsymbol{A}) = \\
&\quad \{\ \boldsymbol{ws_{next}} \in \boldsymbol{WS_{univ}} \mid (\boldsymbol{ws_{ci}}, \boldsymbol{ws_{next}}) \in \boldsymbol{WST_{univ}}\ \wedge \\
&\qquad \forall\ \boldsymbol{a} \in \boldsymbol{A} \mid [\![\boldsymbol{a}]\!]_P(\mathbb{s}, B(\mathbb{s}, \boldsymbol{a}), \boldsymbol{ws_{next}})\ \}
\end{aligned}$$

where $\boldsymbol{WS_{univ}}$ *and* $\boldsymbol{WST_{univ}}$ *denote the set of possible (valid or invalid) world states (see Definition 3.2.3) and world-state transitions (see Definition 3.2.6), respectively.*

Next, $\boldsymbol{ws'_{ci}}$ can be specified as a world state $\boldsymbol{ws_{int}} \in nextWS_{int}(\mathbb{s}, \boldsymbol{A})$ that is obtained by applying to $\boldsymbol{ws_{ci}}$ only the changes specified by the actions $\boldsymbol{A}$[37]. In this case, the current observable world state is left unchanged:

$$\begin{aligned}
\boldsymbol{ws'_c} &= \boldsymbol{ws_c} \\
\boldsymbol{ws'_p} &= \boldsymbol{ws_p} \\
\boldsymbol{ws'_{ci}} &= \boldsymbol{ws_{int}}
\end{aligned}$$

Case 3: The $[\![m]\!]$ transition represents the last small step in a compound big step if $\mathbb{s}$ is unstable and the transition's destination state $\mathbb{s}'$ is stable (e.g., transition $\mathbb{s}_{1.m} \xrightarrow{\mathbb{l}_{1.m}} \mathbb{s}_2$ in Figure 3.52). Recall from Section 3.4.2 that, in this case, the next observable world state $\boldsymbol{ws'_c}$ is obtained in a two-step process:

[37] The changes required by a set of actions – without regard for world-model constraints – are deterministic, with two exceptions: (1) In the case of a C+ action, the particular new C object that gets added to the world state is selected nondeterministically. (2) If the arguments of an action include undefined expressions, the result of the action depends on the nondeterministic evaluation of the undefined expressions (see Section 3.3.4).

- Only the changes specified by the actions $\boldsymbol{A}$ are applied to the current intermediate world state $\boldsymbol{ws_{ci}}$. The result of this is a temporary (and possibly invalid) world state $\boldsymbol{ws_n}$ that accumulates the net effect of all of the small steps in the compound big step.

- An implied set of actions $\boldsymbol{A'}$ is computed for the big step that, if applied directly to $\boldsymbol{ws_c}$, would result in $\boldsymbol{ws_n}$ (see Definition 3.4.11). Finally, $\boldsymbol{ws'_c}$ is chosen nondeterministically from $nextWS_{obs-ctrl}(\$, \boldsymbol{A'})$. $\boldsymbol{ws'_p}$ records the previously current observable world state ($\boldsymbol{ws_c}$), and since $\$'$ is stable, $\boldsymbol{ws'_{ci}}$ is set to $\boldsymbol{ws'_c}$:

$$\begin{aligned} &\boldsymbol{ws'_c} = \textit{some } nextWS_{obs-ctrl}(\$, \boldsymbol{A'}) \\ &\boldsymbol{ws'_p} = \boldsymbol{ws_c} \\ &\boldsymbol{ws'_{ci}} = \boldsymbol{ws'_c} \end{aligned}$$

Caveat: For simplicity of presentation, the above definitions of $nextWS_{obs}(\$, \boldsymbol{A})$ (and $nextWS_{int}(\$, \boldsymbol{A})$) assume that the actions in A are *non-conflicting*. That is, there always exists a world-state transition from the current world state ($\boldsymbol{ws_c}$ or $\boldsymbol{ws_{ci}}$) that satisfies all of the actions' expectations. We relax this assumption in Section 4.2.

3.7 Evaluation

This section describes an evaluation of FORML, with respect to the desired properties for a feature-oriented language for modelling SPL requirements that we listed in Section 1.2; as well as for expressiveness. As explained in Section 3.7.1, FORML's satisfaction of most of these properties – all except for ease of evolution and expressiveness – is achieved by design. The properties of expressiveness and ease of evolution have been evaluated by performing two case studies from the automotive and telephony domains, which are described in Section 3.7.2.

3.7.1 Examining FORML's Design

The following desired properties for a feature-oriented language for modelling SPL requirements (see Section 1.2) are achieved in FORML by design:

Using existing standards and best practices: FORML is based on Jackson and Zave's widely-accepted reference model for RE [50]. Recall that this framework defines a system's requirements as desired properties and behaviours of a *world* comprising the system – as a black box – and its environment. The requirements are to be realized about by the system and possibly other agents in the world (e.g., humans, other systems). Following this notion of requirements, a FORML model includes a model of an SPL's world, as well as a model of the SPL's requirements that is expressed in terms of world phenomena. Some requirements are specified as world-model constraints. Most requirements, however, are specified as a behaviour model: a state-machine model whose inputs are events and conditions over the world model, and whose outputs are actions over the world model.

Second, FORML is based on two standard software-engineering notations: the UML and feature models. This can ease adoption by practitioners, since most practitioners are trained in the use of such languages. Of course, practitioners will still need to learn about differences between FORML and the UML, including additional constructs (e.g., fragments, overrides) and differences in the expression language (e.g., trigger expressions, WCAs) and execution semantics (e.g., compound big steps).

Feature modularity: FORML's primary model of an SPL's requirements (i.e., the behaviour model) is decomposed into feature modules. A feature module separately specifies a feature's stand-alone requirements as state machines (Section 3.4.1), and its enhancements of other features' requirements as state-machine fragments that extend other feature modules (Section 3.4.6). Such a decomposition eases the task of tracing a feature to the model of its requirements, and enables independent development of feature modules.

Support for modelling feature enhancements as differences: Sometimes, the purpose of a new feature is to *enhance* an existing feature: that is, to add new requirements in the context of an existing feature's requirements, to modify an existing feature's requirements (an intended feature interaction), or to specify that an existing feature's requirements take precedence over the requirements of a new feature or another existing feature (a retrospective intended interaction). It is natural to model such enhancements in terms of *differences* from the enhanced features, where the model of the enhanced feature is *reused* as the context for expressing the enhancement. In FORML, a new feature's enhancements of existing features' requirements can be modelled as state-machine fragments that extend existing feature modules (Section 3.4.6). FORML prescribes different types of fragments for modelling different types of enhancements:

- New requirements are modelled as new regions, actions, and transitions.

Construct	**Syntax**	**Semantics**
Concepts and feature model	Metamodel for abstract syntax, and exhaustive examples for concrete syntax (Sections 3.2.1 to 3.2.3)	Sets, relations, and expressions over sets and relations over domain elements (Sections 3.2.4 to 3.2.6)
State machines	FSTs for abstract syntax (Section 3.5.1.1), and exhaustive examples for concrete syntax (Section 3.4.1)	State-transition system (Sections 3.6, 4.1, and 4.2)
Expressions	Grammar (Tables 3.4 to 3.17 in Section 3.3 and Tables 3.18 to 3.20 in Section 3.4.1)	Expressions over sets and relations over domain elements (Sections 3.3 and 3.6.3)
Fragments	FSTs for abstract syntax (Section 3.5.1.1), and exhaustive examples for concrete syntax (Section 3.4.6)	FST superimposition (Section 3.5)
Composition	Not applicable	FST superimposition, and pre- and post-superimposition model transformations (Section 3.5)

Table 3.23: The specification of FORML's syntax and semantics

- Intended interactions are modelled as weakening and strengthening clauses, new preemptive transitions and actions, and new (preemptive or non-preemptive) transitions that enter or exit existing states.
- Retrospective intended interactions are modelled as new transition and action priorities.

Furthermore, enhancements of enhancements can be modelled by extending fragments in existing feature modules.

Explicit modelling of intended feature interactions: As described above, FORML prescribes certain fragment types for modelling intended interactions. The prescription makes intended interactions more apparent to the modeller and the reviewer. Furthermore, Section 4.3.2 illustrates the utility of making intended interactions explicit in enabling analyses that automatically distinguish between intended and unintended cases of interactions, such that the analyzer reports only the unintended interactions.

Precision: FORML has a precise syntax and semantics, which makes FORML models amenable to analysis. Table 3.23 summarizes the way in which the syntax and semantics of various FORML constructs are specified in this thesis[38].

Support for multi-product requirements: Some SPLs have requirements associated with the interaction between multiple products. For example, an SPL of telephone services has requirements on interactions that take place between two or more telephone-service products, where each subscriber has his or her own product and feature subscriptions. FORML supports the modelling of such multi-product requirements as follows: A world state can include multiple product objects (each of which is an instance of the SPL concept in the world model). Each state machine in the composed behaviour model (CBM) is instantiated once for every product object in the world state. Each machine's execution specifies the corresponding product's requirements in terms of changes to the world, in reaction to events and conditions in the world. Interactions between products are represented by synchronizations in the executions of their corresponding machines: two machines can synchronize indirectly by reacting to one another's changes in the world, or directly by observing one another's control state (using *inState* predicates, as described in Section 3.4.1.1) or preempting one another's transitions or actions (using override and priority transitions or actions, as described in Section 3.4.4).

Commutative and associative feature composition: Although features are modelled as separate feature modules, the modeller will eventually want to visualize and (manually or automatically) analyze feature combinations that correspond to products of the SPL. In FORML, all of the feature modules in a behaviour model are composed to obtain a model of the whole SPL (i.e., all possible products). The composition of feature modules comprises three steps (see Section 3.5): (1) a preprocessing step that is applied separately to each feature module, (2) a superimposition step that is applied to the set of preprocessed feature modules, (3) and a postprocessing step that adds feature-presence conditions to the superimposed model. Section 3.5.1 shows that the superimposition of the set of preprocessed feature modules is commutative and associative. Hence, the composition of feature modules is commutative and associative.

The composition of feature modules in FORML can lead to savings in analysis costs in two ways: (1) Analyzing the CBM (of the whole SPL) is a cost-effective means of ana-

[38] In some cases, precise descriptions of semantics are given using a combination of natural language and examples (e.g., model transformations that precede and follow the superimposition phase of feature composition).

lyzing an SPL's set of products [7, 24, 41, 86, 8]. (2) Because composition is commutative and associative, only one rather than (possibly) multiple composition orders need to be analyzed.

Ease of evolution: FORML's design partially satisfies the property of ease of evolution in three ways: (1) The addition of a new feature is achieved through localized changes to the behaviour model; specifically, all changes, including intended interactions, are grouped together in a new feature module. (2) A new feature's requirements are is always modelled as *additions* to the model. That is, a new feature *adds* state machines that model new requirements that are independent of the requirements of existing features; and *adds* to existing feature modules extensions (in the form of state-machine fragments) that model its enhancements of existing features[39]. Hence, a new feature never changes or removes behaviour-model elements. This eases evolution, because dangling references to changed or removed elements are avoided. (3) Adding a new feature requires re-composing the CBM for analysis. The new CBM can be computed automatically by (1) preprocessing the new feature module, (2) superimposing the extended set of feature modules, and (3) adding presence conditions to the new superimposed model. We can save on step (2) if the superimposed model of the current set of feature modules is cached from their latest composition: in this case, step (2) simply involves superimposing the new (preprocessed) feature module with the cached superimposed model. Because composition is commutative, we can compose features in the order in which they are developed rather than worry about composing features in a particular order to achieve a particular composed behaviour (e.g., due to feature overrides).

However, because a FORML world model is shared by all of the behaviour model's feature modules, changes to the world model may have an impact on the behaviour model. Possible changes to the world model include new attributes, new concepts, new message parameters, and refactorings (explained below). A change to the world model may syntactically invalidate world-model expressions in existing feature modules. A refactoring operation can cause the removal or renaming of world-model elements (e.g., concepts, attributes, message parameters), resulting in dangling references to the removed (renamed) elements in expressions. For example, renaming a concept *Driver* in an *AutoSoft* world model to *Person* (to account for passengers) would create a dangling reference in the ex-

[39] To recap from Section 3.4.6, a new feature module can include the following types of fragments: new regions that extend existing states, new transitions that extend existing state machines; new states that extend existing regions or state machines; new weakening or strengthening clauses that extend existing conditions by disjunction or conjunction, respectively; and new transition (action) priorities that extend the target lists of existing priority transitions (actions).

pression $\mathsf{Drivers}$. Also, changing the type of an association role, message parameter, or message might create type errors in expressions. For example, changing the type of a message M from an input message to an output message, would invalidate the expression $o.to$ where o is a reference to an M object. Extending the world model with new elements usually does not affect the syntactic correctness of expressions, with the following exceptions:

- A WCA that adds an object to a world state lists the values of the new object's attributes, roles, or parameters. Hence, if an attribute, role, or parameter is added to an existing concept C in the world model, then any existing WCA of the type $+C$ will be missing references to the added element. For example, adding an attribute $startTime$ to the $Call$ association in Figure 3.36 creates a missing value assignment to $startTime$ in the WCA

  $$+\mathsf{Call}(\mathsf{caller} = \mathsf{user}, \mathsf{callee} = \mathsf{o.target}, \mathsf{status} = \mathsf{request})$$

  in transition $t1$ of BCS.
- Making an existing concept C_1 a subtype of a new or existing concept C_2 effectively adds the attributes of the supertype (C_2) to the subtype (C_1). Hence, the addition of the subtype relationship will likely create missing references in existing WCAs of type $+C_1$ as described above.

The above discusses possible ways in which adding a feature to the FORML model of an SPL impacts existing parts of the model. However, it is not obvious how often in practice the addition of a new feature will impact the model or how often a change to the world model will syntactically invalidate existing feature modules. The impact of evolving a FORML model with new features was evaluated by case studies, described in Section 3.7.2.

3.7.2 Case Studies

We have performed two case studies, one from the automotive domain and one from the telephony domain, with the goals of (1) assessing the expressiveness of FORML and (2) evaluating the impact of evolving a FORML world model with new features.

3.7.2.1 The Domains

Telephony: The telephony case study is an extension of *TelSoft* and is adapted from the Second Feature Interaction Contest [59]. In addition to the six features (basic call service

(BCS), call waiting (CW), caller delivery (CD), caller delivery blocking (CDB), voice mail (VM), teen line (TL)) described in Section 3.1, the case study includes nine additional features:

- Call forwarding on busy (CFB): CFB reacts to calls received while the subscriber is in a call and forwards the calls to a designated user.
- Call transfer (CT): CT enables the subscriber to put the remote party of a call on hold, establish a second call with another user, and finally perform a call transfer by establishing a call between the remote party of the first call and the callee of the second call.
- Three-way calling (TWC): TWC allows the subscriber to put the remote party of a call on hold; establish a second call with another user; and finally establish a three-way call between the subscriber, the remote party of the first call, and the callee of the second call.
- Group ringing (GR): GR reacts to a call to the subscriber by establishing additional calls from the caller to two designated users. As soon as the subscriber or either of the designated users accepts his or her call, the remaining two calls are terminated.
- Ringback when free (RBF): RBF remembers the caller of the first call received while the subscriber is in a call, and calls that user when the subscriber ends his/her current call.
- Terminating-call screening (TCS): TCS blocks calls to the subscriber from designated users.
- Billing: For every call, *billing* charges the caller's account, based on the designated billing rate and the duration of the call.
- Reverse charging (RC): RC charges the subscriber for incoming calls.
- Split billing (SB): SB splits the charge between the subscriber and his or her callers based on a designated percentage.

The telephony case study makes the following adaptations to the original feature descriptions in [59]:

- The original description of BCS includes billing requirements. To better exercise FORML, *Billing* is modelled as a separate feature in the case study.

- In the original descriptions, the telephone services of the users involved in a call synchronize via message passing over a network. The case study abstracts away from the network and models product synchronization in terms of products monitoring one another's changes to a shared context (e.g., the callee's *TelSoft* product reacting to a call created by the caller's *TelSoft* product).

- The case-study models simplify the original descriptions by omitting several output messages sent by telephone services to their users (e.g., CW sending a special tone to a waiting user). The inclusion of such messages would not further exercise the expressiveness of FORML.

Automotive: The automotive case study is an extension of *AutoSoft* and is adapted from GM Feature Technical Specifications for a family of automotive software features. In addition to the three features (basic driving service (BDS), cruise control (CC), headway control (HC)) described in Chapter 3, the case study includes eight additional features:

- Lane change alert (LCA): LCA issues an alert if the driver tries to change lanes when the conditions are not safe for a lane change.

- Forward collision alert (FCA): FCA issues an alert whenever there is danger of a collision with an object ahead, and stops the alert when the danger passes. The alert level can be set by the driver.

- Headway personalization (HP): HP saves the last cruise-headway setting of a driver, and automatically recalls this setting the next time the driver uses the car.

- Speed limit control (SLC): SLC overrides CC's computation of the car's acceleration, whenever the set cruising speed is greater than the speed limit of the road segment that the car is on.

- Lane centring control (LCC): LCC periodically adjusts the car's orientation to centre the vehicle in its current lane. To avoid skidding during lane centring, LCC overrides CC's computation of the car's acceleration, whenever the set cruising speed is greater than the computed safe speed for lane centring.

- Lane change control (LXC): LXC automatically changes the vehicle's lane to a driver-selected lane, provided the conditions are safe. Once the car enters the destination lane, LXC delegates to LCC to centre the car in the destination lane. If the conditions become unsafe during a lane change, LXC delegates to LCC to centre the car in the original lane, if it is safe to do so. If it is not safe to return to the original lane, LXC issues a message to request that the driver take control of the steering. In order to avoid skidding during a lane change, LXC overrides CC's computation of the car's acceleration, whenever the set cruising speed is greater than the computed safe speed for a lane change. In the presence of LXC, FCA issues forward-collision alerts under additional conditions that are specific to lane changes. Finally, the activation and deactivation of LXC automatically engages and disengages LCA.

- Road change alert (RCA): RCA issues an alert when the shape of the car's current lane changes, or, in the case of a lane change, when the shape of the destination lane changes. The alert includes the type of the road change.

- Driver monitoring system (DMS): DMS issues an alert whenever the driver is not attentive, and continues to issue alerts periodically so long as the driver remains unattentive.

The automotive case study makes the following adaptations to the original feature descriptions:

- The case study abstracts away from detailed environment phenomena referenced in the original description, such as detailed properties of vehicles (e.g., longitudinal acceleration) and roads (e.g., curvature). The abstractions manifest in the case study as missing elements or undefined attribute types in the world model, and as undefined set and predicate expressions in the behaviour model. For example, in LXC's original requirements, the vehicle's longitudinal acceleration is used to determine whether a lane change is safe. In the corresponding FORML model, LXC's condition for a safe lane change is abstractly modelled as an undefined predicate named $safeLX$ (see Figure A.47).

- The case study abstracts away from interface devices in the original description, such as the gap switch used by the driver to set the desired headway distance for HC. The inputs and outputs of such devices are abstractly represented as messages in the world model.

- In the original description, features often synchronize via message passing over the vehicle's internal network. The case study abstracts away from the network, and models feature synchronization in terms of features monitoring one another's changes to a shared context (e.g., LCA reacting to BDS's change of the vehicle's steering direction).

- The case study omits some behaviours that are of the same nature as other behaviours that were modelled; the omitted behaviours would not further exercise the expressiveness of FORML. For example, the original description of CC includes a delayed disengagement behaviour that is not (but could be) modelled in the case study: when CC's disengagement condition is not due to driver actions, CC disengages after a timeout period (FORML does not have timing constructs; however, a timeout period can be modelled as an undefined predicate). Also, the case study omits behaviours that reference features described in other GM documents (e.g., traction control).

In the case studies, each feature was modelled twice: once to evaluate expressiveness, and again (at a later time) to evaluate ease of evolution[40]; the purpose of the second study was to assess the impact of evolving a FORML world model with new features.

3.7.2.2 Evaluation of Expressiveness

The case studies were performed in *exploratory* and *confirmatory* phases. In the exploratory phase, the language was refined as needed to specify completely a small subset of the case studies' features (specifically, the BCS, CW, and CFB features in the telephony case study and the BDS, CC, and HC features in the automotive case study). In the confirmatory phase, the refined language was applied to the rest of the features to assess expressiveness. The modelling of the CDB feature in the confirmatory phase led to the development of the $ construct (see Section 3.5)[41]. Otherwise, the language features developed during the exploratory phase were sufficiently expressive for modelling the confirmatory-phase features.

The confirmatory phase resulted in several language refinements to improve usability. Most such refinements were simply changes to the syntax of existing fragment types, with the goal of making fragments easier to read and write (e.g. changing the syntax for specifying clauses, making references to extended elements more concise). The exception was a refinement that added a new fragment type to the language for modelling retrospective intended interactions (see Section 3.4.6.3): that is, new target transitions and actions (see Section 3.4.6.3), which aim to ease the task of specifying that an existing transition (action) has priority over a new transition (action)[42].

[40] Ease of evolution was chosen as an evaluation goal after the first round of modelling to evaluate expressiveness.

[41] Upon revisiting the exploratory-phase models, it was discovered that the $ construct is needed also in the model for the CFB feature.

[42] This new fragment type does not change the expressiveness of FORML. It is possible to specify that

feature	use of fragments for added requirements	use of fragments for interaction interactions	number of $ constructs	number of undefined expressions	evolution of the world model
automotive					
BDS (sm)				3	
CC	1r (BDS)	1sc (BDS)		4	3m
HC	1r (CC)	1pt(CC)		2	1e, 1sub, 1a, 1m, ref
LCA	1r (BDS)			1	3e, 1a, 2sub, 3m, ref
FCA	1r (CC)			1	3m
HP	2a (HC)	1t (HC)			1e, 2a
SLC	1r (CC)	1pt (CC)		2	1at, 2m
LCC	1a (BDS), 1r (CC)	1pt (CC)		4	2m
LXC	1ss, 1bs, 2t (LCC)	1wc, 1sc (FCA), 2t (LCA), 1pt, 9t (LCC)		5	3m
RCA	1r (LCC), 1r (LXC)			2	1m
DMS	1r (LCC)			1	1at, 1m
telephony					
BCS (sm)					
CW	1bs, 2t (BCS)	4pt, 2tt (BCS)			1m
CD	1a (BCS)				1m
CDB		1sc (CD)	1		
CFB	1r (BCS)	1sc (BCS)	1		1a
CT	1bs, 1ss, 5t (BCS)	2sc, 4pt, 2t (BCS)	1		1a, 1m, 1ref
TWC	3bs, 1ss, 8t (BCS)	1sc, 3pt, 2t, 2tt (BCS)	4		2a, 2m
GR	4a (BCS)	3t, 2wc (BCS)	4		4a, 1ref
RBF	1a (BCS)	1pt (BCS)			1a
TL	1bs, 2t (BCS)	1pt (BCS)		2	1a, 2m
TCS		1sc (BCS)			1a
VM	2a (BCS)	1sc, 2pt, 1t (BCS)	4	1	2e, 2a, 1sub, 1m
Billing (sm)				3	1e, 2a
RC		1pa (Billing)	1		
SB		1pt (Billing)	1	2	

add and interaction columns:
Table entries list numbers of extensions to feature (F):
r: region, **bs:** basic state, **ss:** super state, **t:** non-preemptive transition, **a:** non-preemptive action,
pa: preemptive action , **pt:** preemptive transition, **tt:** transition target, **wc:** weakening clause,
sc: strengthening clause

world model column:
Table entries list numbers of changes to the world model:
e: entity, **a:** association, **m:** message, **at:** attribute, **sub:** subtype,
ref: refactoring operation

Table 3.24: Summary of case studies

an existing transition (action) x has priority over a new transition (action) y, without adding y as a target of x: one could add a transition (action) z that (1) overrides x; (2) has priority over y; and (3) has the

Table 3.24 summarizes (in the first five columns) the degrees to which different FORML constructs are used in the case-study models[43]. The requirements for BDS, BCS, and billing are each modelled as a single state machine, as denoted in the table by the postfix *(sm)*. The other features' requirements are modelled as fragments; the second and third columns report the numbers and types of fragments used to specify each feature's added requirements and intended interactions, respectively, in the context of an existing feature's requirements. The fourth column states the number of uses of the $ construct in each feature module, and the fifth column gives the number of undefined expressions that were introduced to abstractly represent data logic and constants. For example, CW (Figure 3.38) from the telephony case study adds one basic state (1bs), two non-preemptive transitions (2t), four preemptive transitions (4pt), and two target transitions (2tt) to BCS's feature module (Figure 3.37). CW's feature module does not use the $ construct, and does not introduce any undefined expressions. The complete set of models is given in Appendix A.

3.7.2.3 Evaluation of Ease of Evolution

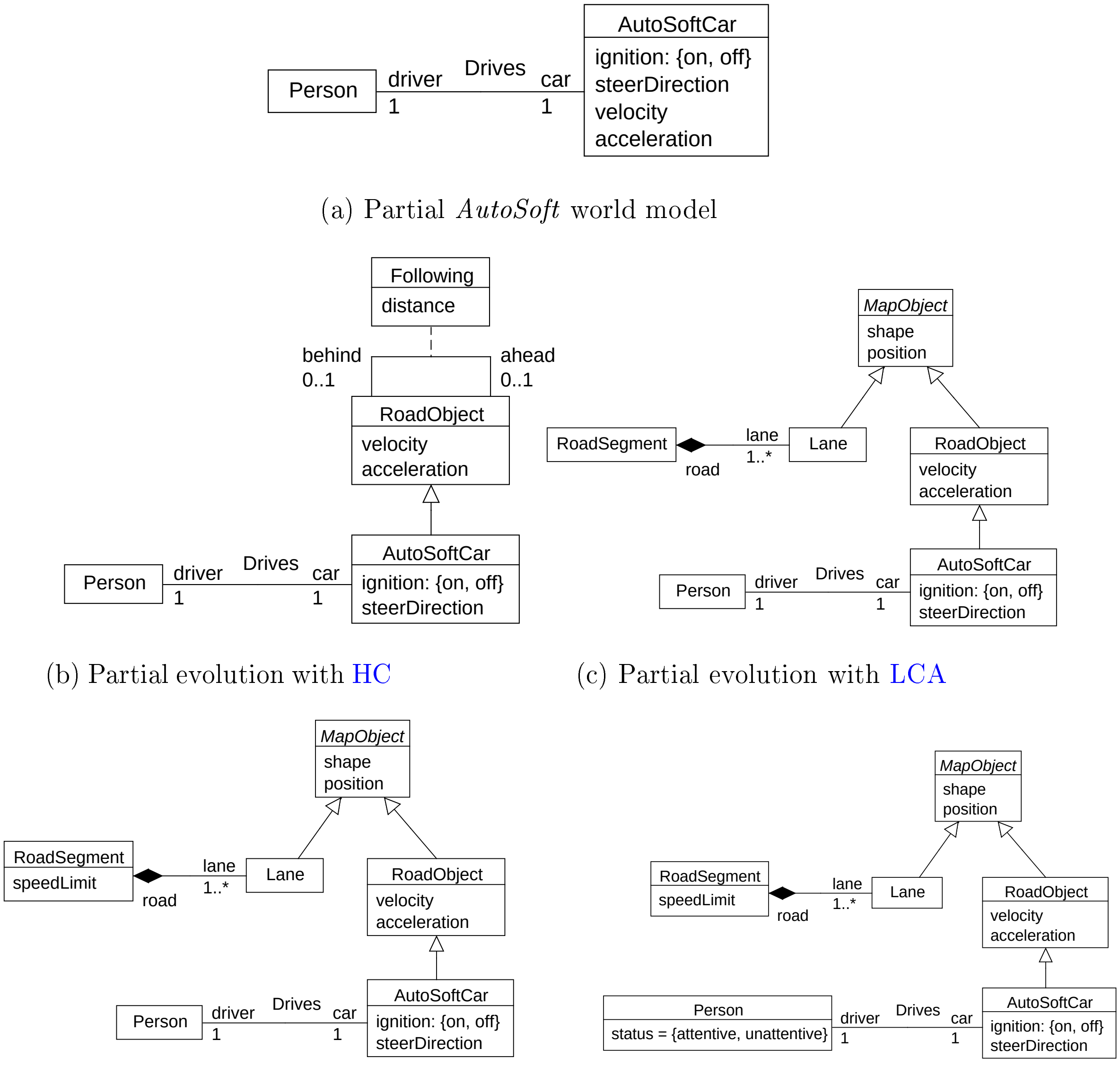
In each case study, the SPL's requirements were modelled incrementally by adding features one at a time. The features were modelled in the order in which they are listed in Table 3.24. The order was constrained by enhancement relationships between feature types (e.g., HC enhances CC), but was otherwise random[44]. The *world model* column in Table 3.24 indicates how the world model changed with the addition of each feature, to reflect new concepts, attributes, associations, and so on – excluding new feature concepts and changes to the feature model. For example, feature HC from the automotive case study adds one entity (1e), one subtype relationship (1sub), one association (1a), and one message (1m); and performs a refactoring operation (ref), which is described below. The evolution of the world model (with the addition of each feature) is shown in Appendix A for each case study.

Most features from the case studies merely extended the world model with new concepts and therefore did not syntactically invalidate existing feature modules. The following fea-

same source state, destination state, and actions as x. However, this approach does not clearly reflect the intent of prioritizing x over y.

[43] The data corresponds to the most recent version of the models: that is, the models produced in the case studies to evaluate ease of evolution.

[44] Here, we refer to the order of *modelling* features and not the order of *composing* features. It is realistic to order the modelling of features according to their enhancement relationships (e.g., in reality, HC is introduced and modelled after CC). However, as described in Section 3.5, the order of composing the features (regardless of their enhancement relationships) does not affect the result.



(a) Partial *AutoSoft* world model

(b) Partial evolution with HC

(c) Partial evolution with LCA

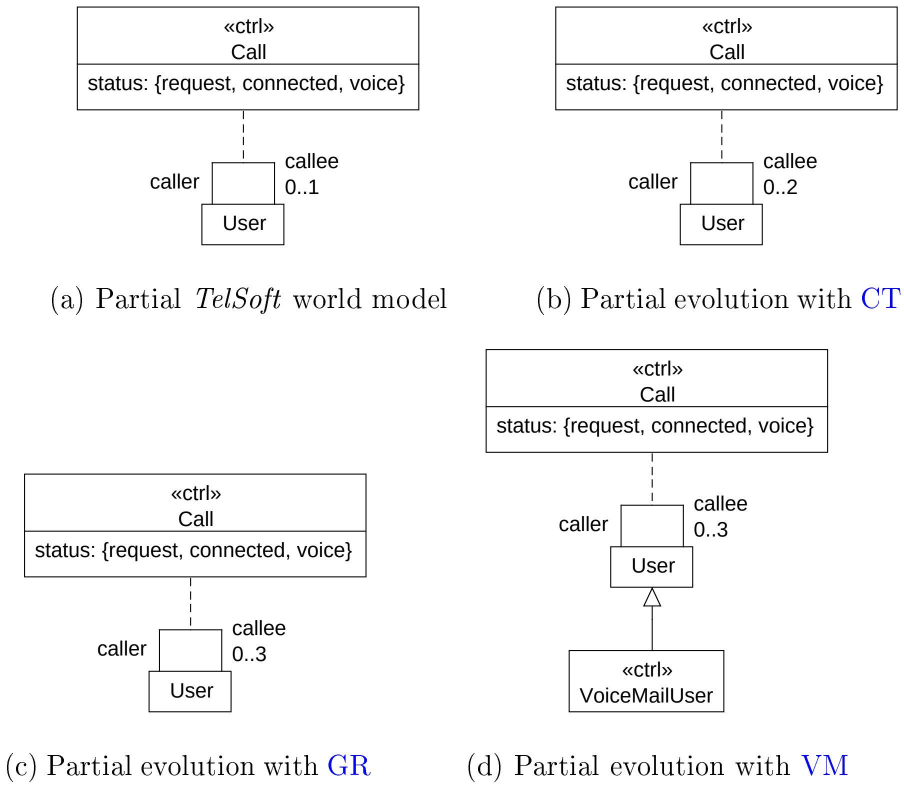
(d) Partial evolution with SLC

(e) Partial evolution with DMS

Figure 3.57: Partial evolution of the *AutoSoft* world model

tures additionally made other types of changes to the world model; however, for the reasons described below, the changes did not syntactically invalidate existing feature modules.

- As shown in Figure 3.57b, the introduction of the feature HC lead to a refactor-


«ctrl»
Call
status: {request, connected, voice}
caller
callee
0..1
User
«ctrl»
Call
status: {request, connected, voice}
caller
callee
0..2
User
«ctrl»
Call
status: {request, connected, voice}
caller
callee
0..3
User
«ctrl»
Call
status: {request, connected, voice}
caller
callee
0..3
User
«ctrl»
VoiceMailUser

(a) Partial *TelSoft* world model (b) Partial evolution with CT

(c) Partial evolution with GR (d) Partial evolution with VM

Figure 3.58: Partial evolution of the *TelSoft* world model

ing of the world model in which *AutoSoftCar* becomes a subtype of a new entity *RoadObject*, and the *velocity* and *acceleration* attributes of *AutoSoftCar* are moved to its supertype *RoadObject*. These changes did not invalidate existing feature modules, because *RoadObject* did not introduce new attributes (which would have lead to missing references in existing +AutoSoftCar WCAs, had the existing feature modules included any +AutoSoftCar WCAs).

- As shown in Figure 3.57c, the introduction of the feature LCA lead to a refactoring of the world model in which the association *Following* is removed, and the entity *RoadObject* and a new entity *Lane* become subtypes of a new entity *MapObject*. These changes did not invalidate existing feature modules, because (1) there were no existing references to the removed association *Following* (computations over distances between vehicles are abstracted away by the undefined predicate *slowRoadObjectAhead* introduced by HC); (2) there were no existing +AutoSoftCar or +RoadObject

WCAs to be invalidated by missing references to the new attributes that *RoadObject* and *AutoSoftCar* inherited from *MapObject*; and (3) there were (obviously) no existing references to the new subtype *Lane* of *MapObject*.

- As shown in Figures 3.57d and 3.57e, the introduction of the features SLC and DMS lead to the addition of the attributes *speedLimit* and *status* to the entities *RoadSegment* and *Person*, respectively. These changes did not invalidate existing feature modules, because there were no existing +RoadSegment and +Person WCAs to be invalidated by missing references to the added attributes.

- As shown in Figures 3.58b and 3.58c, the introduction of the features CT and GR lead to changes to the multiplicity of the *callee* role of the *Call* association, from 0..1 to 0..2 and from 0..2 to 0..3, respectively. These changes did not invalidate existing feature modules because, in general, weakening multiplicities does not syntactically impact world-model expressions.

- As shown in Figure 3.58d, the introduction of the feature VM lead to the addition a new entity *VoiceMailUser* as a subtype of *User*. These changes did not invalidate existing feature modules, because there were (obviously) no existing references to the new subtype *VoiceMailUser* of *User*.

In general, we believe that the impact due to evolving the world model is small: refactoring operations are not likely to occur frequently, and the invalidation of existing WCAs can be avoided by assigning default values to new attributes, roles, and parameters. Furthermore, it is relatively simple to develop analyses that automatically detect syntax errors that are introduced in feature modules by a refactoring operation on the world model.

3.7.2.4 Threats to Validity

The following are threats to the internal and external validity of the results of evaluating FORML with respect to expressiveness and ease of evolution.

Internal validity:

- The case study models were produced by the developer of the FORML language, which can bias the evaluation results. We tried to mitigate this threat by using case studies that originated from external sources (the Second Feature Interaction Contest [59] and GM Feature Technical Specifications). The use of external sources reduces the risk of formulating the case studies in favour of FORML.

- Modelling errors could effect the evaluation results in FORML's favour. For example, the exploratory phase did not reveal the need for the $ construct because of an error in modelling CFB (the need for the $ construct was later revealed in modelling CDB in the confirmatory phase). We tried to mitigate this threat through careful modelling.
- The order in which the features were modelled in evaluating ease of evolution could effect the results. We tried to mitigate this threat by choosing an arbitrary order (within the realistic constraint of modelling an enhanced feature before its enhancements).
- Ease of evolution and expressiveness were evaluated with respect to the same set of features and by the same modeller. The modeller's experience in modelling the features during the first study could affect his choices in re-modelling the features during the second study. The modeller tried to mitigate this threat by modelling each feature without thinking about the features not yet modelled (the fact that at least five months had elapsed between the two studies helped this cause).
- As explained in Section 3.7.2.1, the automotive case study omits some behaviours from the original feature descriptions. The omitted behaviours were deemed similar enough to already-modelled behaviours, so as not to affect the evaluation results. It is possible, of course, that some omitted behaviours would indeed affect the evaluation results despite our assessment to the contrary.

External validity: The evaluation results are based on the modelling of two case studies from only two domains: the telephony and automotive domains. The effect of this threat is somewhat mitigated by the fact that the two domains are quite diverse: For example, the telephony world consists mostly of controlled phenomena (e.g., calls), whereas the automotive world consists mostly of phenomena that can be controlled by both products and the environment (e.g., the vehicle's acceleration). Also, most telephony requirements involve interactions between products, whereas automotive requirements typically concern single products.

3.8 Chapter Summary

This chapter presents FORML, a feature-oriented language for modelling the behavioural requirements of an SPL. A FORML model of an SPL's requirements comprises two main views: (1) a *world model*, which is an ontology of concepts that describe the SPL's products and the environment in which they will operate; and (2) a *behaviour model*, which is a state-machine model of the SPL's requirements. The behaviour model is decomposed into *feature modules*. FORML enables specifying a feature's *enhancements* (i.e., extensions or modifications) of other features as *state-machine fragments* that extend other feature modules. In particular, some fragments explicitly model *intended features interactions*, where one feature modifies the behaviours of another feature by design.

To obtain a view of the requirements of feature combinations, all of the feature modules are composed into a model of the whole SPL; that is, the composed model represents all derivable products. Such a model enables more efficient analyses of the SPL's set of products [7, 24, 41, 86, 8]. At the heart of composition is the integration of the feature modules' fragments and state machines. This step of the composition is formalized as the *superimposition* of the feature modules' abstract syntax trees, called feature structure trees (FSTs). FORML FSTs have properties (e.g., being unordered, not having common terminal nodes) that ensure that their superimposition is commutative and associative. The commutativity and associativity of superimposition (and hence composition) can lead to savings in analysis costs, since only one rather than (possibly) multiple composition orders need to be analyzed.

We performed two case studies from the automotive and telephony domains, the results of which indicate good expressiveness for FORML (a minor adjustment to FORML – the introduction of the $ construct – was needed to improve expressiveness in modelling 20 new features[45]), and low impact of adding new features to a FORML model (adding new features did not result in changes to existing feature modules).

[45] Had the need for the $ construct been identified in modelling CFB during the exploratory phase, then no improvements to expressiveness would be needed in the confirmatory phase. However, note that the confirmatory phase did introduce several language refinements to improve usability (e.g., changes to the syntax of existing fragment types).

Chapter 4

Feature Interactions in FORML

As stated in Chapter 1, an important challenge in FOSD is detecting *unintended feature interactions*: that is, cases where different features unexpectedly influence one another in defining the overall properties and behaviours of their combination [93]. To enable detecting unintended feature interactions, we must define the different ways that feature interactions can *manifest* in a given description of the features' combination.

This chapter describes a taxonomy of feature interactions for FORML models, comprising the following types of feature interactions. Each type of feature interaction is informally described and illustrated by an example, and is then formally defined in terms of the semantic state-transition system described in Section 3.6.

- A **nondeterminism interaction** occurs when there is a nondeterministic choice in the set of enabled transitions that can execute in a small step (see Section 4.1).
- A **conflict interaction** occurs when the set of actions that execute in a small step are inconsistent with one another or with the world model, or when the set of transitions that execute in a step have mutually inconsistent destination states (e.g., the destination states are different child states in the same region; see Section 4.2).
- A **modification interaction** occurs when one feature modifies the behaviours of another feature (see Section 4.3). The notion of a modification interaction generalizes a number of existing feature-interaction types that are special cases of behaviour modification. For completeness, the proposed taxonomy includes two types of feature interactions that are prominent special cases of modification interactions: **deadlock interaction**, which occurs when a feature causes a deadlock that prohibits another

feature's behaviours; and **looping interaction**, which occurs when two features trigger one another's behaviours in an infinite loop. FORML's constructs for modelling intended feature interactions (see Section 3.4.6.2) cause modification interactions. To enable feature-interaction analyses that report only unintended modification interactions, the proposed definition of modification interactions excludes the cases caused by such constructs.

Our proposed feature-interaction taxonomy can be compared to existing taxonomies defined for feature-oriented artifacts (FOAs) expressed as operational models (see Section 2.3):

- Nondeterminism, deadlock, and conflict interactions adapt corresponding feature-interaction types in existing taxonomies (i.e., items 1a, 1b, and 1c in Section 2.3's feature-interaction list, respectively). Resource contentions (item 1d in Section 2.3) can be viewed as conflict interactions caused by inconsistent changes to resources represented in the world model. Similarly, an action in one feature module violating an assertion raised by another feature (item 1e in Section 2.3) can be viewed as a conflict interaction, where the violated assertion is specified as a world-model constraint. Conflicting actuator requests [53] correspond to our notion of conflict interactions. We do not consider conflicts that occur within a time threshold [53], because FORML does not explicitly model time.
- Looping interactions formalize informal notions of infinite loops in existing taxonomies (item 1f in Section 2.3).
- Modification interactions formalize the notion of a feature's actions triggering or prohibiting behaviours of another feature (item 2 in Section 2.3). Furthermore, modification interactions generalize looping and deadlock interactions, and exclude intended interactions.
- Because FORML's feature composition is commutative and associative, we do not consider feature interactions that manifest as sensitivity to composition order (item 3 in Section 2.3).

Before proceeding to the feature-interaction definitions, it is important to emphasize that feature interactions occur between *instances* of features in products. Thus, our definitions of feature interactions specify properties of the executions of machines (state-machine instances) associated with products, and refer to associations between machines and feature objects, so as to identify the feature instances involved in an interaction. For example, conflict interactions refer to inconsistencies between the actions of different features objects in the same or different machines. The association between feature objects and elements

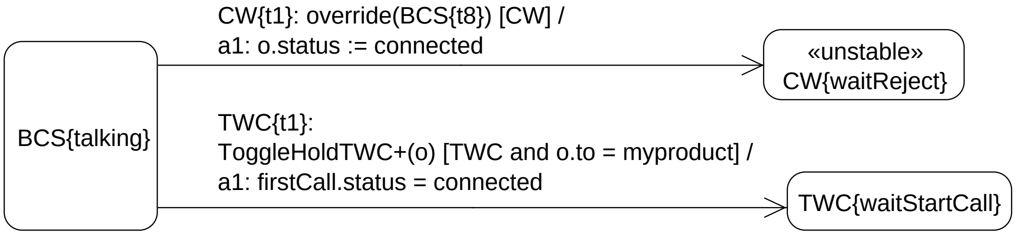

CW{t1}: override(BCS{t8}) [CW] /
a1: o.status := connected
«unstable»
CW{waitReject}
BCS{talking}
TWC{t1}:
ToggleHoldTWC+(o) [TWC and o.to = myproduct] /
a1: firstCall.status = connected
TWC{waitStartCall}

Figure 4.1: Example of a nondeterminism interaction

in a state-machine instance corresponds to the association between feature concepts and elements in a state-machine model. For example, if an action a of a state machine sm is introduced by a feature F, then the a action of an sm machine (instance) corresponding to a product p is associated with the F feature object in p.

4.1 Nondeterminism Interactions

Recall that Sections 3.4 and 3.6 describe the semantics of a composed behaviour model (CBM) assuming that it comprises deterministic state machines. This section relaxes this assumption and considers the case where the CBM includes nondeterministic state machines; that is, it is possible for two or more nonconcurrent, unpreempted transitions to be enabled in the same small step. From each set of such nonconcurrent transitions, at most one transition is *nondeterministically* chosen for execution. Nondeterminism in a CBM may be introduced by a single feature; that is, nonconcurrent transitions of the same feature that are enabled at the same time. When nonconcurrent, unpreempted transitions from different features are enabled in the same small step, we say that a *nondeterminism interaction* has occurred between those features.

Example: Figure 4.1 shows a fragment of the composed *TelSoft* model, which includes two transitions $CW\{t1\}$ and $TWC\{t1\}$ introduced by features CW and TWC, respectively. The transitions have the same source state $BCS\{talking\}$ introduced by BCS. $CW\{t1\}$ models the activation of CW, which occurs when the subscriber receives a second call while he or she is already in a voice call. $TWC\{t1\}$ models the activation of TWC, which occurs when the subscriber requests to put an established voice call on hold, so as to set up a three-way call that adds a third user to the original call. It is possible for both features to be simultaneously activated, in which case both transitions are simultaneously enabled, and a nondeterminism interaction between CW and TWC occurs. □

4.1.1 Formal Definition

This section first revises the definition of the state-transition system $[\![m]\!]$, introduced in Section 3.6, to account for nondeterminism. Then, nondeterminism interactions are formally defined in the context of the revised definition.

Revised semantics: To account for nondeterministic CBMs, we must revise the definition of $T_{exe}(\mathbb{s})$, introduced in Definition 3.6.11 as the subset of enabled transitions to be executed in an $[\![m]\!]$ transition $(\mathbb{s}, \mathbb{l}, \mathbb{s}')$. The revision affects the value of the label $\mathbb{l}$; however, the definition of how the destination state $\mathbb{s}'$ is computed is unchanged.

Specifically, the revised definition partitions the set of enabled transitions computed in Definition 3.6.11 into subsets of nonconcurrent transitions. Each partition represents a nondeterministic choice to be made: thus, the revised definition nondeterministically selects one transition for execution from each partition.

Definition 4.1.1. *Let $T_{en-np}(\mathbb{s})$ denote the set of enabled transitions in an $[\![m]\!]$ transition $(\mathbb{s}, \mathbb{l}, \mathbb{s}')$ that are not preempted by another enabled transition (Definition 3.6.11 revisited):*

$$T_{en-np}(\mathbb{s}) = \{ \ \boldsymbol{t_1} \in T_{en}(\mathbb{s}) \mid \nexists \ \boldsymbol{t_2} \in T_{en}(\mathbb{s}) : (\boldsymbol{t_2}, \boldsymbol{t_1}) \in Preempts_{tran}(\mathbb{s}) \ \}$$

where $T_{en}(\mathbb{s})$ denotes the set of enabled transitions (Definition 3.6.10) and $Preempts_{tran}(\mathbb{s})$ denotes the preemption relation between transitions (Definition 3.6.12).

Let the sets $T_{nc_1}(\mathbb{s}), \ldots, T_{nc_k}(\mathbb{s})$ denote a partitioning of $T_{en-np}(\mathbb{s})$, where two transitions are in a partition if and only if they are nonconcurrent (Definition 3.4.7). We can now provide a revised definition, denoted $T^n_{exe}(\mathbb{s})$[1], of the set of transitions to be executed in an $[\![m]\!]$ transition:

$$T^n_{exe}(\mathbb{s}) = \bigcup_{1 \ \leq \ j \ \leq \ k} \{some \ T_{nc_j}(\mathbb{s})\}$$

Note that if the CBM is deterministic, there will be a singleton partition for each transition in $T_{en-np}(\mathbb{s})$ and therefore $T^n_{exe}(\mathbb{s}) = T_{en-np}(\mathbb{s})$.

A nondeterminism interaction occurs when an $[\![m]\!]$ transition requires a nondeterministic choice between transitions that belong to different features.

[1] The superscript n denotes versions of definitions that apply to nondeterministic models.

Definition 4.1.2. *Let $T_{en-np}(\mathbb{s})$ denote the set of enabled transitions in an $\llbracket m \rrbracket$ transition $(\mathbb{s}, \mathbb{l}, \mathbb{s}')$ that are not preempted by another enabled transition (Definition 4.1.1). A nondeterminism interaction occurs when*

- *Some partition $T_{nc_j}(\mathbb{s})$ of $T_{en-np}(\mathbb{s})$ contains more than one transition*
- *The transitions in $T_{nc_j}(\mathbb{s})$ belong to a set F of two or more features*

In this case, we say that there is a nondeterminism interaction between the features in F.

4.2 Conflict Interactions

This section relaxes the assumption in Sections 3.4 and 3.6, that a small step of a CBM never results in a *conflict*. A small step results in a conflict under either of the following conditions:

- The small step includes transitions with *conflicting destination states*: that is, the transitions' destination states cannot be realized simultaneously in a valid configuration of some state-machine instance.
- The small step results in *conflicting actions*:
  - If the small step forms a simple big step, conflicting actions mean that there is no valid world-state transition that leads to a next observable world state in which the small step's actions are all satisfied.
  - If the small step occurs within (but not at the end of) a compound big step, conflicting actions mean that there is no (valid or invalid) world-state transition that leads to a next (intermediate) world state in which the small step's actions are all satisfied.
  - Finally, if the small step is the last one in a compound big step, conflicting actions mean that (1) there is no (valid or invalid) world-state transition that leads to a temporary world state $\boldsymbol{ws_n}$ in which the small step's actions are all satisfied; or (2) there is no valid world-state transition that leads to a next observable world state in which the implied set of actions computed for the big step (see Definition 3.4.11) are all satisfied.

As with nondeterminism, conflicts may be caused by transitions or actions that are specified by the same single feature or by different features. The latter case gives rise to *conflict interactions*: in the case of conflicting destination states, the transitions involved come from different features; and in the case of conflicting actions, the actions belong to different features[2]. Note that conflicting actions from different features can belong to the same transition (when the actions are fragments introduced by different features that extend the transition), to different transitions of the same feature, or to different transitions of different features.

Example: Figure 4.2 shows a fragment of the composed *AutoSoft* model, which includes a transition $CC\{t6\}$ of CC, a transition $HC\{t2\}$ of HC that overrides $CC\{t6\}$ when there is a slower road object ahead, and a transition $SLC\{t3\}$ of SLC that overrides $CC\{t6\}$ when the speed limit is exceeded. Suppose that $HC\{t2\}$ and $SLC\{t3\}$ are concurrently enabled (i.e., there is a slower road object ahead and simultaneously the speed limit is exceeded). The actions $HC\{a1\}$ of $HC\{t2\}$ and $SLC\{a1\}$ of $SLC\{t3\}$ may simultaneously try to reduce the vehicle's acceleration by different amounts, so as to maintain a desired headway distance from the vehicle ahead and to align the vehicle's speed with the speed limit, respectively. In this case, a conflict interaction occurs between features HC and SLC.

As another example, Figure 4.3 shows a fragment of the composed *TelSoft* model, which includes two concurrent transitions: transition $TL\{t2\}$, which is enabled when the subscriber enters an invalid PIN; and transition $CFB\{t1\}$, which is enabled when a second call arrives while the subscriber is in a call. When $TL\{t2\}$ and $CFB\{t1\}$ are simultaneously enabled, a conflict interaction occurs between the TL and CFB features because $TL\{t2\}$ and $CFB\{t1\}$ have conflicting destination states: $BCS\{idle\}$ and $BCS\{inCall\}.CFB\{main.waitCall\}$, respectively. □

[2] Note that in general, the computed actions of a compound big step cannot be associated with feature objects; hence, if the computed actions conflict, it may not be possible to exactly determine the features involved in the interaction.

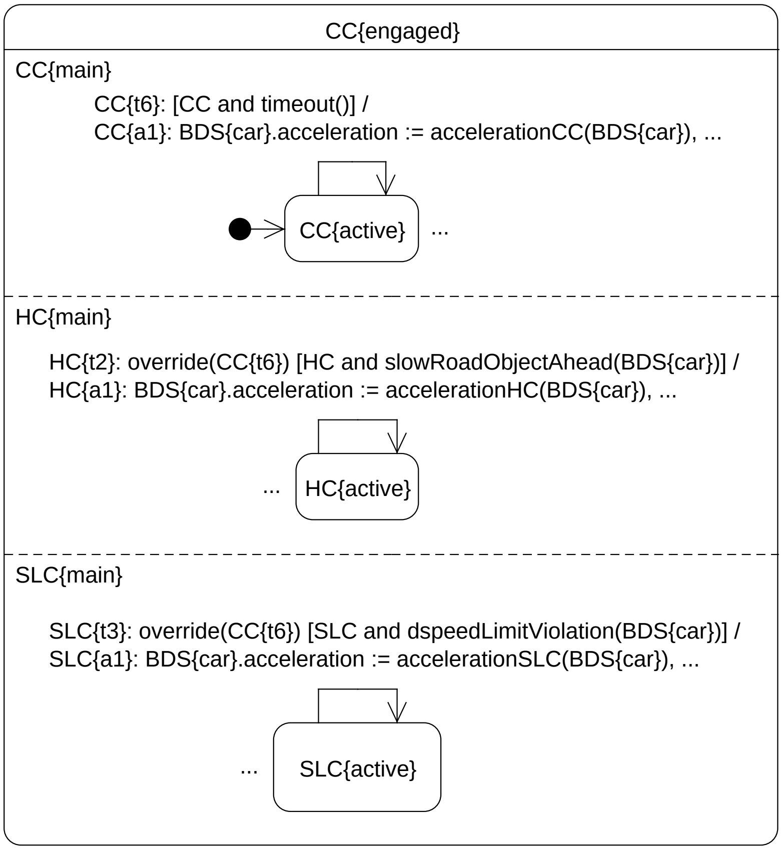

CC{engaged}
CC{main}
CC{t6}: [CC and timeout()] /
CC{a1}: BDS{car}.acceleration := accelerationCC(BDS{car}), ...
CC{active}
...
HC{main}
HC{t2}: override(CC{t6}) [HC and slowRoadObjectAhead(BDS{car})] /
HC{a1}: BDS{car}.acceleration := accelerationHC(BDS{car}), ...
...
HC{active}
SLC{main}
SLC{t3}: override(CC{t6}) [SLC and dspeedLimitViolation(BDS{car})] /
SLC{a1}: BDS{car}.acceleration := accelerationSLC(BDS{car}), ...
...
SLC{active}

Figure 4.2: Example of conflicting actions

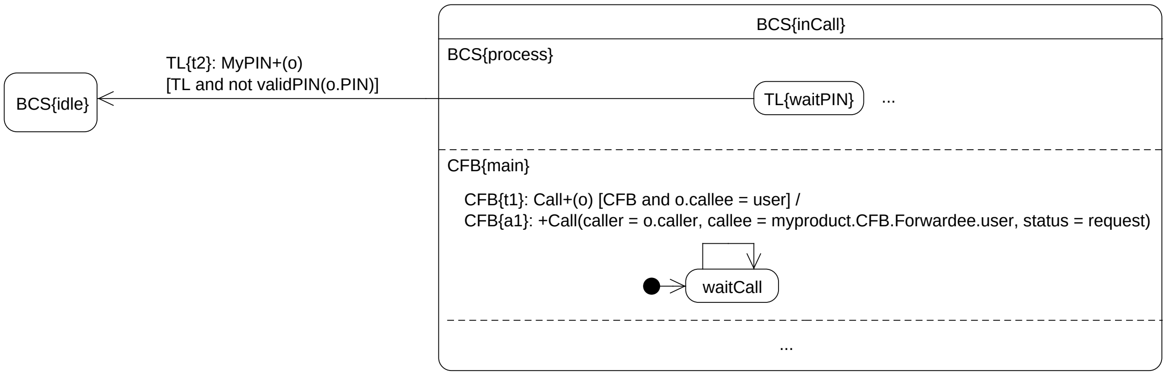

BCS{inCall}
BCS{process}
TL{t2}: MyPIN+(o)
[TL and not validPIN(o.PIN)]
BCS{idle}
TL{waitPIN}
...
CFB{main}
CFB{t1}: Call+(o) [CFB and o.callee = user] /
CFB{a1}: +Call(caller = o.caller, callee = myproduct.CFB.Forwardee.user, status = request)
waitCall
...

Figure 4.3: Example of conflicting destination states

4.2.1 Formal Definition

This section first revises the definition of the state-transition system $\llbracket m \rrbracket$, introduced in Section 3.6, to account for conflicts. Then, conflict interactions are formally defined in the context of the revised definition.

Revised semantics: To account for conflicts, we must revise the definition of $\llbracket m \rrbracket$ in the following ways: First, we formally represent a conflict as the special state $\mathbb{conflict} \in \mathbb{S}$, where $\mathbb{S}$ is the set of states of $\llbracket m \rrbracket$. An $\llbracket m \rrbracket$ transition that results in a conflict is of the form

$$\mathbb{s} \xrightarrow{\mathbb{l}} \mathbb{conflict}$$

where $\mathbb{s}$ and $\mathbb{l}$ are the source and label[3] of the $\llbracket m \rrbracket$ transition, respectively. Next, we revise the function for computing the destination state of an $\llbracket m \rrbracket$ transition to include the case in which an $\llbracket m \rrbracket$ transition results in a conflict. Consider an $\llbracket m \rrbracket$ transition with source state $\mathbb{s} = \boldsymbol{(mode, ws_c, ws_p, ws_{ci})}$ and label $\mathbb{l} = \boldsymbol{(T, A)}$.

Definition 4.2.1. *An $\llbracket m \rrbracket$ transition results in a conflict if for some machine $\boldsymbol{smi}$ of type sm in $\boldsymbol{mode}$, the subset of $\boldsymbol{smi}$'s transitions in $\boldsymbol{T}$ have conflicting destination states, such that executing these transitions does not result in a possible configuration of $\boldsymbol{smi}$:*

$$nextConfig(\boldsymbol{mode}, \boldsymbol{T}, \boldsymbol{smi}) \notin Config_{sm}$$

where $Config_{sm}$ is the set of possible configurations of sm; and $nextConfig(\boldsymbol{mode}, \boldsymbol{T}, \boldsymbol{smi})$ denotes the new configuration of $\boldsymbol{smi}$, after executing $\boldsymbol{smi}$'s transitions in $\boldsymbol{T}$, starting from its configuration in $\boldsymbol{mode}$ (Definition 3.6.20).

Definition 4.2.2. *An $\llbracket m \rrbracket$ transition results in a conflict if the actions $\boldsymbol{A}$ are in conflict, such that a next world state cannot be computed:*

- *If the $\llbracket m \rrbracket$ transition executes as a simple big step (i.e., $\mathbb{s}$ is stable, and executing $\boldsymbol{T}$ results in stable configurations for all of the machine in $\boldsymbol{mode}$), then the actions $\boldsymbol{A}$ conflict if there is no valid world-state transition from $\boldsymbol{ws_c}$ that leads to a next observable world state in which the actions $\boldsymbol{A}$ are satisfied:*

  $$nextWS_{obs}(\mathbb{s}, \boldsymbol{A}) = \emptyset$$

  where $nextWS_{obs}(\mathbb{s}, \boldsymbol{A})$ denotes the set of possible next observable world states (Definition 3.6.21).

[3] The computation of the label $\mathbb{l}$ is assumed to have taken nondeterminism into account, as described in Section 4.1.

- *If the* $[\![m]\!]$ *transition executes as an intermediate step within a compound big step (i.e., executing* $\boldsymbol{T}$ *would result in unstable configurations for some machine in* $\boldsymbol{mode}$*), then the actions* $\boldsymbol{A}$ *conflict if there is no (valid or invalid) world-state transition from* $\boldsymbol{ws_{ci}}$ *that leads to a next (intermediate) world state in which the actions* $\boldsymbol{A}$ *are satisfied:*

  $$nextWS_{int}(\mathbb{s}, \boldsymbol{A}) = \emptyset$$

  where $nextWS_{int}(\mathbb{s}, \boldsymbol{A})$ *denotes the set of possible next intermediate world states (Definition 3.6.24).*

- *If the* $[\![m]\!]$ *transition executes as the last small step in a compound big step (i.e.,* $\mathbb{s}$ *is unstable, and executing* $\boldsymbol{T}$ *would result in stable configurations for all of the machines in* $\boldsymbol{mode}$*), then the actions* $\boldsymbol{A}$ *conflict if either (1) there is no (valid or invalid) world-state transition from* $\boldsymbol{ws_{ci}}$ *that leads to a next (temporary) world state* $\boldsymbol{ws_n}$ *(computed in the same way as the next intermediate world state) in which the actions* $\boldsymbol{A}$ *are satisfied:*

  $$nextWS_{int}(\mathbb{s}, \boldsymbol{A}) = \emptyset$$

  or (2) there is no valid world-state transition from $\boldsymbol{ws_c}$ *that leads to a next observable world state in which the implied set of actions* $\boldsymbol{A'}$ *computed for the big step (see Definition 3.4.11) are satisfied:*

  $$nextWS_{obs}(\mathbb{s}, \boldsymbol{A'}) = \emptyset$$

An execution of $[\![m]\!]$ ends when it reaches the $\mathbb{conflict}$ state. To reflect this behaviour, we stipulate that the $\mathbb{conflict}$ state cannot be the source of an $[\![m]\!]$ transition:

$$\mathbb{T}^{\mathbb{c}} \subseteq (\mathbb{S} - \{\mathbb{conflict}\}) \times \mathbb{L} \times \mathbb{S}$$

where $\mathbb{T}^{\mathbb{c}}$ denotes $[\![m]\!]$'s transition relation with conflicts taken into account, and $\mathbb{S}$ and $\mathbb{L}$ denote $[\![m]\!]$'s set of states and possible labels, respectively (Definition 3.6.1).

Next, we define conflict interactions starting with the special cases and concluding with the general case. Consider an $[\![m]\!]$ transition

$$\mathbb{s} \xrightarrow{\mathbb{l}} \mathbb{conflict}$$

with source state $\mathbb{s} = (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$ and label $\mathbb{l} = (\boldsymbol{T}, \boldsymbol{A})$.

Definition 4.2.3. *A conflicting-destination-states interaction occurs in an* $[\![m]\!]$ *transition if for some machine* $\boldsymbol{smi}$ *in* $\boldsymbol{mode}$*, there exists a minimal subset* $\boldsymbol{T_c}$ *of smi's transitions* $\boldsymbol{T(smi)}$*, such that*

- *The transitions in* $\boldsymbol{T_c}$ *have conflicting destination states (see Definition 4.2.1).*
- *Removing any transition from* $\boldsymbol{T_c}$ *results in a subset that no longer conflicts.*
- *The transitions in* $\boldsymbol{T_c}$ *belong to a set* F *of two or more features.*

In this case, we say that there is a conflict interaction between the features in F. The minimality condition avoids reporting spurious conflict interactions in cases where the transitions in $\boldsymbol{T(smi)}$ conflict and belong to multiple features, but the core transitions that cause the conflict belong to a single feature. Note that there may be multiple minimal conflict subsets of $\boldsymbol{T(smi)}$, each leading to a different conflict interaction.

Definition 4.2.4. *A conflicting-actions interaction occurs in an* $[\![m]\!]$ *transition if there exists a minimal subset* $\boldsymbol{A_c} \subseteq \boldsymbol{A}$*, such that*

- *The actions in* $\boldsymbol{A_c}$ *conflict (see Definition 4.2.2).*
- *Removing any action from* $\boldsymbol{A_c}$ *results in a subset that no longer conflicts.*
- *The actions in* $\boldsymbol{A_c}$ *belong to a set* F *of two or more features.*

In this case, we say that there is a conflict interaction between the features in F[4].

Definition 4.2.5. *A conflict interaction occurs in an* $[\![m]\!]$ *transition if either a conflicting-destination-states (see Definition 4.2.3) or a conflicting-actions (see Definition 4.2.2) interaction occurs in the* $[\![m]\!]$ *transition.*

In an alternative approach taken by Juarez-Dominguez et al. [52], conflicts are defined as inconsistent *requests* to change some value (e.g., a vehicle's throttle) as opposed to inconsistent changes to the value. In their approach, conflicting requests are modelled as distinct output events with conflicting parameter values; hence, conflicts do not result in invalid (conflict) states as with our approach. Thus, their approach enables analysis that can detect multiple conflicts along an execution path.

[4] Note that the notion of feature interactions is not defined for the case of conflicts among the implied set of actions computed for a compound big step (see Definition 3.4.11) because implied actions cannot be associated with features.

4.3 Modification Interactions

An important type of feature interaction arises because new features are generally *non-monotonic* extensions of products [92]: that is, a new feature added to a product p can modify the behaviours of p's existing features, or even the features of other products that operate in p's environment. This can occur, for example, if a new feature y's behaviours create conditions that trigger or prohibit the behaviours of an existing feature x (e.g., lane change control (LXC) triggers lane change alert (LCA) by changing the vehicle's lane). When y modifies the behaviours of x, we say that y has a *modification interaction* with x. Note that modification interactions are defined for pairs of features; in contrast, nondeterminism and conflict interactions are defined for arbitrary sets of features. Furthermore, nondeterminism and conflict interactions are generally unintended[5], whereas modification interactions can be both intended and unintended. Intended modification interactions are explicitly modelled in FORML, using the constructs described in Section 3.4.6.

In Section 4.3.2, we revise the formal definition of modification interactions – given in Section 4.3.1 – to exclude intended cases: intended cases are distinguished by the use of FORML constructs for expressing intended interactions. Distinguishing intended interactions enables feature-interaction analyses that report only unintended interactions. For completeness, Section 4.3.3 defines two special cases of modification interactions, namely deadlock and looping interactions, that are prominent in the literature [70, 20, 56].

Throughout this section, we assume that FORML models may be nondeterministic and have conflicts, as described in Definitions 4.1.1 to 4.2.5. Modification interactions manifest in FORML models as follows. Consider a particular product configuration P (Definition 3.2.4), comprising a particular set of products each with a particular feature configuration, operating in a shared environment. Suppose that a feature y is added to some product p in P. Note that adding y to p may require adding to p a set of other features that y depends on to ensure that p's feature configuration remains valid; the result is product p' and the extended product configuration P'. To determine whether y has a modification interaction with an existing feature x in p' or in some other product in P', we compare the executions of products in P and P'. The set of features that y depends on is not always unique; hence, there may be multiple product configuations P' to compare against.

Definition 4.3.1. *Let P be a product configuration that includes feature x and let P' be the product configuration that results from adding to P feature y and a complete set F of*

[5] At the requirements level, nondeterminism may be desirable for deferring design decisions, but is generally undesirable when specifying behavioural requirements, especially for safety-critical systems.

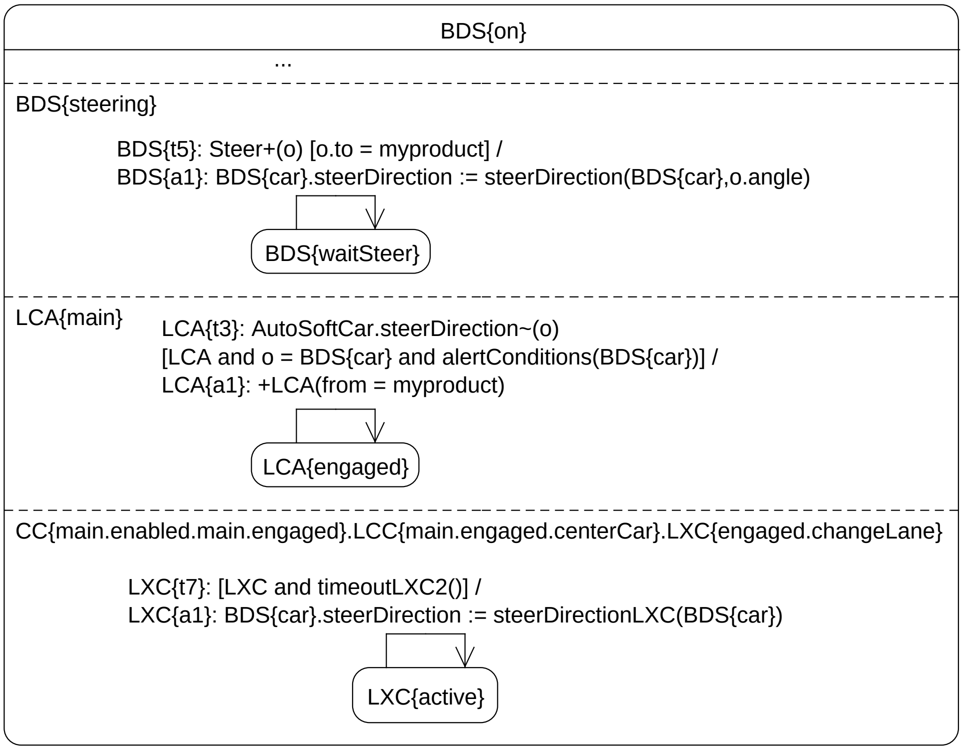

BDS{on}
...
BDS{steering}
BDS{t5}: Steer+(o) [o.to = myproduct] /
BDS{a1}: BDS{car}.steerDirection := steerDirection(BDS{car},o.angle)
BDS{waitSteer}
LCA{main}
LCA{t3}: AutoSoftCar.steerDirection~(o)
[LCA and o = BDS{car} and alertConditions(BDS{car})] /
LCA{a1}: +LCA(from = myproduct)
LCA{engaged}
CC{main.enabled.main.engaged}.LCC{main.engaged.centerCar}.LXC{engaged.changeLane}
LXC{t7}: [LXC and timeoutLXC2()] /
LXC{a1}: BDS{car}.steerDirection := steerDirectionLXC(BDS{car})
LXC{active}

Figure 4.4: Example of a modification interaction

features that y depends on (complete in the sense that excluding any feature from F results in an invalid product configuration P'). Informally, y is said to have a modification interaction with x in P' if the set of possible x transitions and actions differ in corresponding points in the executions of P and P'.

A formal definition of this manifestation is given below in Section 4.3.1.

Example: Figure 4.4 shows a fragment of the composed *AutoSoft* model, which includes the following transitions: transition $BDS\{t5\}$ of the basic driving service (BDS), which changes the vehicle's steering direction based on a steer command from the driver; transition $LCA\{t3\}$ of LCA, which alerts the driver if a change in the vehicle's steering direction results in a lane change; and transition $LXC\{t7\}$ of LXC, which changes the vehicle's steering direction in response to a lane-change request from the driver. In a vehicle that has BDS and LCA, the steering direction can be changed only by feature BDS and, therefore, transition $LCA\{t3\}$ is always triggered only by the actions of transition $BDS\{t5\}$. Now suppose that we add to the vehicle the feature LXC and the features that it depends on (i.e., CC and LCC): now, the vehicle's steering direction can additionally be changed by the feature LXC; and the transition $LCA\{t3\}$ can be triggered by the actions of either $LXC\{t7\}$ or $BDS\{t5\}$. In other words, LCA's transition $t3$ is *triggered* by LXC. Thus, feature LXC has a modification interaction with feature LCA.

As another example, Figure 4.5 shows a fragment of the composed *TelSoft* model, which includes the following elements:

- The transition $BCS\{t5\}$ of the basic call service (BCS) connects one incoming call.
- The transition $BCS\{t6\}$ of BCS rejects all other simultaneous incoming calls.
- The transition $VM\{t1\}$ of the voice mail (VM) feature connects a caller to the voicemail service of the callee, if a call request is not answered after some timeout period.
- The strengthening clause $TCS\{s1\}$ of TCS (intentionally) prohibits the connection of an incoming call, if the caller is on a screening list associated with TCS.

In a telephone service that includes features BCS and VM, transitions $BCS\{t5\}$ and $BCS\{t6\}$ execute in sequence when an idle subscriber receives a call; if the subscriber does not answer the call in time, transition $VM\{t1\}$ will execute. However, in a telephone service that includes all three features BCS, VM, and TCS, the sequence of transitions $BCS\{t5\}$, $BCS\{t6\}$, and $VM\{t1\}$ do not execute when a call request comes from a caller on TCS's screening list, because the strengthening clause $TCS\{s1\}$ disables the first transition in the sequence. In other words, BCS's and VM's transitions are *prohibited* by TCS. Thus, feature TCS has a modification interaction with features BCS and VM. □

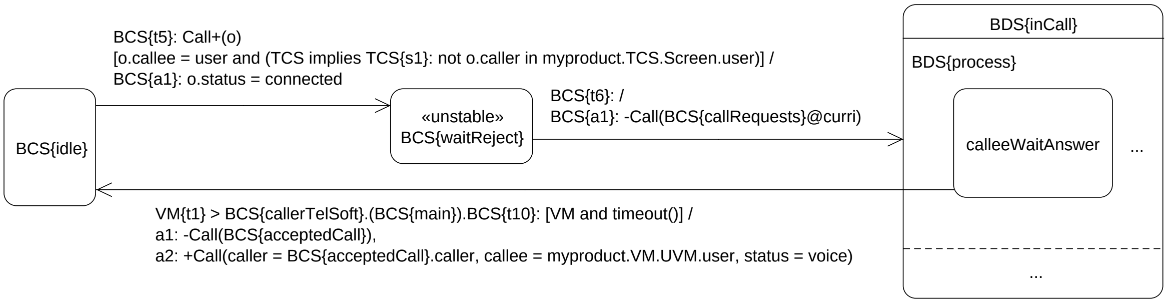

BCS{t5}: Call+(o)
[o.callee = user and (TCS implies TCS{s1}: not o.caller in myproduct.TCS.Screen.user)] /
BCS{a1}: o.status = connected
BCS{idle}
«unstable»
BCS{waitReject}
BCS{t6}: /
BCS{a1}: -Call(BCS{callRequests}@curri)
BDS{inCall}
BDS{process}
calleeWaitAnswer
...
...
VM{t1} > BCS{callerTelSoft}.(BCS{main}).BCS{t10}: [VM and timeout()] /
a1: -Call(BCS{acceptedCall}),
a2: +Call(caller = BCS{acceptedCall}.caller, callee = myproduct.VM.UVM.user, status = voice)

Figure 4.5: Example of a modification interaction due to prohibiting behaviour

4.3.1 Formal Definition

To formalize the notion of a modification interaction, we first formally define machine executions that correspond to a particular product configuration. In the following, a product configuration is formally represented by a set of tuples of the form (p, F), one tuple for each product p, where F is p's set of features (Definition 3.2.4). For example

$$\{\ (\boldsymbol{p1}, \{\boldsymbol{f1}\}), (\boldsymbol{p2}, \{\boldsymbol{f2}, \boldsymbol{f3}\})\ \}$$

represents a product configuration comprising a product $p1$ with a single feature $f1$ and product $p2$ with the two features $f2$ and $f3$. The notation $\pi^J(K)$ is used to denote a projection over a concept K based on a projection criterion J; informally, the projection represents a part of K that is selected based on criterion J.

Definition 4.3.2. *Let $[\![m]\!] = (\mathbb{S}, \mathbb{S}_0, \mathbb{L}, \mathbb{T})$ be the state-transition system for a FORML model m. Projection $\pi^P([\![m]\!])$ denotes a state-transition system whose executions correspond to product configuration P: it is the subset of $[\![m]\!]$ executions whose initial states have product configuration P:*

$$\pi^P([\![m]\!]) = (\mathbb{S}, \pi^P(\mathbb{S}_0), \mathbb{L}, \mathbb{T})$$

where $\pi^P(\mathbb{S}_0)$ denotes the subset of $[\![m]\!]$'s initial states that correspond to P[6]*:*

$$\pi^P(\mathbb{S}_0) = \{\ (\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}}) \in \mathbb{S}_0 \mid PC(\boldsymbol{ws_c}) = P\ \}$$

where $PC(\boldsymbol{ws})$ denotes the product configuration of world state $\boldsymbol{ws}$ (Definition 3.2.4).

[6] Recall from Section 3.6.2 that in an initial state $(\boldsymbol{mode}, \boldsymbol{ws_c}, \boldsymbol{ws_p}, \boldsymbol{ws_{ci}})$ of $[\![m]\!]$, $\boldsymbol{ws_c} = \boldsymbol{ws_p} = \boldsymbol{ws_{ci}}$.

Feature interaction: Let $P + y$ denote the product configuration that results from adding feature y and a complete set F of additional features that y depends on to an arbitrary product in an arbitrary product configuration P[7].

To determine whether y has a modification interaction with a feature x in $P + y$, we compare the executions of $\pi^P(\llbracket m \rrbracket)$ with those of $\pi^{P+y}(\llbracket m \rrbracket)$ with respect to the transitions and actions of x. The comparison is formally expressed as a *bisimilarity* check between $\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ over x. Bisimilarity is a widely used notion of behavioural equivalence for state-transition systems [74]. The following definition of bisimilarity between $\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ over x is an adaptation of Lee and Seshia's definition of strong bisimilarity [64]. We make two adaptations to Lee and Sehsia's definition: (1) we weaken their definition to compare $\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ only with respect to the transitions and actions of x; and (2) we adjust their definition, which assumes that a state-transition system has a single initial state, to account for the multiple initial states of $\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$.

In the following, we use the superscripts P and $P + y$ to distinguish the elements of $\pi^P(\llbracket m \rrbracket)$ from the elements of $\pi^{P+y}(\llbracket m \rrbracket)$. Hence, we assume

$$\pi^P(\llbracket m \rrbracket) = (\mathbb{S}^P, \mathbb{S}_0^P, \mathbb{L}^P, \mathbb{T}^P), \text{ and}$$
$$\pi^{P+y}(\llbracket m \rrbracket) = (\mathbb{S}^{P+y}, \mathbb{S}_0^{P+y}, \mathbb{L}^{P+y}, \mathbb{T}^{P+y})$$

We use the same convention to refer to individual states and transition labels in the two transition systems (e.g., $\mathbb{s}_i^P$ denotes the ith state in an execution of $\pi^P(\llbracket m \rrbracket)$).

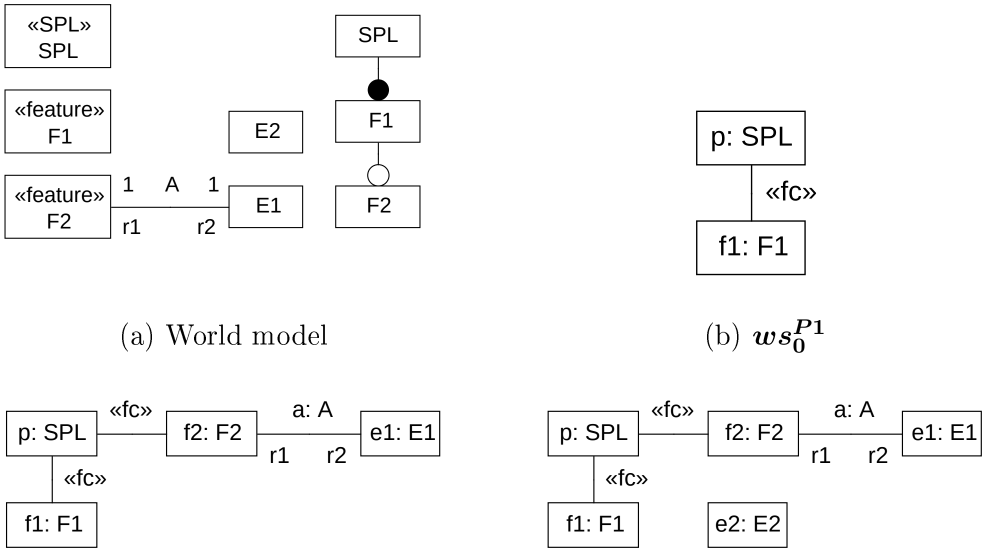
Bisimilarity Whether $\pi^P(\llbracket m \rrbracket)$ is bisimilar to $\pi^{P+y}(\llbracket m \rrbracket)$ over x is determined by playing a set of *matching games*. A game is played for each pair of *corresponding* initial states that the two transition systems can start in.

Definition 4.3.3. *Let*

$$\mathbb{s}_0^P = (\boldsymbol{mode_0^P}, \boldsymbol{ws_0^P}, \boldsymbol{ws_0^P}, \boldsymbol{ws_0^P}) \in \mathbb{S}_0^P, \textit{ and}$$
$$\mathbb{s}_0^{P+y} = (\boldsymbol{mode_0^{P+y}}, \boldsymbol{ws_0^{P+y}}, \boldsymbol{ws_0^{P+y}}, \boldsymbol{ws_0^{P+y}}) \in \mathbb{S}_0^{P+y}$$

$\mathbb{s}_0^P$ corresponds to $\mathbb{s}_0^{P+y}$, denoted correspond$(\mathbb{s}_0^P, \mathbb{s}_0^{P+y})$, if and only if (1) the nodes $\boldsymbol{mode_0^P}$ and $\boldsymbol{mode_0^{P+y}}$ are identical, and (2) the differences between the world states $\boldsymbol{ws_0^P}$ and $\boldsymbol{ws_0^{P+y}}$ are minimal: that is, the only differences are that $\boldsymbol{ws_0^{P+y}}$ includes feature y and any other feature that y depends on, and includes world objects that y depends on (as per the world-model constraints).

[7] The set of additional features that y depends on is not always unique.



(a) World model

(b) $\boldsymbol{ws_0^{P1}}$

(c) $\boldsymbol{ws_0^{P1+f2}}$

(d) $\boldsymbol{ws_0'^{P1+f2}}$

Figure 4.6: Example of corresponding initial states

Example: As an example of the notion of minimally different world states, consider the world model and world states shown in Figure 4.6. The world state $\boldsymbol{ws_0^{P1}}$ (Figure 4.6b) has a product configuration $P1$ comprising a product p with a feature $f1$; and the world states $\boldsymbol{ws_0^{P1+f2}}$ (Figure 4.6c) and $\boldsymbol{ws_0'^{P1+f2}}$ (Figure 4.6c) each have a product configuration $P1 + f2$ obtained by adding feature $f2$ to product p in $P1$. Besides the added feature $f2$, $\boldsymbol{ws_0^{P1+f2}}$'s only difference from $\boldsymbol{ws_0^{P1}}$ is the link a between $f2$ and the object $e1$; this difference is required by the one-to-one association A between $F2$ and $E1$. Hence, $\boldsymbol{ws_0^{P1+f2}}$ is minimally different from $\boldsymbol{ws_0^{P1}}$. On the other hand, $\boldsymbol{ws_0'^{P1+f2}}$ is additionally different from $\boldsymbol{ws_0^{P1}}$ in the presence of the new object $e2$. This new object of type $E2$ is not strictly required by the world-model constraints. Hence, $\boldsymbol{ws_0'^{P1+f2}}$ is not minimally different from $\boldsymbol{ws_0^{P1}}$. □

In each round of a game, either transition system can take a semantic transition representing a big step. In the following, a semantic transition taken in a round of a matching game is called a *move*. For all possible next moves in either system, there must exist a *matching move* – with respect to x – in the other system.

Definition 4.3.4. *Let* $\mathbb{s}^P \xrightarrow{\mathbb{l}^P} \mathbb{s}'^P$ *and* $\mathbb{s}^{P+y} \xrightarrow{\mathbb{l}^{P+y}} \mathbb{s}'^{P+y}$ *be semantic transitions (moves) of*

$\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$, respectively. The moves match with respect to x if

$$\pi^x(\mathbb{l}^P) = \pi^x(\mathbb{l}^{P+y})$$

where $\pi^x(\mathbb{l})$ is a projection comprising the transitions and actions in a label $\mathbb{l} = (\boldsymbol{T}, \boldsymbol{A})$ that belong to feature x:

$$\pi^x(\mathbb{l}) = (\pi^x(\boldsymbol{T}), \pi^x(\boldsymbol{A}))$$

where

$$\begin{aligned} \pi^x(\boldsymbol{T}) &= \{\boldsymbol{t} \in \boldsymbol{T} \mid \boldsymbol{t} \textit{ belongs to } x\} \textit{ and} \\ \pi^x(\boldsymbol{A}) &= \{\boldsymbol{a} \in \boldsymbol{A} \mid \boldsymbol{a} \textit{ belongs to } x\} \end{aligned}$$

A matching game is a *winning game* if the two transition systems can match each other's moves in all rounds of the matching game. $\pi^P(\llbracket m \rrbracket)$ is bisimilar to $\pi^{P+y}(\llbracket m \rrbracket)$ over x if all possible matching games between the two transition systems (i.e., one game for each pair of corresponding initial states of the two transition systems) are winning games. A winning game can be succinctly specified in terms of the pairs of states in which the two transition systems reside, in each round of the game. Using such a specification, we can provide a more formal definition for bisimulation:

Definition 4.3.5. *$\pi^P(\llbracket m \rrbracket)$ is bisimilar to $\pi^{P+y}(\llbracket m \rrbracket)$ over x if and only if there exists a bisimulation relation $BSR \subseteq \mathbb{S}^P \times \mathbb{S}^{P+y}$ that specifies a winning game for every possible start to a game:*

1. *Every pair of corresponding initial states of the two transition systems is a pair in BSR (i.e., the set of possible starting points):*

$$\{\ (\mathbb{s}_0^P, \mathbb{s}_0^{P+y}) \in \mathbb{S}_0^P \times \mathbb{S}_0^{P+y} \mid correspond(\mathbb{s}_0^P, \mathbb{s}_0^{P+y})\ \} \subseteq BSR$$

2. *Every move of $\pi^P(\llbracket m \rrbracket)$ is matched by a move of $\pi^{P+y}(\llbracket m \rrbracket)$ with respect to x:*

   if $(\mathbb{s}^P, \mathbb{s}^{P+y}) \in BSR$, then

   for all $\mathbb{s}^P \xrightarrow{\mathbb{l}^P} \mathbb{s}'^P \in \mathbb{T}^P$:

   there is a $\mathbb{s}^{P+y} \xrightarrow{\mathbb{l}^{P+y}} \mathbb{s}'^{P+y} \in \mathbb{T}^{P+y}$ such that:

   (a) $\pi^x(\mathbb{l}^P) = \pi^x(\mathbb{l}^{P+y})$, and

(b) $(\mathbb{s}'^{P}, \mathbb{s}'^{P+y}) \in BSR$

3. *Every move of* $\pi^{P+y}([\![m]\!])$ *is matched by a move of* $\pi^{P}([\![m]\!])$ *with respect to* x*:*

   if $(\mathbb{s}^{P}, \mathbb{s}^{P+y}) \in BSR$*, then*

   for all $\mathbb{s}^{P+y} \xrightarrow{\mathbb{l}^{P+y}} \mathbb{s}'^{P+y} \in \mathbb{T}^{P+y}$*:*

   there is a $\mathbb{s}^{P} \xrightarrow{\mathbb{l}^{P}} \mathbb{s}'^{P} \in \mathbb{T}^{P}$ *such that:*

   (a) $\pi^{x}(\mathbb{l}^{P}) = \pi^{x}(\mathbb{l}^{P+y})$*, and*

   (b) $(\mathbb{s}'^{P}, \mathbb{s}'^{P+y}) \in BSR$

4.3.2 Distinguishing Intended Interactions

In FORML, a feature y's intended (modification) interactions with another feature x can be modelled explicitly by the following constructs (as described in Section 3.4.6.2):

- *Preemptive transitions and actions*: feature y introduces a preemptive transition (or action) that preempts specific transitions (or actions) of feature x.

- *Weakening and strengthening clauses*: feature y introduces a weakening or strengthening clause in the guard condition of one of feature x's transitions or actions, which causes that transition or action to execute under more or fewer conditions, respectively.

- *Fragments that trigger or prohibit state changes*: feature y intentionally triggers or prohibits the entry or exit of a state sx introduced by x causing a state change (Definition 3.4.14). Specifically, if some behaviour of y intentionally triggers a transition that enters (or exits) state sx, then the transitions and actions of x whose enabledness depends on state sx will execute under more (or fewer) conditions. Conversely, if some behaviour of y intentionally prohibits a transition that enters (or exits) state sx, then the transitions and actions of sx whose enabledness depends on state sx will execute under fewer (or more) conditions.

The general definition of a modification interaction given in Section 4.3 includes all types of modification interactions, including those caused by the above constructs for modelling intended interactions. In the following, we revise the definition to exclude intended interactions, so as to support analyses that report only unintended interactions.

Example: Recall that TCS has a modification interaction with BCS, as shown in Figure 4.5. Ideally, this interaction would not be reported because it is intended: Transition $BCS\{t5\}$ of BCS does not execute because of TCS's strengthening clause $TCS\{s1\}$. Furthermore, the preemption of $BCS\{t5\}$ prohibits the entry of state $BCS\{waitReject\}$ of BCS, which prevents transition $BCS\{t6\}$ of BCS from executing. □

4.3.2.1 Revised Modification Interactions

Suppose that a feature y is added to a product configuration P, resulting in the product configuration $P + y$. Recall that in the general definition given in Section 4.3, y has a modification interaction with a feature x in P if the projections $\pi^{P}(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ of $\llbracket m \rrbracket$ cannot match one another's moves (transitions) in all rounds of all of the possible matching games that they can play. In the general definition, respective moves in the two systems are considered to be a match if they perform the same transitions and actions of x. However, two respective moves may fail to match by design because there are intended interactions in y that trigger or prohibit the transitions and actions of x in the current or future moves of $\pi^{P+y}(\llbracket m \rrbracket)$:

- A transition xt or action xa of x does not execute in the current $\pi^{P+y}(\llbracket m \rrbracket)$ move because of a preemptive transition, a preemptive action, or strengthening clause of y. In this case, the transition xt or action xa does not appear in the label of the $\pi^{P+y}(\llbracket m \rrbracket)$ move, but does appear in a respective $\pi^{P}(\llbracket m \rrbracket)$ move. Analogously, a transition xt or action xa of x executes in the current $\pi^{P+y}(\llbracket m \rrbracket)$ move only because of a weakening clause of y. In this second case, the transition xt or action xa appears in the label of the $\pi^{P+y}(\llbracket m \rrbracket)$ move but not in the label of any respective $\pi^{P}(\llbracket m \rrbracket)$ move. In order to ignore such intended interactions, we weaken the notion of a match between moves to allow differences caused by preemptive transitions and actions, and by weakening and strengthening clauses.

- A transition xt or action xa of x does not execute in a future $\pi^{P+y}(\llbracket m \rrbracket)$ move, because of a transition or clause of y in the current $\pi^{P+y}(\llbracket m \rrbracket)$ move that intentionally triggers the exit or prohibits the entry of an x state (causing a state change) on which xt's or xa's enabledness depends. If the y transition or clause intentionally triggers the exit or prohibits the entry of the x state, then transition xt or action xa may appear in the label of a $\pi^{P}(\llbracket m \rrbracket)$ move, but not appear in the label of any respective $\pi^{P+y}(\llbracket m \rrbracket)$ move. Analogously, if the y transition or clause intentionally triggers the entry or prohibits the exit of the x state, the transition xt or action xa may appear in the label of a future $\pi^{P+y}(\llbracket m \rrbracket)$ move but not in the label of any respective $\pi^{P}(\llbracket m \rrbracket)$ move.

> In order to ignore such intended interactions, we weaken the notion of a winning game as follows: if in the course of a matching game $\pi^{P+y}(\llbracket m \rrbracket)$ performs a move that intentionally triggers or prohibits the entry or exit of an x state (causing a state change) via a y transition or clause (as described above), any future mismatches between the moves of $\pi^{P}(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ are tolerated.

The weakened notion of a winning game correctly excludes *intended* future mismatches between the moves of $\pi^{P}(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$: that is, future mismatches that are initiated by an intended entry or exit of an x state in the current $\pi^{P+y}(\llbracket m \rrbracket)$ move. However, it may also exclude *unintended* future mismatches, which are not caused by the state change. The unintended future mismatches might be caught in other moves of the matching game in which the state change does not occur in the current $\pi^{P+y}(\llbracket m \rrbracket)$ move. For example, suppose that the current $\pi^{P+y}(\llbracket m \rrbracket)$ move includes two y transitions ty_1 and ty_2, where ty_1 intentionally triggers the entry of an x state (causing a state change) and ty_2 performs a WCA that unintentionally disables an x transtion tx in the next $\pi^{P+y}(\llbracket m \rrbracket)$ move. In this $\pi^{P+y}(\llbracket m \rrbracket)$ move, the game is won because of ty_1; thus, the unintended disabling of tx in the next $\pi^{P+y}(\llbracket m \rrbracket)$ move is not reported. However, if there exists an alternative current $\pi^{P+y}(\llbracket m \rrbracket)$ move that includes ty_2 and does not include ty_1, the game is not won and the unintended disabling of tx in the next $\pi^{P+y}(\llbracket m \rrbracket)$ move is reported.

Definition 4.3.6 below, describes the weakened notion of a match, called an *intended match*, in more detail. Following that, Definition 4.3.7 presents a revised definition of bisimilarity that uses the notion of an intended match and encodes the weakened notion of a winning game.

Definition 4.3.6. *Let $\mathbb{s}^P \xrightarrow{\mathbb{I}^P} \mathbb{s}'^P$ and $\mathbb{s}^{P+y} \xrightarrow{\mathbb{I}^{P+y}} \mathbb{s}'^{P+y}$ be moves of $\pi^{P}(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$, respectively, where $\mathbb{I}^P = \boldsymbol{(T^P, A^P)}$ and $\mathbb{I}^{P+y} = \boldsymbol{(T^{P+y}, A^{P+y})}$. The moves intentionally match with respect to x if and only if*

$$\pi^x(\mathbb{I}^P) \approx \pi^x(\mathbb{I}^{P+y})$$

which denotes the following conditions:

- *Any transition or action of x that is in $\mathbb{I}^P$ but not in $\mathbb{I}^{P+y}$ is either (1) preempted in the move of $\pi^{P+y}(\llbracket m \rrbracket)$ by transitions or actions belonging to y, or (2) disabled in the move of $\pi^{P+y}(\llbracket m \rrbracket)$ because of a strengthening clause introduced by y. More precisely, for each transition $\boldsymbol{t^x} \in \pi^x(\boldsymbol{T^P} - \boldsymbol{T^{P+y}})$ of feature x, one of the following conditions hold:*

1. $\exists\ \boldsymbol{t^y} \in \pi^y(T_{en}(\mathbb{s}^{P+y})) : (\boldsymbol{t^y}, \boldsymbol{t^x}) \in Preempts_{tran}(\mathbb{s}^{P+y})$

   where $\pi^y(T_{en}(\mathbb{s}^{P+y}))$ *is the set of* y *transitions that are enabled in* $\mathbb{s}^{P+y}$*, and* $Preempts_{tran}(\mathbb{s}^{P+y})$ *is the preemption relation between the transitions in* $\mathbb{s}^{P+y}$ *(Definition 3.6.12).*

2. $\boldsymbol{t^x} \notin T_{en}(\mathbb{s}^{P+y})$ *only because* $\boldsymbol{t^x}$*'s guard is false due to a strengthening clause* $\boldsymbol{sc^y}$ *from feature* y*, such that if* $\boldsymbol{sc^y}$ *evaluated to true (or, equivalently, if* $\boldsymbol{sc^y}$ *were omitted from the guard), then* $\boldsymbol{t^x} \in T_{en}(\mathbb{s}^{P+y})$*.*

Analogously, for every action $\boldsymbol{a^x} \in \pi^x(\boldsymbol{A^P} - \boldsymbol{A^{P+y}})$ *of feature* x*, one of the following conditions hold:*

1. $\exists\ \boldsymbol{a^y} \in \pi^y(A_{en}(\mathbb{s}^{P+y})) : (\boldsymbol{a^y}, \boldsymbol{a^x}) \in Preempts_{act}(\mathbb{s}^{P+y})$

   where $\pi^y(A_{en}(\mathbb{s}^{P+y}))$ *is the set of* y *actions that are enabled in* $\mathbb{s}^{P+y}$*, and* $Preempts_{act}(\mathbb{s}^{P+y})$ *is the preemption relation between the actions in* $\mathbb{s}^{P+y}$ *(Definition 3.6.19).*

2. $\boldsymbol{a^x} \notin A_{en}(\mathbb{s}^{P+y})$ *only because* $\boldsymbol{a^x}$*'s guard is false due to a strengthening clause* $\boldsymbol{sc^y}$ *from feature* y*, such that if* $\boldsymbol{sc^y}$ *evaluated to true (or, equivalently, if* $\boldsymbol{sc^y}$ *were omitted from the guard), then* $\boldsymbol{a^x} \in A_{en}(\mathbb{s}^{P+y})$*.*

- *Any transition of* x *that is in* $\mathbb{I}^{P+y}$ *but not in* $\mathbb{I}^P$ *is enabled in the move of* $\pi^{P+y}(\llbracket m \rrbracket)$ *because of a weakening clause of* y*. More precisely, for every* $\boldsymbol{t^x} \in \pi^x(\boldsymbol{T^{P+y}} - \boldsymbol{T^P})$*, the transition is enabled only because it includes a weakening clause* $\boldsymbol{wc^y}$ *from feature* y*, such that if* $\boldsymbol{wc^y}$ *evaluated to false (or, equivalently, if* $\boldsymbol{wc^y}$ *were omitted from the transition's guard), then the transition would not be enabled.*

  Analogously, any action of x *that is in* $\mathbb{I}^{P+y}$ *but not in* $\mathbb{I}^P$ *is enabled in the move of* $\pi^{P+y}(\llbracket m \rrbracket)$ *because of a weakening clause of* y*. More precisely, for every action* $\boldsymbol{a^x} \in \pi^x(\boldsymbol{A^{P+y}} - \boldsymbol{A^P})$*, the guard of* $\boldsymbol{a^x}$ *is true only because it includes a weakening clause* $\boldsymbol{wc^y}$ *from feature* y*, such that if* $\boldsymbol{wc^y}$ *evaluated to false (or, equivalently, if* $\boldsymbol{wc^y}$ *were omitted from the guard), then the guard would be false.*

We now provide a new definition for bisimilarity, which revises clauses 2(a) and 3(a) to use the notion of an intended match between the moves of $\pi^P(\llbracket m \rrbracket)$ and $\pi^{P+y}(\llbracket m \rrbracket)$ as defined above; and revises clauses 2(b) and 3(b) to encode the weakened notion of a matching game described above. The original clauses 2(b) and 3(b) are augmented with a precondition of the form $if\ \neg c\ then$, where c is the condition under which the following moves in a matching game need not be matched for the game to be won. The revised clauses are shown in red.

Definition 4.3.7. *$\pi^P(\llbracket m \rrbracket)$ is intentionally bisimilar to $\pi^{P+y}(\llbracket m \rrbracket)$ over x if and only if there exists a bisimulation relation $BSR \subseteq \mathbb{S}^P \times \mathbb{S}^{P+y}$ such that:*

1. *Every pair of corresponding initial states of the two transition systems is a pair in BSR (i.e., the set of possible starting points):*

$$\{ \ (\mathbb{s}_0^P, \mathbb{s}_0^{P+y}) \in \mathbb{S}_0^P \times \mathbb{S}_0^{P+y} \mid correspond(\mathbb{s}_0^P, \mathbb{s}_0^{P+y}) \ \} \subseteq BSR$$

2. *Every move of $\pi^P(\llbracket m \rrbracket)$ is matched by a move of $\pi^{P+y}(\llbracket m \rrbracket)$ with respect to x:*

   if $(\mathbb{s}^P, \mathbb{s}^{P+y}) \in BSR$, then

   for all $\mathbb{s}^P \xrightarrow{\mathbb{I}^P} \mathbb{s}'^P \in \mathbb{T}^P$:

   there is a $\mathbb{s}^{P+y} \xrightarrow{\mathbb{I}^{P+y}} \mathbb{s}'^{P+y} \in \mathbb{T}^{P+y}$ such that:

   (a) $\pi^x(\mathbb{I}^P) \approx \pi^x(\mathbb{I}^{P+y})$, and

   (b) if $\mathbb{I}^{P+y}$ includes no y transition or clause that intentionally triggers or prohibits the entry or exit of an x state (causing a state change), then $(\mathbb{s}'^P, \mathbb{s}'^{P+y}) \in BSR$

3. *Every move of $\pi^{P+y}(\llbracket m \rrbracket)$ is matched by a move of $\pi^P(\llbracket m \rrbracket)$ with respect to x:*

   if $(\mathbb{s}^P, \mathbb{s}^{P+y}) \in BSR$, then

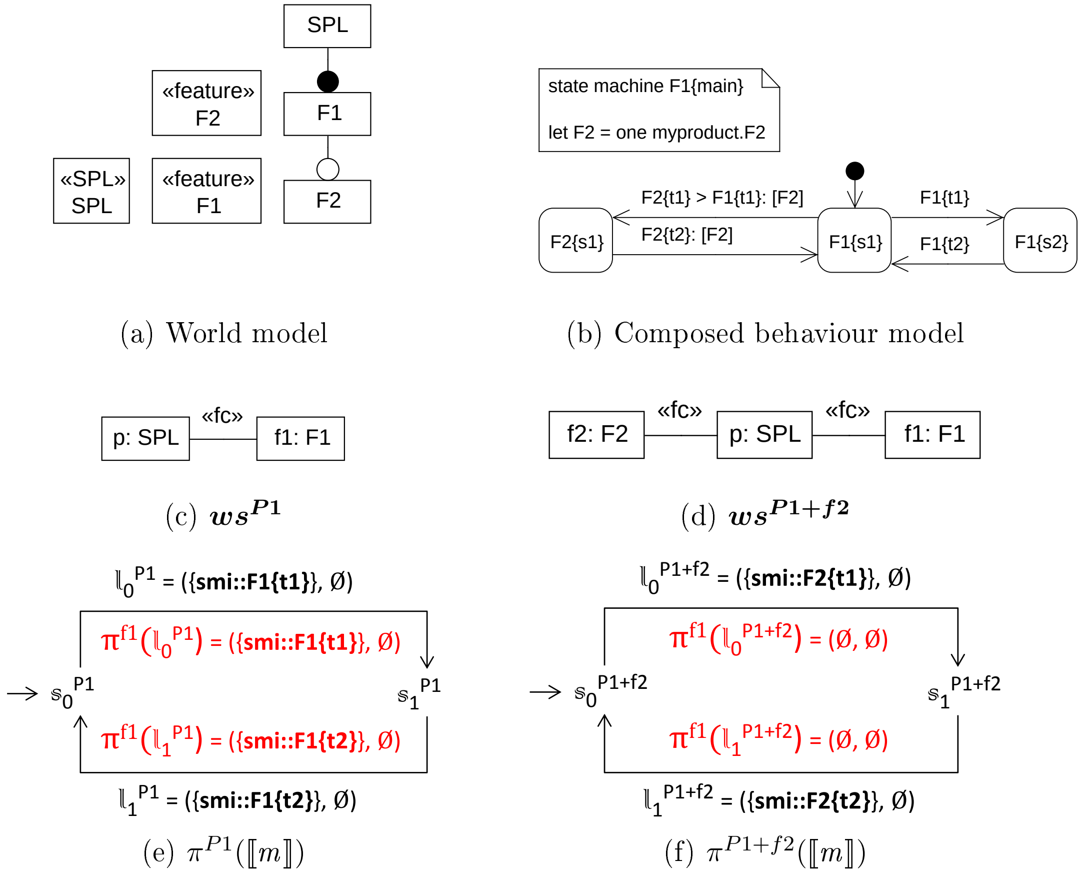
   for all $\mathbb{s}^{P+y} \xrightarrow{\mathbb{I}^{P+y}} \mathbb{s}'^{P+y} \in \mathbb{T}^{P+y}$:

   there is a $\mathbb{s}^P \xrightarrow{\mathbb{I}^P} \mathbb{s}'^P \in \mathbb{T}^P$ such that:

   (a) $\pi^x(\mathbb{I}^P) \approx \pi^x(\mathbb{I}^{P+y})$, and

   (b) if $\mathbb{I}^{P+y}$ includes no y transition or clause that intentionally triggers or prohibits the entry or exit of an x state (causing a state change), then $(\mathbb{s}'^P, \mathbb{s}'^{P+y}) \in BSR$

The revised definition of Definition 4.3.7 is sound in that it excludes all cases of modification interactions caused by FORML's constructs for modelling intended interactions: The revised clauses 2(a) and 3(b) use the notion of an intended match to exclude the effect of feature y's preemptive transitions and actions, and weakening and strengthening clauses, on feature x's transitions and actions in the current move of a game. The revised clauses 2(b) and 3(b) use the weakened notion of a winning game to exclude the effect of feature y's



(a) World model

(b) Composed behaviour model

(c) $\boldsymbol{ws^{P1}}$

(d) $\boldsymbol{ws^{P1+f2}}$

(e) $\pi^{P1}(\llbracket m \rrbracket)$

(f) $\pi^{P1+f2}(\llbracket m \rrbracket)$

Figure 4.7: Example of bisimilarity in the presence of intended interactions

state changes (via transitions or clauses) on x's transitions and actions in the future moves of a game. However, the revised definition is not complete because, as explained above, the weakened notion of a winning game may also exclude some unintended mistmaches in future moves that are not caused by y's state changes.

The following gives a simple pedagogical example of bisimilarity in the presence of intended feature interactions. Let m be the FORML model of an SPL with a mandatory feature $F1$ and an optional feature $F2$. To keep the example simple, the only concepts in m's world model (Figure 4.7a) are the SPL and feature concepts. $F2$ has an intended interaction with $F1$, as modelled by transition $F2\{t2\}$ in m's CBM (Figure 4.7b), which takes priority over transition $F1\{t1\}$. Let $P1$ be a product configuration comprising a product with an $F1$ feature $f1$, and the let $P1+f2$ be the product configuration comprising a product with both an $F1$ feature $f1$ and an $F2$ feature $f2$. We prove that $\pi^{P1}(\llbracket m \rrbracket)$ is bisimilar to $\pi^{P1+f2}(\llbracket m \rrbracket)$ over $f1$ by constructing a bisimulation relation BSR that conforms to Definition 4.3.7.

$\pi^{P1}(\llbracket m \rrbracket)$ and $\pi^{P1+f2}(\llbracket m \rrbracket)$ are shown graphically in Figures 4.7e and 4.7f, respectively. $\pi^{P1}(\llbracket m \rrbracket)$'s states are defined as

$$\begin{aligned} \mathbb{s}_0^{P1} &= (\boldsymbol{mode_{F1\{s1\}}}, \boldsymbol{ws^{P1}}, \boldsymbol{ws^{P1}}, \boldsymbol{ws^{P1}}) \\ \mathbb{s}_1^{P1} &= (\boldsymbol{mode_{F1\{s2\}}}, \boldsymbol{ws^{P1}}, \boldsymbol{ws^{P1}}, \boldsymbol{ws^{P1}}) \end{aligned}$$

where $\boldsymbol{mode_s}$ denotes the mode comprising an instance of state machine $F1\{main\}$ (Figure 4.7b) in configuration s, and $\boldsymbol{ws^{P1}}$ denotes the (only) world state with product configuration $P1$[8] (Figure 4.7c). $\mathbb{s}_0^{P1}$ is the initial state. $\pi^{P1}(\llbracket m \rrbracket)$ transitions from $\mathbb{s}_0^{P1}$ to $\mathbb{s}_1^{P1}$ and vice versa by executing the transitions $F1\{t1\}$ and $F1\{t2\}$ of smi, as specified by the semantic transition labels $\mathbb{l}_0^{P1}$ and $\mathbb{l}_0^{P1+f2}$, respectively. Transition $F2\{t1\}$ is not enabled in $\mathbb{s}_0^{P1}$ because product p has no $F2$ feature. $F1\{t1\}$ and $F1\{t2\}$ have no actions and therefore do not change the world state. Note that because $\pi^{P1}(\llbracket m \rrbracket)$'s transitions perform only $f1$ transitions,

$$\begin{aligned} \pi^{f1}(\mathbb{l}_0^{P1}) &= \mathbb{l}_0^{P1} \\ \pi^{f1}(\mathbb{l}_1^{P1}) &= \mathbb{l}_1^{P1} \end{aligned}$$

Analogously, $\pi^{P1+f2}(\llbracket m \rrbracket)$'s states are defined as

$$\begin{aligned} \mathbb{s}_0^{P1+f2} &= (\boldsymbol{mode_{F1\{s1\}}}, \boldsymbol{ws^{P1+f2}}, \boldsymbol{ws^{P1+f2}}, \boldsymbol{ws^{P1+f2}}) \\ \mathbb{s}_1^{P1+f2} &= (\boldsymbol{mode_{F2\{s1\}}}, \boldsymbol{ws^{P1+f2}}, \boldsymbol{ws^{P1+f2}}, \boldsymbol{ws^{P1+f2}}) \end{aligned}$$

where $\boldsymbol{ws^{P1+f2}}$ denotes the (only) world state with product configuration $P1+f2$ (Figure 4.7d). $\pi^{P1+f2}(\llbracket m \rrbracket)$ transitions from the initial state $\mathbb{s}_0^{P1+f2}$ to state $\mathbb{s}_1^{P1+f2}$ and vice versa by executing the transitions $F2\{t1\}$ and $F2\{t2\}$ of smi, as specified by the semantic transition labels $\mathbb{l}_0^{P1+f2}$ and $\mathbb{l}_0^{P1+f2}$, respectively. Transition $F1\{t1\}$ is enabled in $\mathbb{s}_0^{P1+f2}$, but is preempted by transition $F2\{t1\}$. $F2\{t1\}$ and $F2\{t2\}$ have no actions and therefore do not change the world state. Note that since neither of $\pi^{P1+f2}(\llbracket m \rrbracket)$'s transitions performs any $f1$ transitions,

$$\pi^{f1}(\mathbb{l}_0^{P1+f2}) = \pi^{f1}(\mathbb{l}_1^{P1+f2}) = (\emptyset, \emptyset)$$

[8] A world state of m is uniquely identified by its product configuration, since the only concepts in m's world model (Figure 4.7a) are the SPL and feature concepts. Because product configurations are static (Section 3.2.6) the world state of m is static.

The bisimulation relation BSR for $\pi^{P1}(\llbracket m \rrbracket)$ and $\pi^{P1+f2}(\llbracket m \rrbracket)$ can be constructed as follows:

- $\pi^{P1}(\llbracket m \rrbracket)$'s initial state $\mathbb{s}_0^{P1}$ corresponds to $\pi^{P1+f2}(\llbracket m \rrbracket)$'s initial state $\mathbb{s}_0^{P1+f2}$, because their modes are identical ($\boldsymbol{mode_{F1\{s1\}}}$), and the world state $\boldsymbol{ws^{P1+f2}}$ differs from the world state $\boldsymbol{ws^{P1}}$ in only the presence of feature $f2$. Hence, we initialize BSR to

$$BSR = \{\ (\mathbb{s}_0^{P1}, \mathbb{s}_0^{P1+f2})\ \}$$

- $\pi^{P1}(\llbracket m \rrbracket)$'s only move in $\mathbb{s}_0^{P1}$ is to take the transition $\mathbb{s}_0^{P1} \xrightarrow{\mathbb{l}_0^{P1}} \mathbb{s}_1^{P1}$. This move is intentionally matched by $\pi^{P1+f2}(\llbracket m \rrbracket)$'s move of taking the transition $\mathbb{s}_0^{P1+f2} \xrightarrow{\mathbb{l}_0^{P1+f2}} \mathbb{s}_1^{P1+f2}$. $\pi^{f1}(\mathbb{l}_0^{P1}) \approx \pi^{f1}(\mathbb{l}_0^{P1+f2})$ because the $f1$ transition $F1\{t1\}$ in $\mathbb{l}_0^{P1}$ is preempted by the $f2$ transition $F2\{t1\}$ in $\mathbb{l}_0^{P1+f2}$. Furthermore, since the $f2$ transition $F2\{t1\}$ in $\mathbb{l}_0^{P1+f2}$ prohibits entry to the $f1$ state $F1\{s2\}$ (the destination state of the preempted $f1$ transition $F2\{t1\}$), the matching game need not continue with further moves; that is, we need not add $(\mathbb{s}_1^{P1}, \mathbb{s}_1^{P1+f2})$ to BSR.

- Conversely, $\pi^{P1+f2}(\llbracket m \rrbracket)$'s only move in $\mathbb{s}_0^{P1+f2}$ is to take the transition $\mathbb{s}_0^{P1+f2} \xrightarrow{\mathbb{l}_0^{P1+f2}} \mathbb{s}_1^{P1+f2}$. This move is intentionally matched by $\pi^{P1}(\llbracket m \rrbracket)$'s move of taking the transition $\mathbb{s}_0^{P1} \xrightarrow{\mathbb{l}_0^{P1}} \mathbb{s}_1^{P1}$, since $\pi^{f1}(\mathbb{l}_0^{P1}) \approx \pi^{f1}(\mathbb{l}_0^{P1+f2})$, as shown above. Furthermore, as with the case above, we need not add $(\mathbb{s}_1^{P1}, \mathbb{s}_1^{P1+f2})$ to BSR.

Hence, we conclude that $\pi^{P1}(\llbracket m \rrbracket)$ is bisimilar to $\pi^{P1+f2}(\llbracket m \rrbracket)$, due to the existence of the bisimulation relation

$$BSR = \{\ (\mathbb{s}_0^{P1}, \mathbb{s}_0^{P1+f2})\ \}$$

which conforms to Definition 4.3.7.

4.3.3 Special Cases

Our notion of a modification interaction generalizes existing feature-interaction types that are special cases of behaviour modification. For completeness, this section defines two prominent special cases: looping and deadlock interactions.

4.3.3.1 Looping Interactions

To define a looping interaction, we must first define the notion of looping in the context of FORML.

Looping: An execution of $\llbracket m \rrbracket$ includes an infinite loop between a transition or action $\boldsymbol{x_1}$ and another transition or action $\boldsymbol{x_2}$, if after some execution state $\mathbb{s}_i$, each pair of consecutive labels $\mathbb{l}_{i+2k}$ and $\mathbb{l}_{i+2k+1}$ ($k \geq 0$) includes $\boldsymbol{x_1}$ but not $\boldsymbol{x_2}$, and $\boldsymbol{x_2}$ but not $\boldsymbol{x_1}$, respectively; that is, $\mathbb{l}_i$ includes $\boldsymbol{x_1}$ (but not $\boldsymbol{x_2}$) and $\mathbb{l}_{i+1}$ includes $\boldsymbol{x_2}$ (but not $\boldsymbol{x_1}$), $\mathbb{l}_{i+2}$ includes $\boldsymbol{x_1}$ (but not $\boldsymbol{x_2}$) and $\mathbb{l}_{i+3}$ includes $\boldsymbol{x_2}$ (but not $\boldsymbol{x_1}$), and so on.

Feature interaction: We can now define a looping interaction as follows: Let x be a feature in a product configuration P, and let be $P + y$ be the product configuration that results from adding y (and the features that it depends on) to P. y has a *looping interaction* with x, if there is an infinite loop between a transition or action of x and a transition or action of y in some execution of $\pi^{P+y}(\llbracket m \rrbracket)$. In this case, the transition or action of y triggers the transition or action of x and vice versa; hence looping interactions are instances of modification interactions.

Looping interactions are defined informally in the literature (e.g., [70, 20]). Our formal definition covers only minimal loops between a pair of transitions or actions $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$: that is, loops comprising only two $\llbracket m \rrbracket$ transitions, where each $\llbracket m \rrbracket$ transition includes only one of $\boldsymbol{x_1}$ or $\boldsymbol{x_2}$. This definition enables efficient analyses to detect minimal loops at the expense of excluding interactions with larger loops. However, larger loops can still be detected using (more expensive) analyses to detect general modification interactions.

Example: Figure 4.8 shows a fragment of the composed *TelSoft* model, which includes two transitions: transition $BCS\{t1\}$ of BCS, which models the initiation of a call by a user; and transition $CFB\{t1\}$ of CFB, which models the forwarding of an incoming call to a designated user, when the callee is already in a call. Consider a product configuration comprised of the three *TelSoft* products ts_1, ts_2, and ts_3, where (1) all three products have feature BCS; (2) ts_2 includes feature CFB, which forwards calls to ts_1's user; and (3) ts_1 includes feature CFB, which forwards calls to ts_2's user. This can result in a looping interaction between the CFB features of ts_1 and ts_2: Suppose that ts_1's user calls ts_2's user, who is already in a call with ts_3's user. This scenario is modelled by the execution of transition $ts_1 :: BCS\{t1\}$ of ts_1's machine, which triggers the execution of transition $ts_2 :: CFB\{t1\}$ of ts_2's machine, since ts_2's user is in a call with ts_3. The execution of

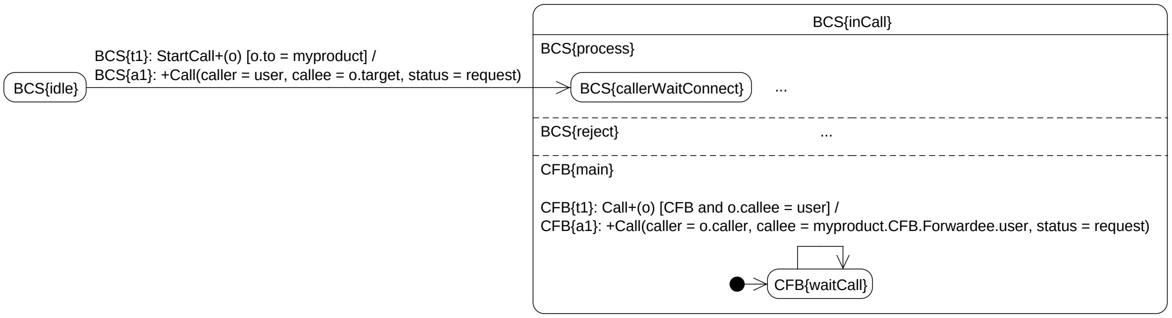

BCS{idle}
BCS{t1}: StartCall+(o) [o.to = myproduct] /
BCS{a1}: +Call(caller = user, callee = o.target, status = request)
BCS{inCall}
BCS{process}
BCS{callerWaitConnect}
...
BCS{reject}
...
CFB{main}
CFB{t1}: Call+(o) [CFB and o.callee = user] /
CFB{a1}: +Call(caller = o.caller, callee = myproduct.CFB.Forwardee.user, status = request)
CFB{waitCall}

Figure 4.8: Example of a looping interaction

transition $ts_2 :: CFB\{t1\}$ triggers the execution of transition $ts_1 :: CFB\{t1\}$, because ts_2 forwards the call to ts_1's user. Analogously, transition $ts_1 :: CFB\{t1\}$, in turn, triggers the execution of $ts_2 :: CFB\{t1\}$, starting an infinite loop between between the $CFB\{t1\}$ transitions of products ts_1 and ts_2. □

4.3.3.2 Deadlock Interactions

To define a deadlock interaction, we must first define the notion of deadlock in the context of FORML.

Deadlock: A machine *smi* (i.e., an instance of a state machine *sm* in the composed behaviour model (CBM)) is said to *deadlock* in its execution if it arrives at an execution state after which none of *smi*'s transitions can execute.

More precisely, consider an $\llbracket m \rrbracket$ execution corresponding to the concurrent execution of a set of machines, including *smi*. The $\llbracket m \rrbracket$ execution causes *smi* to deadlock if there is exists an $\llbracket m \rrbracket$ state $\mathbb{s}_i$ in the execution, such that for any execution e of $\llbracket m \rrbracket$ with a suffix

$$(\mathbb{s}_i, \mathbb{l}_i, \mathbb{s}_{i+1}), (\mathbb{s}_{i+1}, \mathbb{l}_{i+1}, \mathbb{s}_{i+2}), \ldots$$

none of the labels $\mathbb{l}_i, \mathbb{l}_{i+1}, \ldots$ in e include any transitions of *smi*.

Feature interaction: We can now define a deadlock interaction as follows: Let x be a feature in a product configuration P, and let be $P + y$ be the product configuration that results from adding y (and the features that it depends on) to P. y has a *deadlock* *interaction* with x if a machine *smi* that includes transitions or actions of x is deadlock-free in executions of $\pi^{P}(\llbracket m \rrbracket)$ but not in executions of $\pi^{P+y}(\llbracket m \rrbracket)$. In this case, y has prohibited some transitions and actions of x; hence, deadlock interactions are instances of modification interactions.

Example: Our case studies do not appear to include instances of deadlock interactions. However, to demonstrate the idea, an pedagogical example is given below. Figure 4.9 shows the CBM of an SPL comprising three optional features $F1$, $F2$, and $F3$. Consider a product configuration comprising two products p_1 with feature $F1$ and product p_2 with feature $F2$. The two products in this product configuration execute in a loop: The $F2\{sm\}$ machine executes transition $F2\{t1\}$, which adds a C object to the world state. The added C object triggers transition $F1\{t1\}$ of the $F1\{sm\}$ machine and, concurrently, the $F2\{sm\}$ machine executes transition $F2\{t2\}$, which removes a C object from the world state. The removed C object triggers transition $F1\{t2\}$ of the $F1\{sm\}$ machine and, concurrently, the $F2\{sm\}$ machine executes transition $F2\{t1\}$, and so on. Now suppose that feature $F3$ is added to p_2. This leads to a deadlock interaction between $F3$ and $F1$ in p_1: Transition $F3\{t1\}$ always preempts transition $F2\{t2\}$, which (assuming that C is a controlled concept) permanently removes the enabling condition of transition $F1\{t2\}$; hence, the $F1\{sm\}$ machine gets stuck in state $F1\{s2\}$ and a deadlock occurs. □

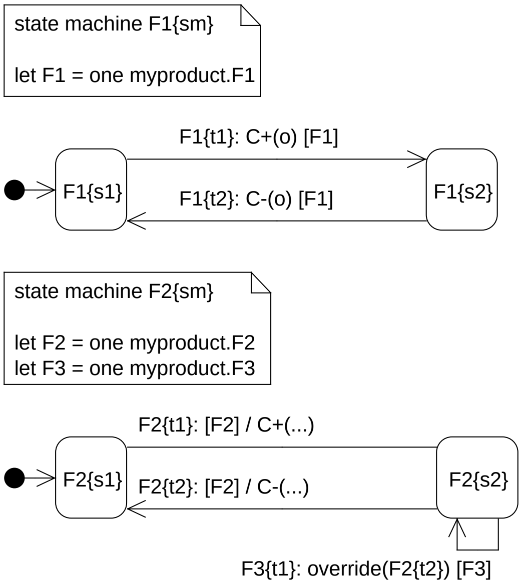


Figure 4.9: Example of a deadlock interaction

4.4 Chapter Summary

This chapter presents a formal taxonomy of feature interactions for FORML. Our proposed taxonomy adapts existing taxonomies defined for feature-oriented artifacts (FOAs) expressed as operational models (see Section 2.3) and comprises the following feature-interaction types: *Nondeterminism interactions* occur when there is a nondeterministic choice among the enabled transitions of multiple features. *Conflict interactions* occur when the enabled transitions or actions of multiple features have inconsistent destination states or effects on the world, respectively. *Modification interactions* occur when the behaviours of one feature trigger or prohibit the behaviours of another feature. Finally, we consider two special cases of modification interactions: *deadlock interactions* occur when one feature causes a deadlock to occur, which in turn prohibits behaviours of another feature; and *looping interactions* occur when two features trigger one another's behaviours in an infinite loop.

Our main adaptation to existing taxonomies is in our notion of modification interactions. Modification interactions formalize the informal notion of behaviour modification in existing taxonomies by checking whether two transition systems are bisimilar. The two transition systems represent the behaviours of a feature x in two product configurations that differ in the presence of a feature y and its dependent features; the bisimilarity check reveals whether y modifies the behaviours of x. Furthermore, our notion of modification interactions excludes behaviour modifications caused by FORML's constructs for modelling intended interactions. By excluding intended interactions, the definition enables analyses that report only unintended feature interactions.

Chapter 5

Conclusion and Future Work

This chapter presents a summary of this thesis and its contributions, as well as possible directions for future work.

5.1 Summary of Thesis and Contributions

This thesis introduces the feature-oriented requirements modelling language (FORML) for specifying feature-oriented models of a software product line (SPL)'s requirements, as well as a taxonomy of feature interactions for FORML to support the detection of unintended feature interactions at the requirements level.

5.1.1 FORML

A FORML model of an SPL's requirements is comprised of two main views:

- A *world model* is an ontology of concepts that describes a world comprising the SPL's products and the environment in which they will operate. The concepts in the world model are expressed in the UML class-diagram notation. The world model also includes auxiliary validity constraints over single instances of the world model, called *world states*, as well as pairs of consecutive world states. The world-model constraints are expressed in FORML's expression language.

- A *behaviour model* is a state-machine model that describes the requirements for an SPL's products. The model's inputs are events and conditions over the world model, and its outputs are actions over the world model. A behaviour model is structured in terms of features: that is, the requirements of each feature of an SPL are localized in a separate *feature module*. Those requirements of a feature that are independent of existing features are expressed as a set of parallel *state machines*. If the feature *enhances* (i.e., extends or modifies) existing features, the enhancements are expressed as a set of *state-machine fragments* that extend existing feature modules. The graphical syntax for the behaviour model is based on the UML state-machine notation. The inputs and outputs of the behaviour model are expressed in FORML's expression language. To obtain a view of the requirements of feature combinations, all of the feature modules are composed into a model of the whole SPL (i.e., all possible products). At the heart of composition is the integration of the feature modules' fragments and state machines. This step of composition is formalized as the *superimposition* of the feature module's abstract syntax trees, called feature structure trees (FSTs). FORML FSTs have properties (e.g., being unordered, not having common terminal nodes) that ensure that their superimposition is commutative and associative. The commutativity and associativity of superimposition (and hence composition) can lead to savings in analysis costs, since only one rather than (possibly) multiple composition orders need to be analyzed.

Two case studies have been performed, one from the automotive domain and one from the telephony domain, with the goals of (1) exploring the expressiveness of FORML and (2) evaluating the impact of evolving a FORML model with new features. The second goal was to evaluate the likelihood that a new feature module prompts changes to existing feature modules and to measure the extents of such changes. The automotive case study was adapted from GM Feature Technical Specifications for a set of 11 automotive software features. The telephony case study was adapted from the Second Feature Interaction Contest [59] and comprises 15 telephone-service features. The results of the case studies indicate good expressiveness for FORML (a minor adjustment to FORML was needed to improve expressiveness in modelling 20 new features), and low impact of adding new features to a FORML model (adding new features did not result in changes to existing feature modules).

5.1.1.1 Contributions

FORML combines and adapts the best practices for requirements modelling with feature modularity techniques from FOSD research. FORML is distinguished from existing requirements modelling and FOSD approaches in the following ways:

- A FORML world model includes a feature-oriented representation of an SPL's products: FORML introduces *SPL* and *feature concepts* to represent products and their feature configurations. Feature concepts can also be used to group product phenomena (i.e., observable phenomena introduced by products) based on the features to which they pertain. Furthermore, a FORML world model includes a *feature model*, which specifies constraints on the valid feature configurations of products.

- FORML provides a systematic treatment of modelling feature enhancements in state-machine models of feature requirements. FORML distinguishes between different types of enhancements: enhancements that *add* new requirements in the context of an existing feature's requirements, and enhancements that *modify* the requirements of existing features (i.e., intended feature interactions). Intended interactions are further distinguished based on whether they trigger, prohibit, or override the requirements of existing features; or specify an existing feature's priority over other (new or existing) features (i.e., retrospective intended interactions). FORML prescribes different types of fragments for modelling different types of enhancements. In particular, FORML introduces novel constructs for modelling requirements overrides in state-machine models of feature requirements: namely, special transitions that override other transitions in the same or different state machine, and special transition actions that override other actions in the same transition.

- The composition of FORML feature modules is commutative and associative, despite models of intended interactions. As is discussed in Section 2.2, existing FOSD approaches typically sacrifice commutativity for the sake of modelling intended interactions.

5.1.2 Feature-Interaction Taxonomy

This thesis presents a taxonomy of feature interactions for FORML that is an adaptation of existing taxonomies for operational models of feature behaviour. The taxonomy consists of the following feature-interaction types, which are precisely defined to enable analyses for detecting unintended feature interactions in FORML models:

- A *nondeterminism interaction* occurs when there is a nondeterministic choice in the set of enabled transitions that can execute in an execution step.

- A *conflict interaction* occurs when the set of actions that execute in an execution step are inconsistent with one another or with the world model, or when the set of transitions that execute in a step have mutually inconsistent destination states (e.g., the destination states are different child states in the same region).

- A *modification interaction* occurs when one feature modifies the behaviours of another feature. The notion of a modification interaction generalizes a number of existing feature-interaction types that are special cases of behaviour modification. For completeness, the proposed taxonomy includes two types of feature interactions that are prominent special cases of modification interactions: *deadlock interaction*, which occurs when a feature causes a deadlock that prohibits another feature's behaviours; and *looping interaction*, whichs occur when two features trigger one another's behaviours in an infinite loop.

A set of seven examples have been developed of the different feature-interaction types in the proposed taxonomy: one example of nondeterminism interaction, two examples of conflict interaction, and four examples of modification interaction. Six of the examples are based on the FORML models of the telephony and automotive case studies; the remaining example is pedagogical and was created to illustrate deadlock interaction.

5.1.2.1 Contributions

The taxonomy of feature interactions for FORML makes the following adaptations to existing taxonomies:

- The taxonomy includes a formal definition for behaviour modification. Behaviour modification covers several special cases of feature interactions in existing taxonomies.

- FORML's constructs for modelling intended feature interactions cause modification interactions. To enable feature-interaction analyses that only report unintended modification interactions, the proposed definition of modification interactions excludes the cases caused by such constructs.

5.2 Future Work

The following are possible directions for extending the work presented in this thesis.

5.2.1 More Validation

More case studies (from the automotive and telephony domains, as well as other domains such as banking, email, and smart homes) should be performed to evaluate the expressiveness of FORML, as well as the ease of evolving FORML models with new features. Furthermore, the utility of the feature-interaction taxonomy should be further evaluated by (manually or automatically) detecting all instances of the taxonomy's feature interactions in the case-study models. Finally, user studies should be performed to (1) evaluate the ease of reading and writing FORML models, and (2) to compare the common approach of realizing behaviour overrides by ordered composition (e.g., AHEAD [11], DFC [94], LSM [16]) with FORML's approach of modelling behaviour overrides using transition and action priorities.

5.2.2 Extending FORML

FORML can be extended in several ways:

- FORML can be extended from supporting purely discrete behaviour models to supporting *hybrid* behaviour models that specify both discrete and continuous behaviours. This extension would enable more complete models of cyber-physical systems such as automobile software controllers.
- FORML's expression language can be extended to support temporal logic for specifying world-model constraints.
- Currently, the state machines in the composed behaviour model (CBM) are instantiated once per product. FORML can be extended to support custom instantiation criteria for state machines to allow more flexibility in specifying behaviours. For example, to support feature cardinalities that allow a product to have more than one instance of a feature, it could be specified that a state machine is instantiated once per feature instance. However, if state machines are instantiated per feature instance, specifying the common case of intended interactions between the features of the same product will require references to elements in other features' machines; in contrast, if state machines are instantiated per product (our current approach), such interactions can be specified using simpler references to elements in the same machine.
- FORML can be extended to support timing requirements.

5.2.3 Feature-Interaction Analyses

Automating the detection of the proposed taxonomy's feature-interaction types is an obvious direction for future work. There is already ongoing work in our group for using model checking to detect conflict interactions: this work involves automatically composing FORML feature modules using the FeatureHouse framework [5] for superimposing FSTs, translating FORML models into SMV using a semantically-configurable translator [31], and automatically generating conflict-detection properties from FORML models. Model checking can also be used to detect nondeterminism interactions, deadlock interactions, and looping interactions. However, bisimulation analyses should be investigated for detecting the general case of modification interactions.

An important consideration in analysing FORML models is dealing with the (generally) infinite space of valid world states. One way to address this issue is to perform bounded analysis: that is, to either to scope the types defined by the world model to a finite set of elements (similar to how Alloy models are analyzed[48]), or to place a bound on the length of the execution paths considered in the analysis (e.g., to perform bounded model checking [14]). Another consideration is dealing with dynamic product configurations: that is, cases where a product configuration changes in the course of an execution (e.g., the features of a particular product change). Addressing this issue will require modifying the semantics of FORML, which currently assumes static product configurations.

5.2.4 Tool Support

FORML can benefit from tool support both for the purpose of automating analysis as described above, and also for creating and editing FORML models. Our group has developed a textual syntax for FORML that is currently being used as input to the feature composer. However, a graphical editor for FORML models is yet to be produced.

APPENDICES

Appendix A

Case-Study Models

This appendix shows the evolution of FORML models for a telephony SPL called *TelSoft* (Section A.1), and an automotive case study called *AutoSoft* (Section A.2). Each evolution step corresponds to the addition of a new feature and is shown as the new feature's feature module in the behaviour model, followed by the world model as evolved by the new feature.

A.1 Telephony Case Study

Figures A.1 to A.30 show, in order, the evolution steps of a FORML model of *TelSoft* .

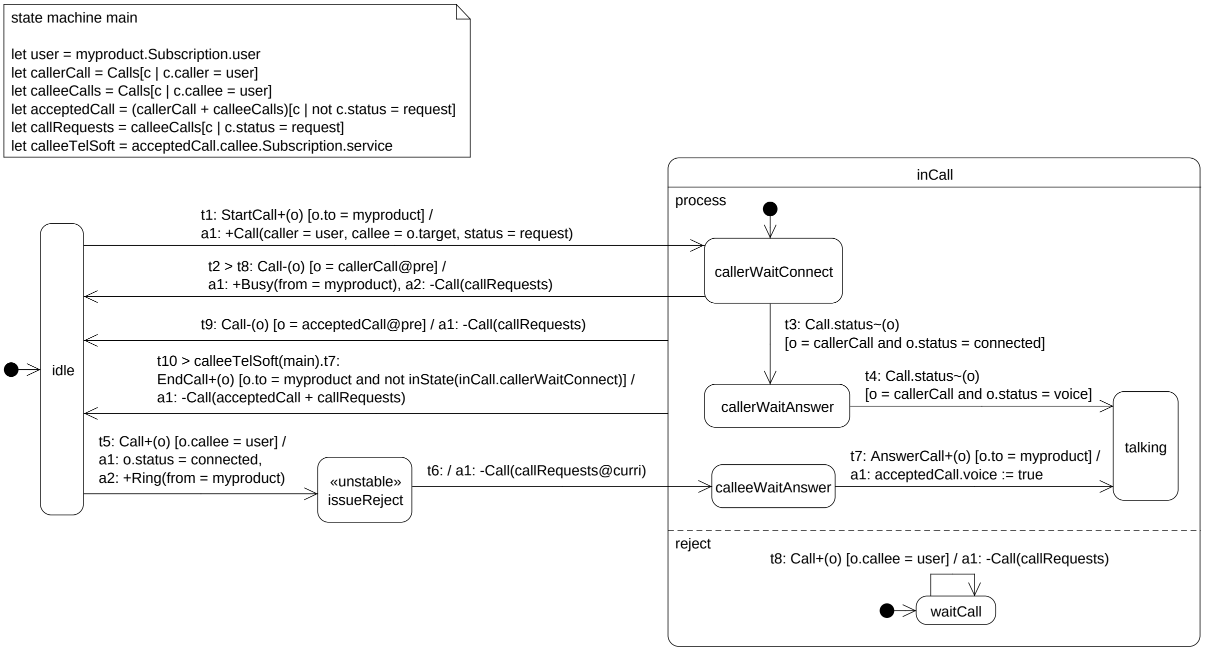

state machine main
let user = myproduct.Subscription.user
let callerCall = Calls[c | c.caller = user]
let calleeCalls = Calls[c | c.callee = user]
let acceptedCall = (callerCall + calleeCalls)[c | not c.status = request]
let callRequests = calleeCalls[c | c.status = request]
let calleeTelSoft = acceptedCall.callee.Subscription.service
inCall
process
idle
t1: StartCall+(o) [o.to = myproduct] /
a1: +Call(caller = user, callee = o.target, status = request)
t2 > t8: Call-(o) [o = callerCall@pre] /
a1: +Busy(from = myproduct), a2: -Call(callRequests)
t9: Call-(o) [o = acceptedCall@pre] / a1: -Call(callRequests)
t10 > calleeTelSoft(main).t7:
EndCall+(o) [o.to = myproduct and not inState(inCall.callerWaitConnect)] /
a1: -Call(acceptedCall + callRequests)
t5: Call+(o) [o.callee = user] /
a1: o.status = connected,
a2: +Ring(from = myproduct)
«unstable»
issueReject
t6: / a1: -Call(callRequests@curri)
callerWaitConnect
t3: Call.status~(o)
[o = callerCall and o.status = connected]
callerWaitAnswer
t4: Call.status~(o)
[o = callerCall and o.status = voice]
talking
calleeWaitAnswer
t7: AnswerCall+(o) [o.to = myproduct] /
a1: acceptedCall.voice := true
reject
t8: Call+(o) [o.callee = user] / a1: -Call(callRequests)
waitCall

Figure A.1: The feature module of BCS

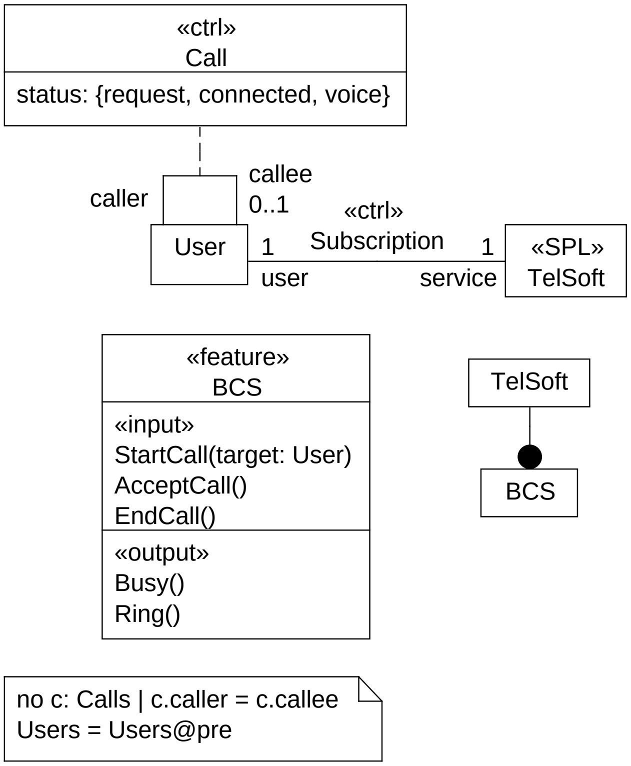
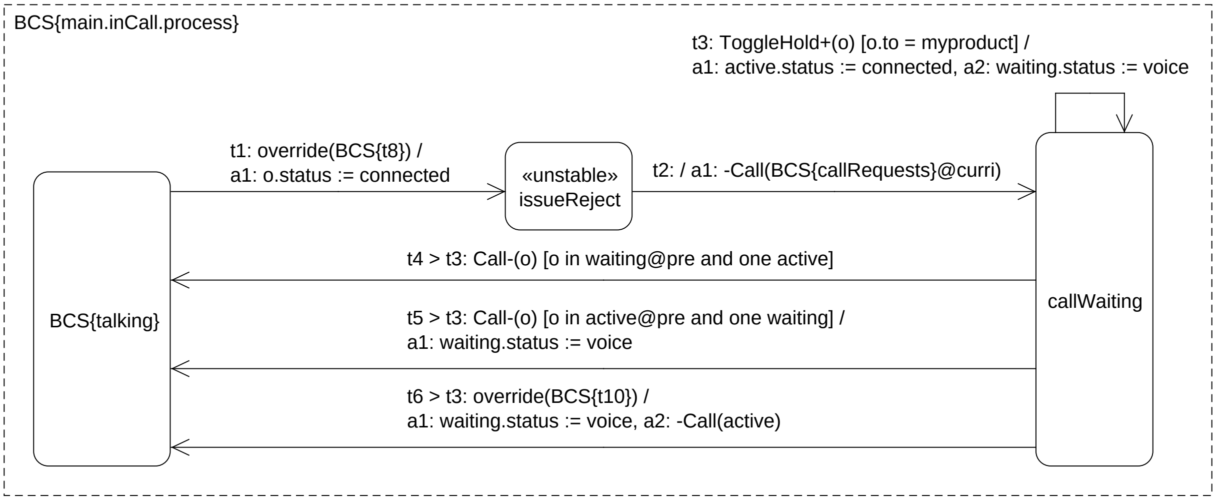


Figure A.2: The *TelSoft* world model as evolved by BCS

```
let calls = Calls[c | c.caller = BCS{user} or c.callee = BCS{user}]
let active = calls[c | c.status = voice]
let waiting = calls[c | c.status = connected]
let acceptedCalls = active + waiting

let holder = if(waiting.caller = BCS{user}) then waiting.callee else waiting.caller
let holderTelSoft = holder.Subscription.service
transition BCS{t10} > holderTelSoft(BCS{main}).CW{t5}, holderTelSoft(BCS{main}).CW{t6}
```



BCS{main.inCall.process}
t3: ToggleHold+(o) [o.to = myproduct] /
a1: active.status := connected, a2: waiting.status := voice
t1: override(BCS{t8}) /
a1: o.status := connected
«unstable»
issueReject
t2: / a1: -Call(BCS{callRequests}@curri)
t4 > t3: Call-(o) [o in waiting@pre and one active]
BCS{talking}
callWaiting
t5 > t3: Call-(o) [o in active@pre and one waiting] /
a1: waiting.status := voice
t6 > t3: override(BCS{t10}) /
a1: waiting.status := voice, a2: -Call(active)

Figure A.3: The feature module of CW

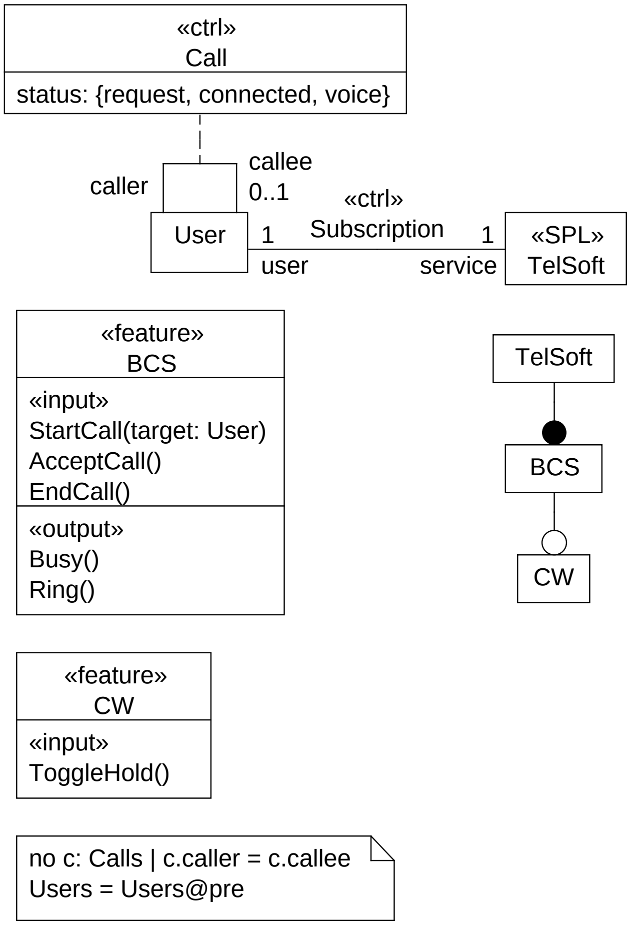


Figure A.4: The *TelSoft* world model as evolved by CW

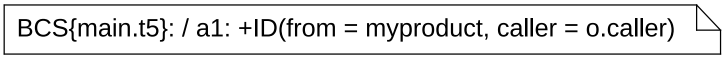

BCS{main.t5}: / a1: +ID(from = myproduct, caller = o.caller)

Figure A.5: The feature module of CD

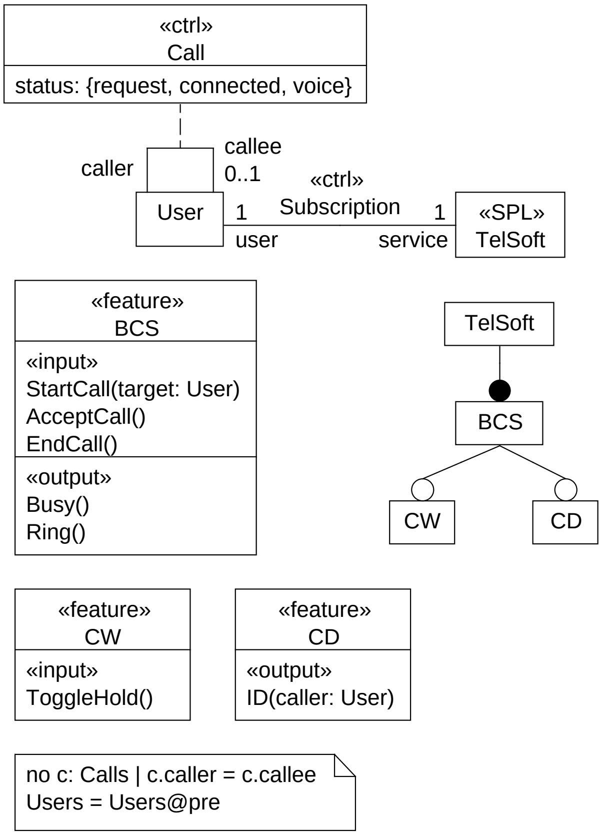


Figure A.6: The *TelSoft* world model as evolved by CD

strengthen action CD{a1} with $s1: no (o.caller).Subscription.service.CDB

Figure A.7: The feature module of CDB

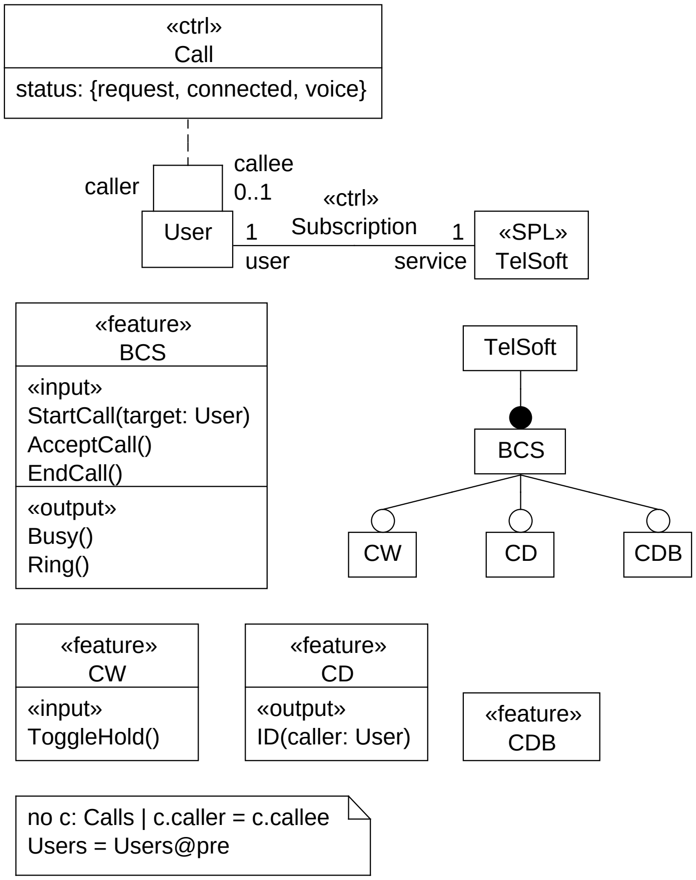


Figure A.8: The *TelSoft* world model as evolved by CDB

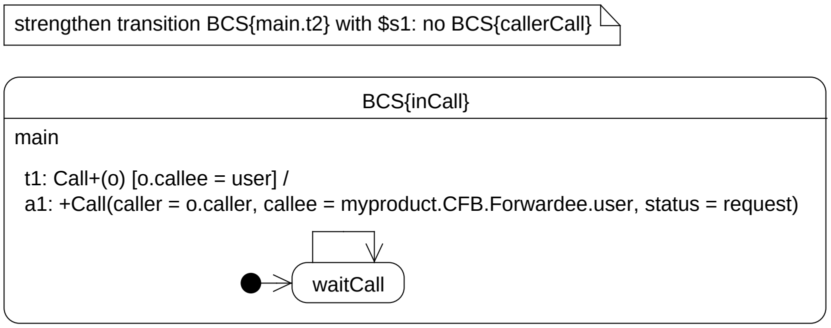

strengthen transition BCS{main.t2} with $s1: no BCS{callerCall}
BCS{inCall}
main
t1: Call+(o) [o.callee = user] /
a1: +Call(caller = o.caller, callee = myproduct.CFB.Forwardee.user, status = request)
waitCall

Figure A.9: The feature module of CFB

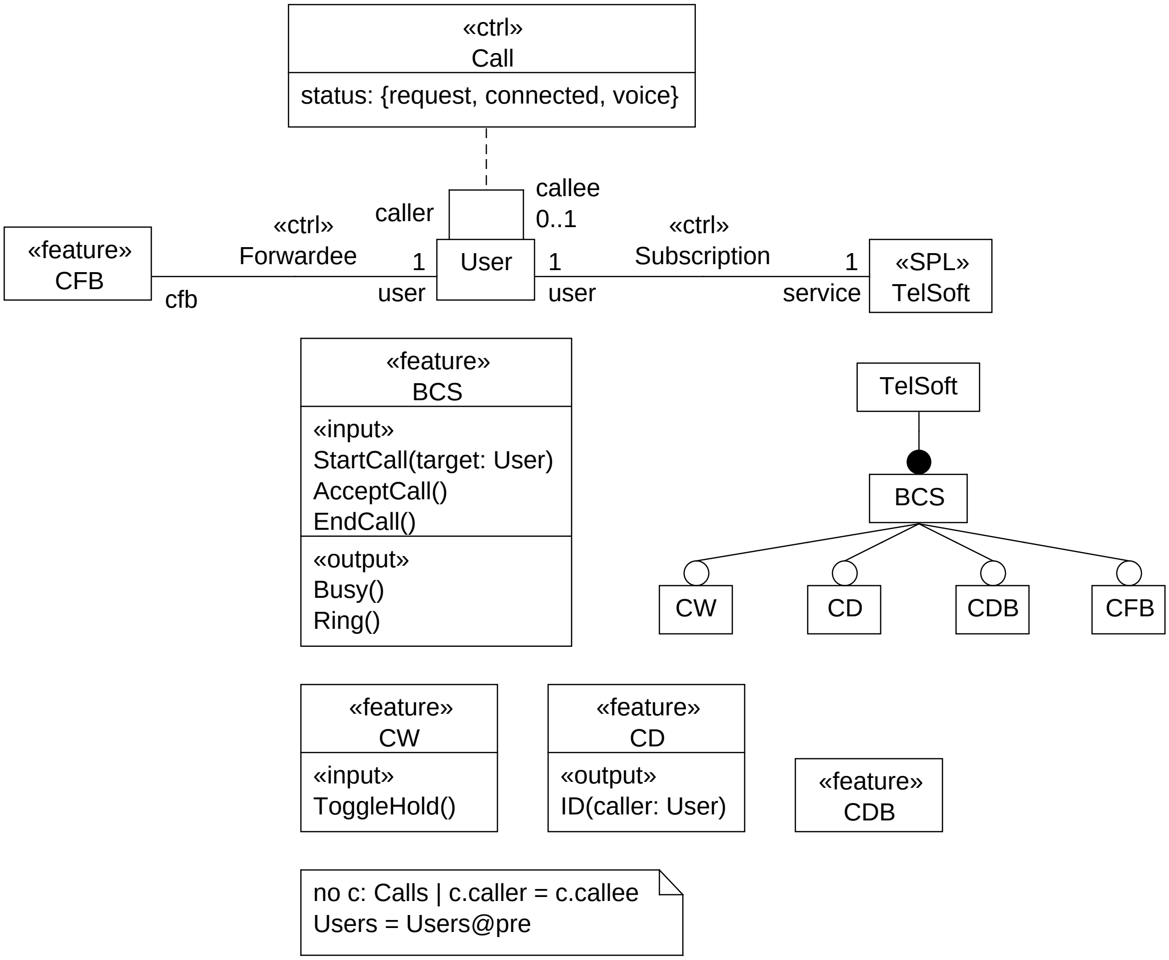


Figure A.10: The *TelSoft* world model as evolved by CFB

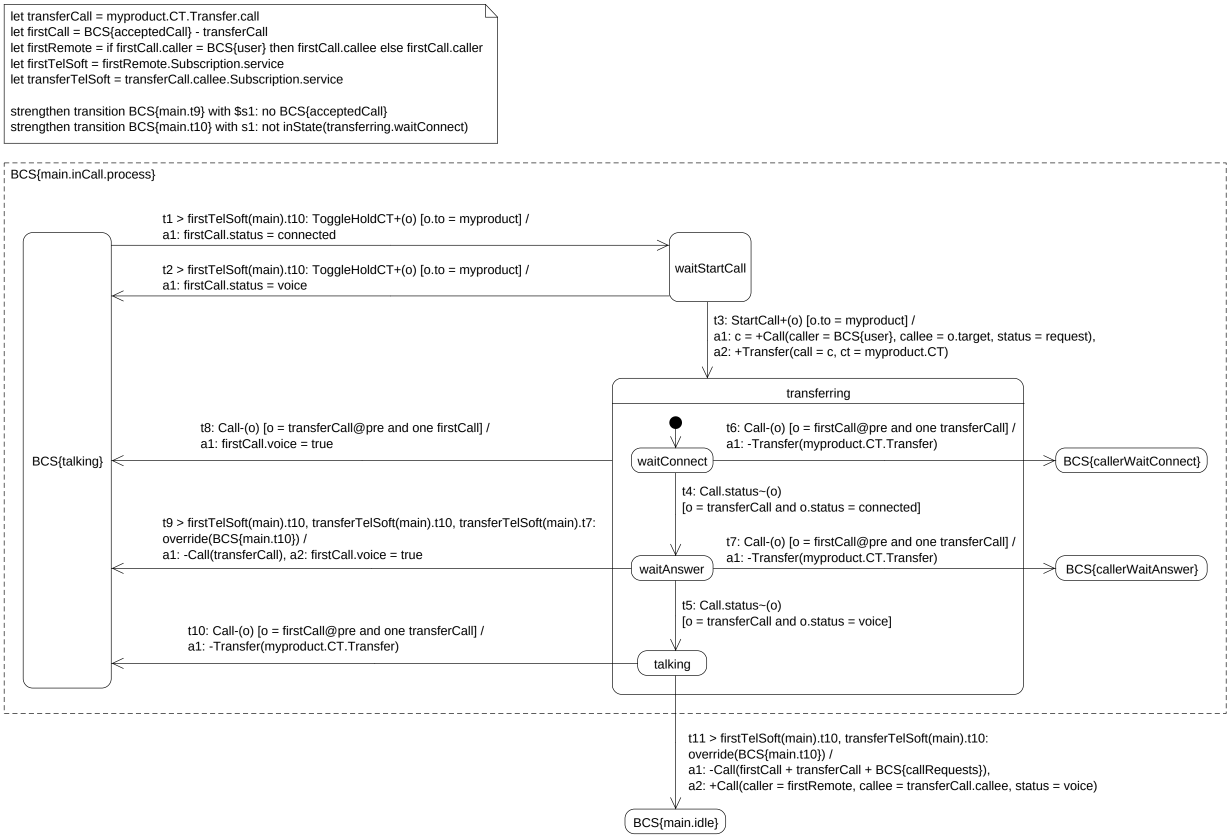

let transferCall = myproduct.CT.Transfer.call
let firstCall = BCS{acceptedCall} - transferCall
let firstRemote = if firstCall.caller = BCS{user} then firstCall.callee else firstCall.caller
let firstTelSoft = firstRemote.Subscription.service
let transferTelSoft = transferCall.callee.Subscription.service
strengthen transition BCS{main.t9} with $s1: no BCS{acceptedCall}
strengthen transition BCS{main.t10} with s1: not inState(transferring.waitConnect)
BCS{main.inCall.process}
t1 > firstTelSoft(main).t10: ToggleHoldCT+(o) [o.to = myproduct] /
a1: firstCall.status = connected
t2 > firstTelSoft(main).t10: ToggleHoldCT+(o) [o.to = myproduct] /
a1: firstCall.status = voice
waitStartCall
t3: StartCall+(o) [o.to = myproduct] /
a1: c = +Call(caller = BCS{user}, callee = o.target, status = request),
a2: +Transfer(call = c, ct = myproduct.CT)
transferring
t8: Call-(o) [o = transferCall@pre and one firstCall] /
a1: firstCall.voice = true
BCS{talking}
waitConnect
t6: Call-(o) [o = firstCall@pre and one transferCall] /
a1: -Transfer(myproduct.CT.Transfer)
BCS{callerWaitConnect}
t4: Call.status~(o)
[o = transferCall and o.status = connected]
t9 > firstTelSoft(main).t10, transferTelSoft(main).t10, transferTelSoft(main).t7:
override(BCS{main.t10}) /
a1: -Call(transferCall), a2: firstCall.voice = true
t7: Call-(o) [o = firstCall@pre and one transferCall] /
a1: -Transfer(myproduct.CT.Transfer)
waitAnswer
BCS{callerWaitAnswer}
t5: Call.status~(o)
[o = transferCall and o.status = voice]
t10: Call-(o) [o = firstCall@pre and one transferCall] /
a1: -Transfer(myproduct.CT.Transfer)
talking
t11 > firstTelSoft(main).t10, transferTelSoft(main).t10:
override(BCS{main.t10}) /
a1: -Call(firstCall + transferCall + BCS{callRequests}),
a2: +Call(caller = firstRemote, callee = transferCall.callee, status = voice)
BCS{main.idle}

Figure A.11: The feature module of CT

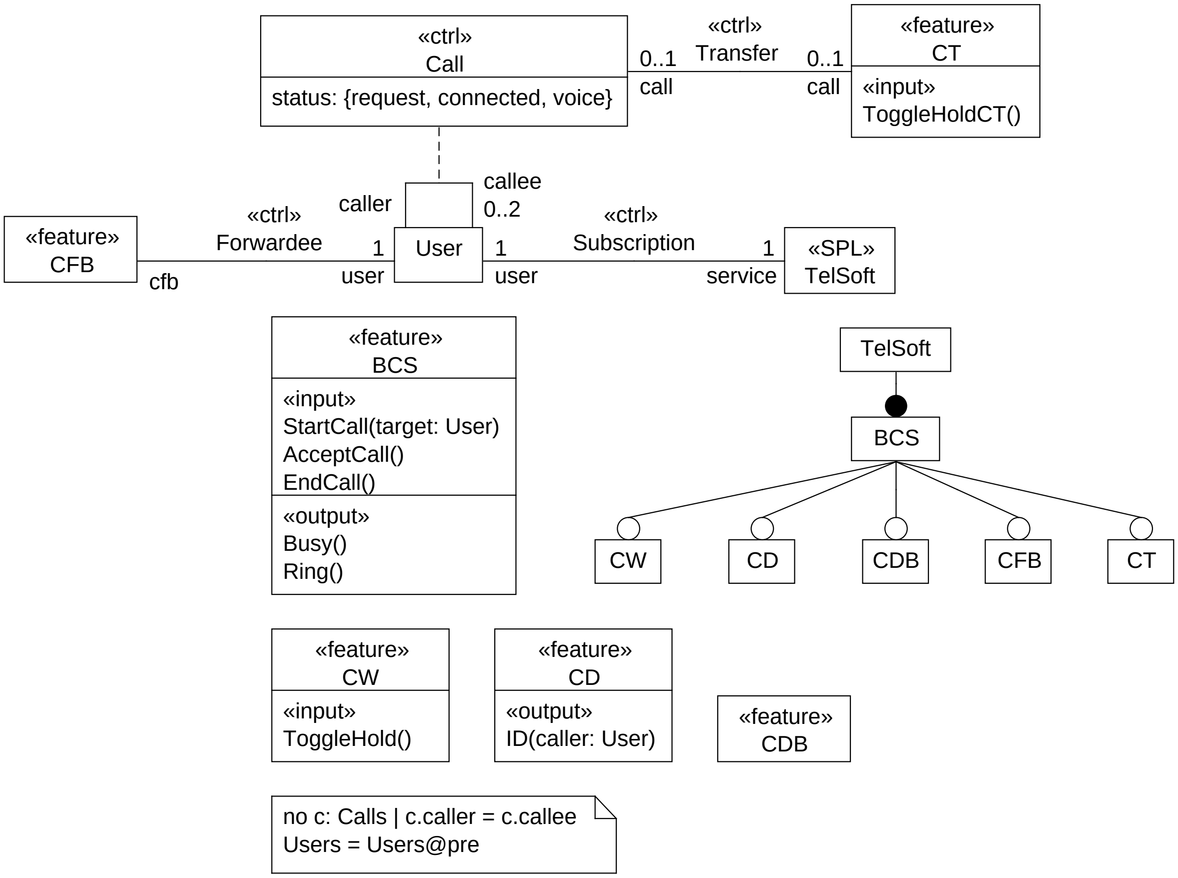


Figure A.12: The *TelSoft* world model as evolved by CT

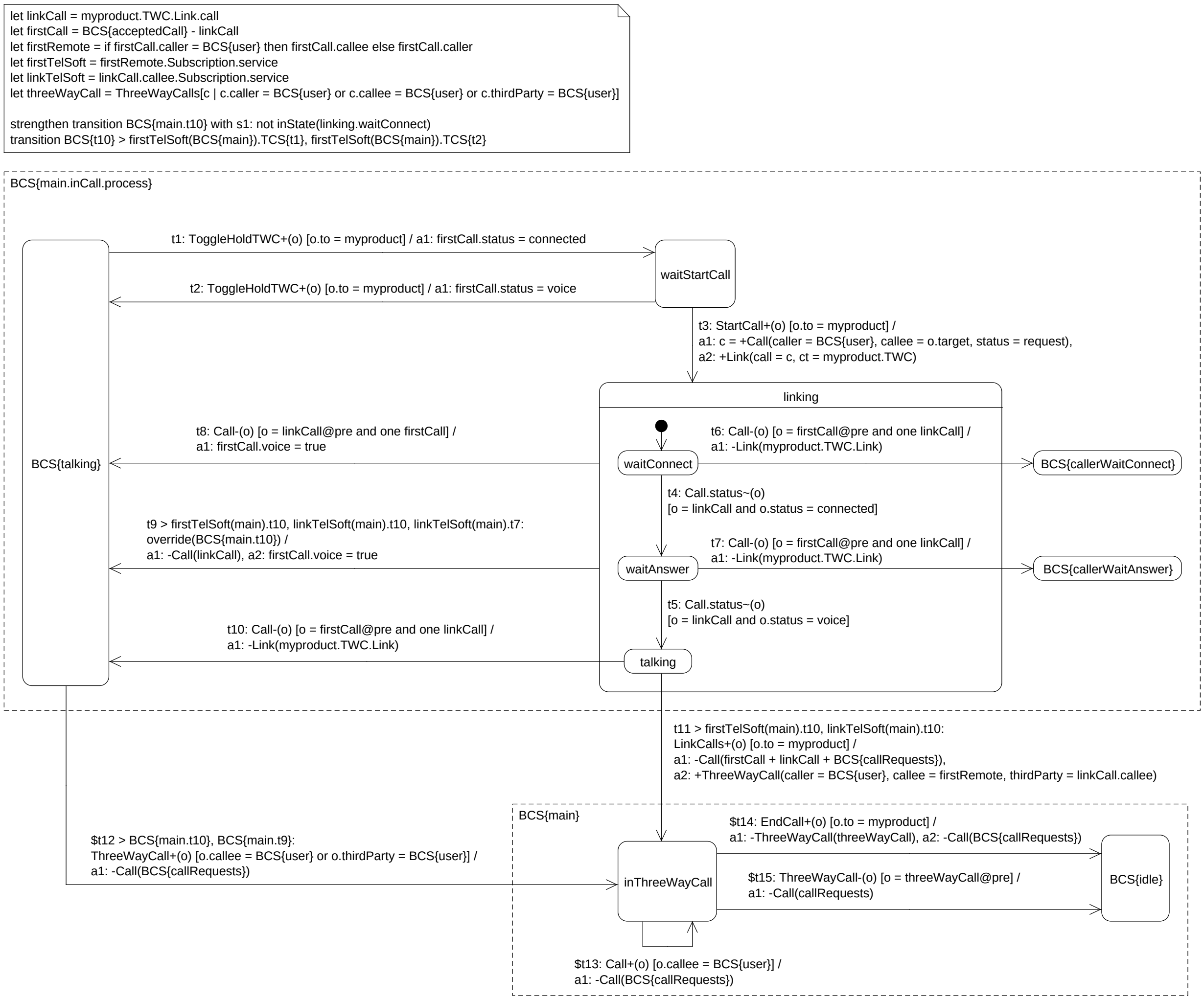

let linkCall = myproduct.TWC.Link.call
let firstCall = BCS{acceptedCall} - linkCall
let firstRemote = if firstCall.caller = BCS{user} then firstCall.callee else firstCall.caller
let firstTelSoft = firstRemote.Subscription.service
let linkTelSoft = linkCall.callee.Subscription.service
let threeWayCall = ThreeWayCalls[c | c.caller = BCS{user} or c.callee = BCS{user} or c.thirdParty = BCS{user}]
strengthen transition BCS{main.t10} with s1: not inState(linking.waitConnect)
transition BCS{t10} > firstTelSoft(BCS{main}).TCS{t1}, firstTelSoft(BCS{main}).TCS{t2}
BCS{main.inCall.process}
t1: ToggleHoldTWC+(o) [o.to = myproduct] / a1: firstCall.status = connected
waitStartCall
t2: ToggleHoldTWC+(o) [o.to = myproduct] / a1: firstCall.status = voice
t3: StartCall+(o) [o.to = myproduct] /
a1: c = +Call(caller = BCS{user}, callee = o.target, status = request),
a2: +Link(call = c, ct = myproduct.TWC)
linking
t8: Call-(o) [o = linkCall@pre and one firstCall] /
a1: firstCall.voice = true
t6: Call-(o) [o = firstCall@pre and one linkCall] /
a1: -Link(myproduct.TWC.Link)
BCS{talking}
waitConnect
BCS{callerWaitConnect}
t4: Call.status~(o)
[o = linkCall and o.status = connected]
t9 > firstTelSoft(main).t10, linkTelSoft(main).t10, linkTelSoft(main).t7:
override(BCS{main.t10}) /
a1: -Call(linkCall), a2: firstCall.voice = true
t7: Call-(o) [o = firstCall@pre and one linkCall] /
a1: -Link(myproduct.TWC.Link)
waitAnswer
BCS{callerWaitAnswer}
t5: Call.status~(o)
[o = linkCall and o.status = voice]
t10: Call-(o) [o = firstCall@pre and one linkCall] /
a1: -Link(myproduct.TWC.Link)
talking
t11 > firstTelSoft(main).t10, linkTelSoft(main).t10:
LinkCalls+(o) [o.to = myproduct] /
a1: -Call(firstCall + linkCall + BCS{callRequests}),
a2: +ThreeWayCall(caller = BCS{user}, callee = firstRemote, thirdParty = linkCall.callee)
BCS{main}
$t14: EndCall+(o) [o.to = myproduct] /
a1: -ThreeWayCall(threeWayCall), a2: -Call(BCS{callRequests})
$t12 > BCS{main.t10}, BCS{main.t9}:
ThreeWayCall+(o) [o.callee = BCS{user} or o.thirdParty = BCS{user}] /
a1: -Call(BCS{callRequests})
inThreeWayCall
$t15: ThreeWayCall-(o) [o = threeWayCall@pre] /
a1: -Call(callRequests)
BCS{idle}
$t13: Call+(o) [o.callee = BCS{user}] /
a1: -Call(BCS{callRequests})

Figure A.13: The feature module of TWC

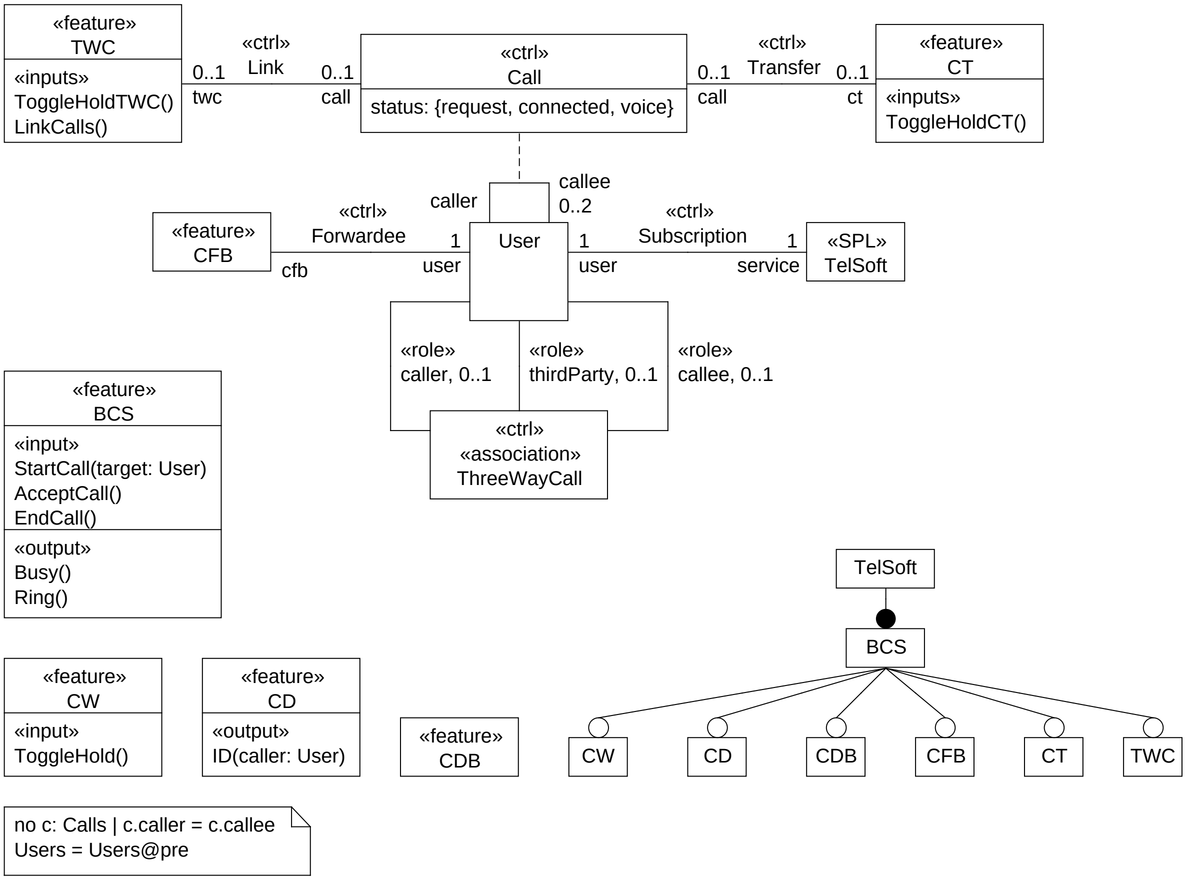
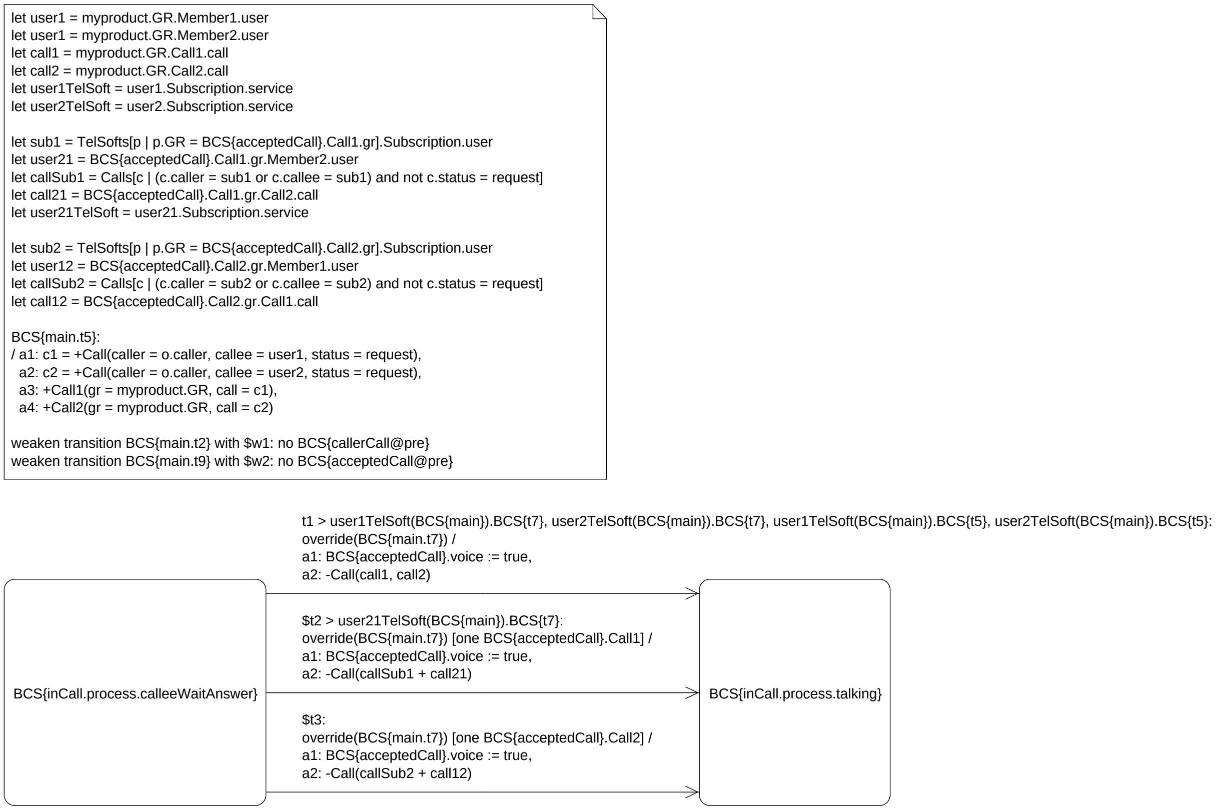


Figure A.14: The *TelSoft* world model as evolved by TWC



```
let user1 = myproduct.GR.Member1.user
let user1 = myproduct.GR.Member2.user
let call1 = myproduct.GR.Call1.call
let call2 = myproduct.GR.Call2.call
let user1TelSoft = user1.Subscription.service
let user2TelSoft = user2.Subscription.service

let sub1 = TelSofts[p | p.GR = BCS{acceptedCall}.Call1.gr].Subscription.user
let user21 = BCS{acceptedCall}.Call1.gr.Member2.user
let callSub1 = Calls[c | (c.caller = sub1 or c.callee = sub1) and not c.status = request]
let call21 = BCS{acceptedCall}.Call1.gr.Call2.call
let user21TelSoft = user21.Subscription.service

let sub2 = TelSofts[p | p.GR = BCS{acceptedCall}.Call2.gr].Subscription.user
let user12 = BCS{acceptedCall}.Call2.gr.Member1.user
let callSub2 = Calls[c | (c.caller = sub2 or c.callee = sub2) and not c.status = request]
let call12 = BCS{acceptedCall}.Call2.gr.Call1.call

BCS{main.t5}:
/ a1: c1 = +Call(caller = o.caller, callee = user1, status = request),
  a2: c2 = +Call(caller = o.caller, callee = user2, status = request),
  a3: +Call1(gr = myproduct.GR, call = c1),
  a4: +Call2(gr = myproduct.GR, call = c2)

weaken transition BCS{main.t2} with $w1: no BCS{callerCall@pre}
weaken transition BCS{main.t9} with $w2: no BCS{acceptedCall@pre}
```

```
t1 > user1TelSoft(BCS{main}).BCS{t7}, user2TelSoft(BCS{main}).BCS{t7}, user1TelSoft(BCS{main}).BCS{t5}, user2TelSoft(BCS{main}).BCS{t5}:
override(BCS{main.t7}) /
a1: BCS{acceptedCall}.voice := true,
a2: -Call(call1, call2)
```

```
$t2 > user21TelSoft(BCS{main}).BCS{t7}:
override(BCS{main.t7}) [one BCS{acceptedCall}.Call1] /
a1: BCS{acceptedCall}.voice := true,
a2: -Call(callSub1 + call21)
```

```
$t3:
override(BCS{main.t7}) [one BCS{acceptedCall}.Call2] /
a1: BCS{acceptedCall}.voice := true,
a2: -Call(callSub2 + call12)
```

BCS{inCall.process.calleeWaitAnswer}
BCS{inCall.process.talking}

Figure A.15: The feature module of GR

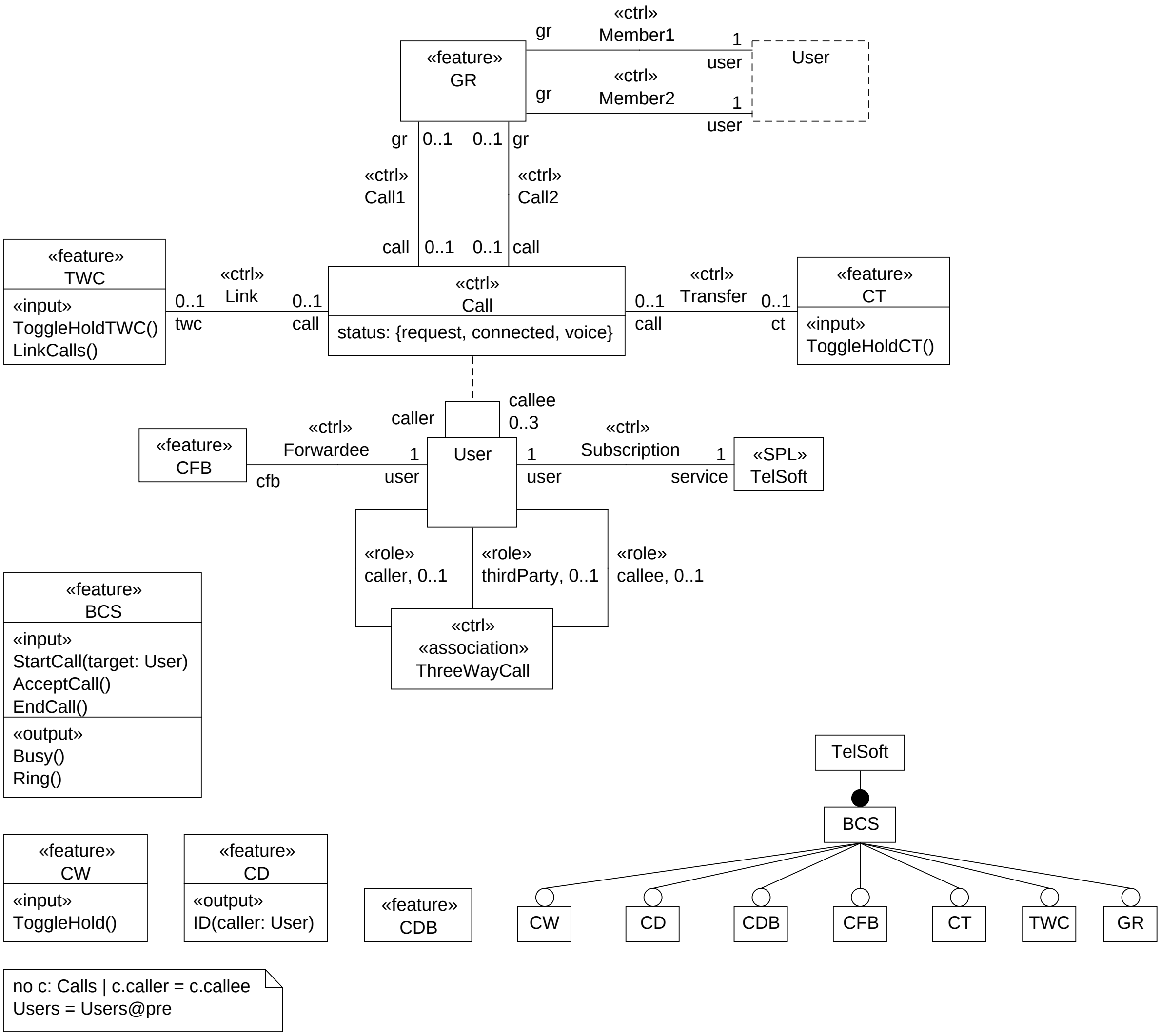


Figure A.16: The *TelSoft* world model as evolved by GR

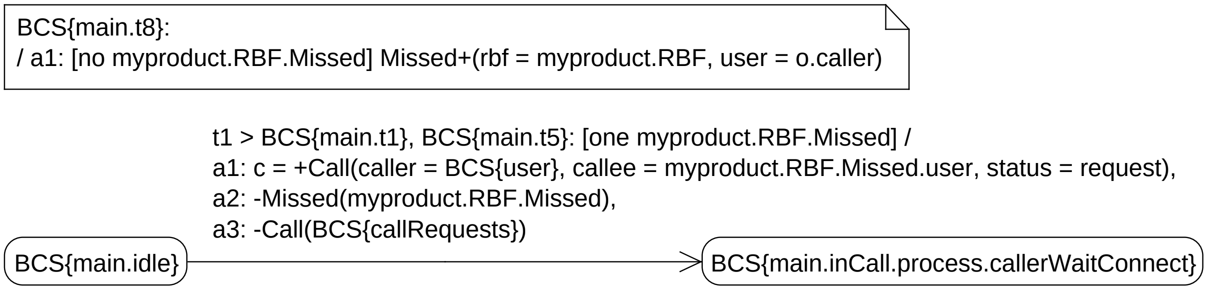

BCS{main.t8}:
/ a1: [no myproduct.RBF.Missed] Missed+(rbf = myproduct.RBF, user = o.caller)
t1 > BCS{main.t1}, BCS{main.t5}: [one myproduct.RBF.Missed] /
a1: c = +Call(caller = BCS{user}, callee = myproduct.RBF.Missed.user, status = request),
a2: -Missed(myproduct.RBF.Missed),
a3: -Call(BCS{callRequests})
BCS{main.idle}
BCS{main.inCall.process.callerWaitConnect}

Figure A.17: The feature module of RBF

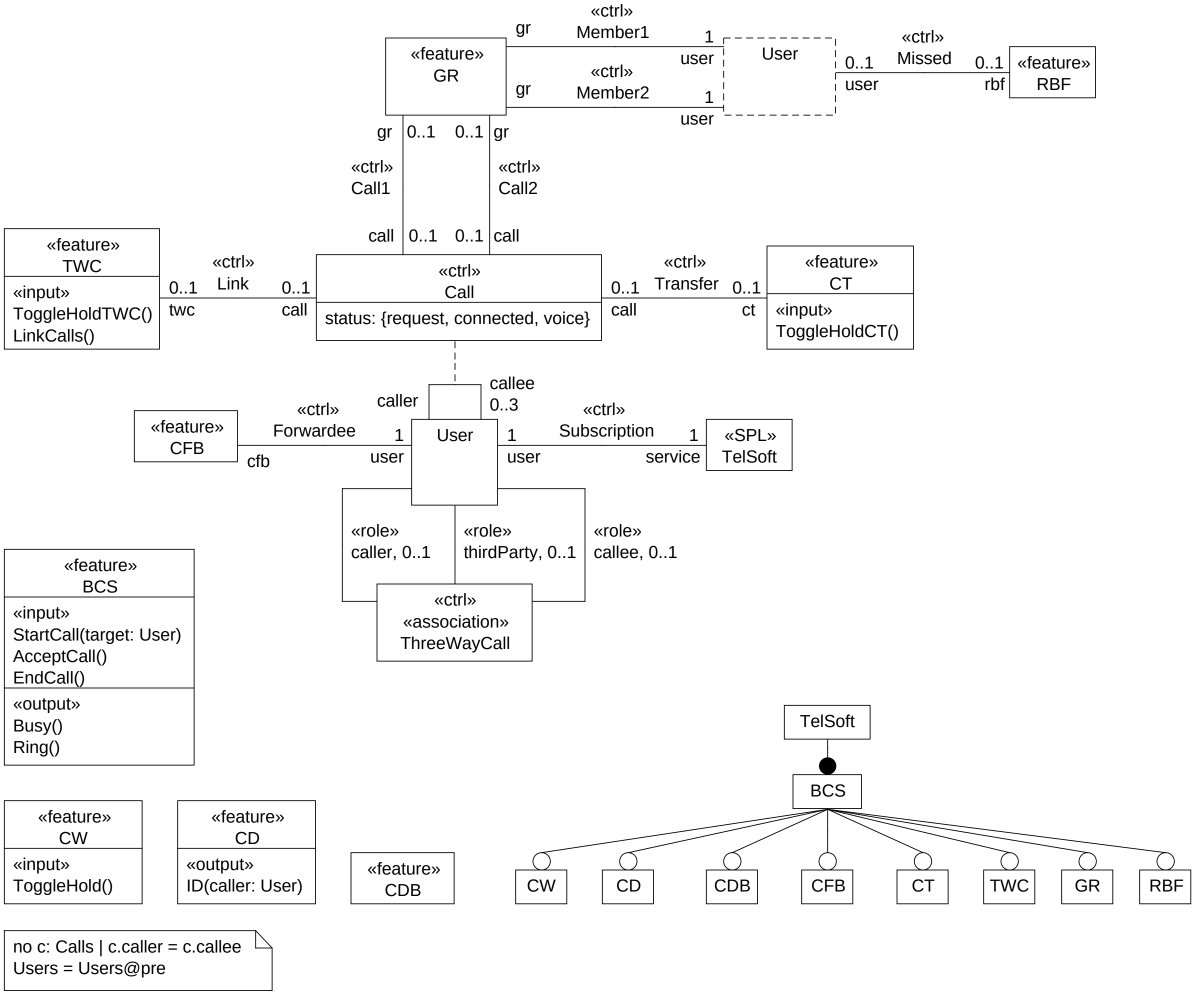


Figure A.18: The *TelSoft* world model as evolved by RBF

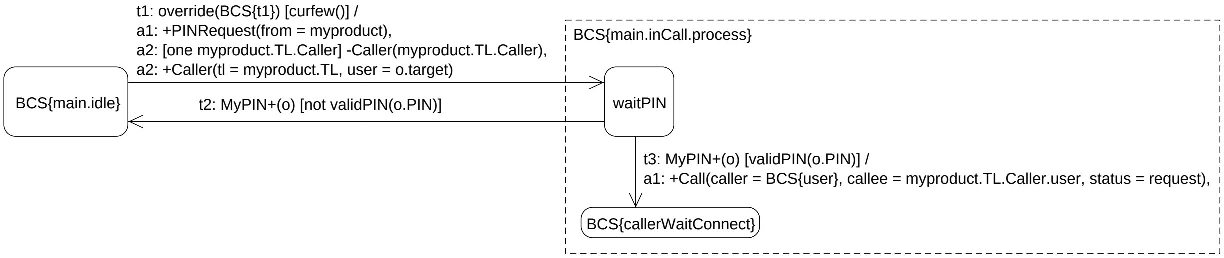

t1: override(BCS{t1}) [curfew()] /
a1: +PINRequest(from = myproduct),
a2: [one myproduct.TL.Caller] -Caller(myproduct.TL.Caller),
a2: +Caller(tl = myproduct.TL, user = o.target)
BCS{main.inCall.process}
BCS{main.idle}
waitPIN
t2: MyPIN+(o) [not validPIN(o.PIN)]
t3: MyPIN+(o) [validPIN(o.PIN)] /
a1: +Call(caller = BCS{user}, callee = myproduct.TL.Caller.user, status = request),
BCS{callerWaitConnect}

Figure A.19: The feature module of TL

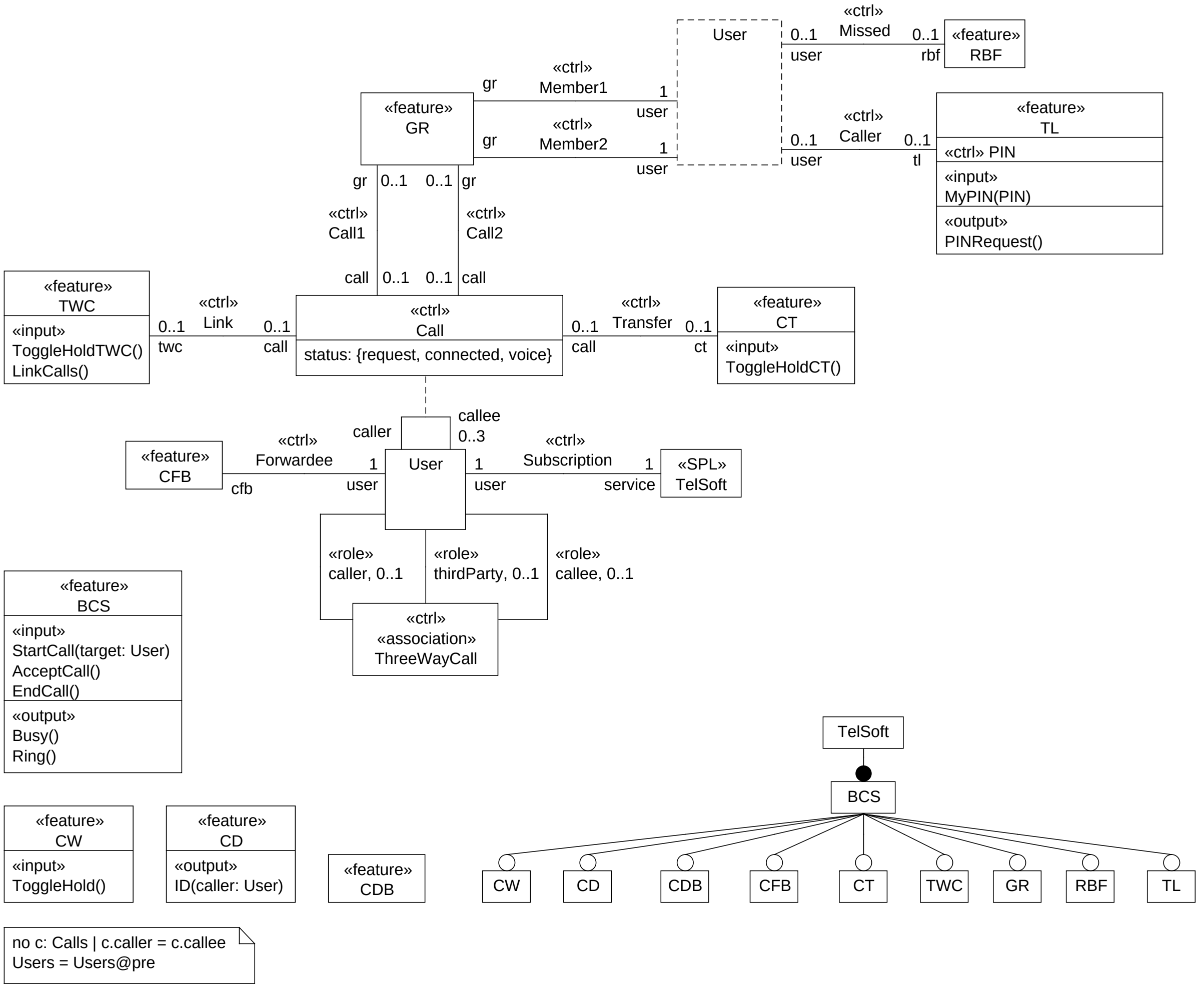
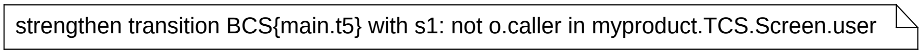


Figure A.20: The *TelSoft* world model as evolved by TL



```
strengthen transition BCS{main.t5} with s1: not o.caller in myproduct.TCS.Screen.user
```

Figure A.21: The feature module of TCS

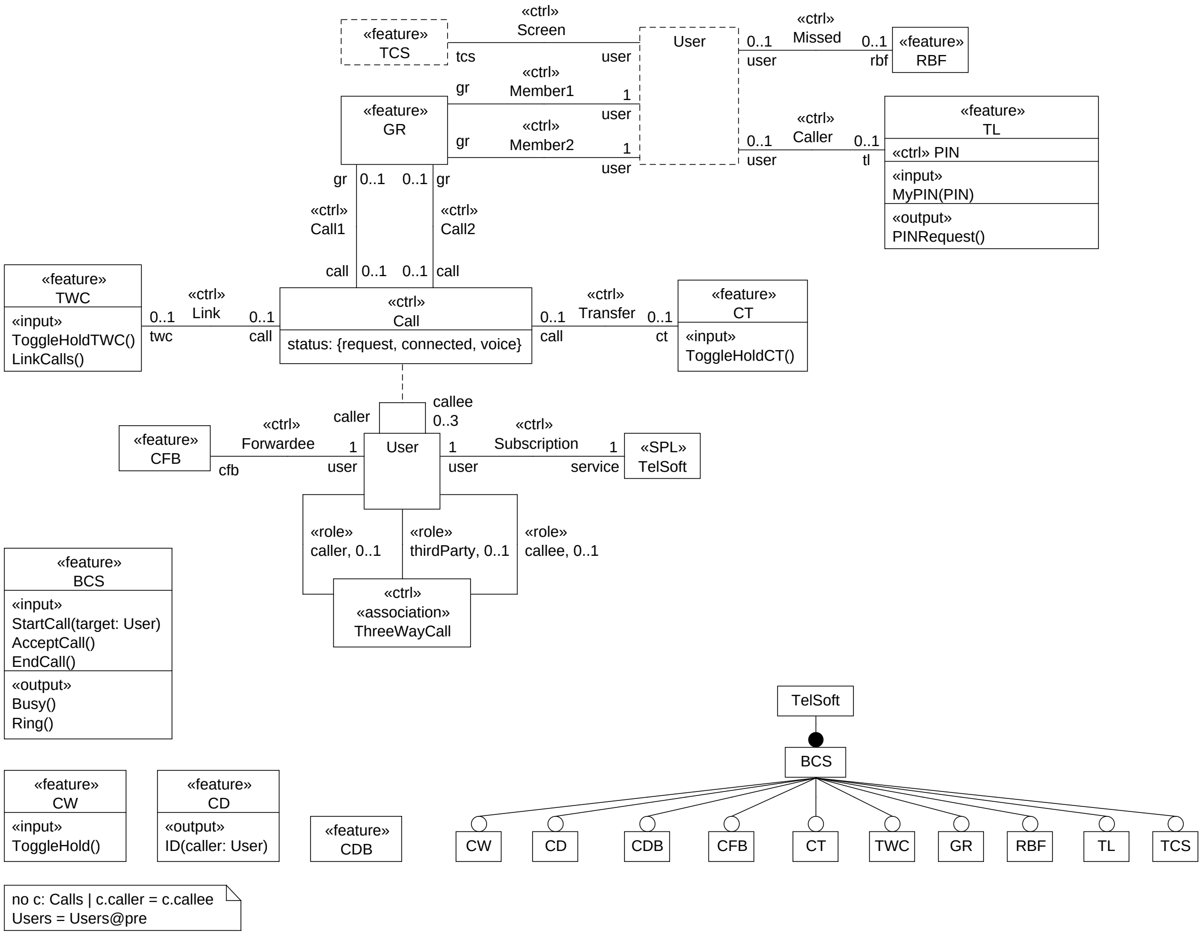


Figure A.22: The *TelSoft* world model as evolved by TCS

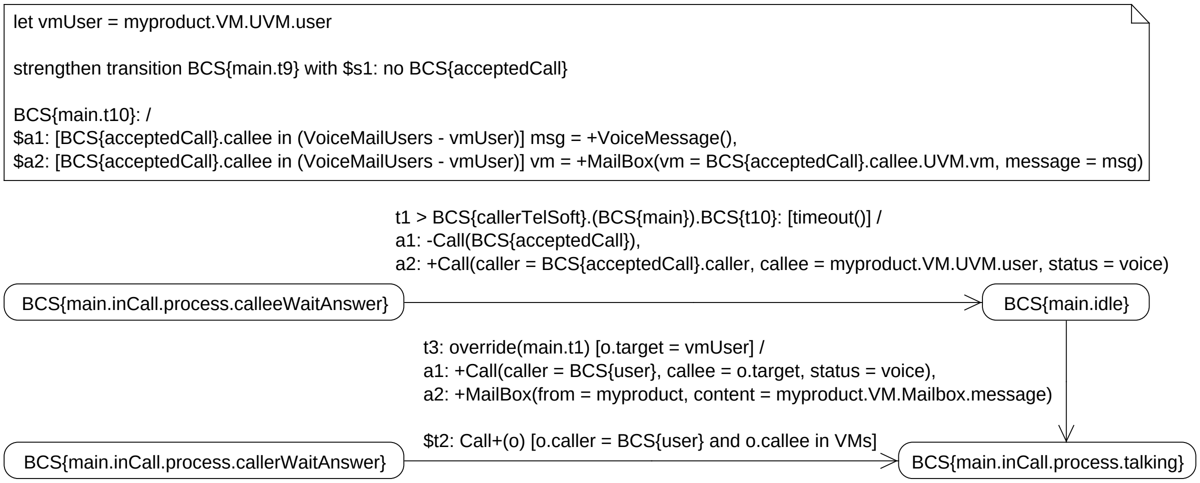

let vmUser = myproduct.VM.UVM.user
strengthen transition BCS{main.t9} with $s1: no BCS{acceptedCall}
BCS{main.t10}: /
$a1: [BCS{acceptedCall}.callee in (VoiceMailUsers - vmUser)] msg = +VoiceMessage(),
$a2: [BCS{acceptedCall}.callee in (VoiceMailUsers - vmUser)] vm = +MailBox(vm = BCS{acceptedCall}.callee.UVM.vm, message = msg)
t1 > BCS{callerTelSoft}.(BCS{main}).BCS{t10}: [timeout()] /
a1: -Call(BCS{acceptedCall}),
a2: +Call(caller = BCS{acceptedCall}.caller, callee = myproduct.VM.UVM.user, status = voice)
BCS{main.inCall.process.calleeWaitAnswer}
BCS{main.idle}
t3: override(main.t1) [o.target = vmUser] /
a1: +Call(caller = BCS{user}, callee = o.target, status = voice),
a2: +MailBox(from = myproduct, content = myproduct.VM.Mailbox.message)
$t2: Call+(o) [o.caller = BCS{user} and o.callee in VMs]
BCS{main.inCall.process.callerWaitAnswer}
BCS{main.inCall.process.talking}

Figure A.23: The feature module of VM

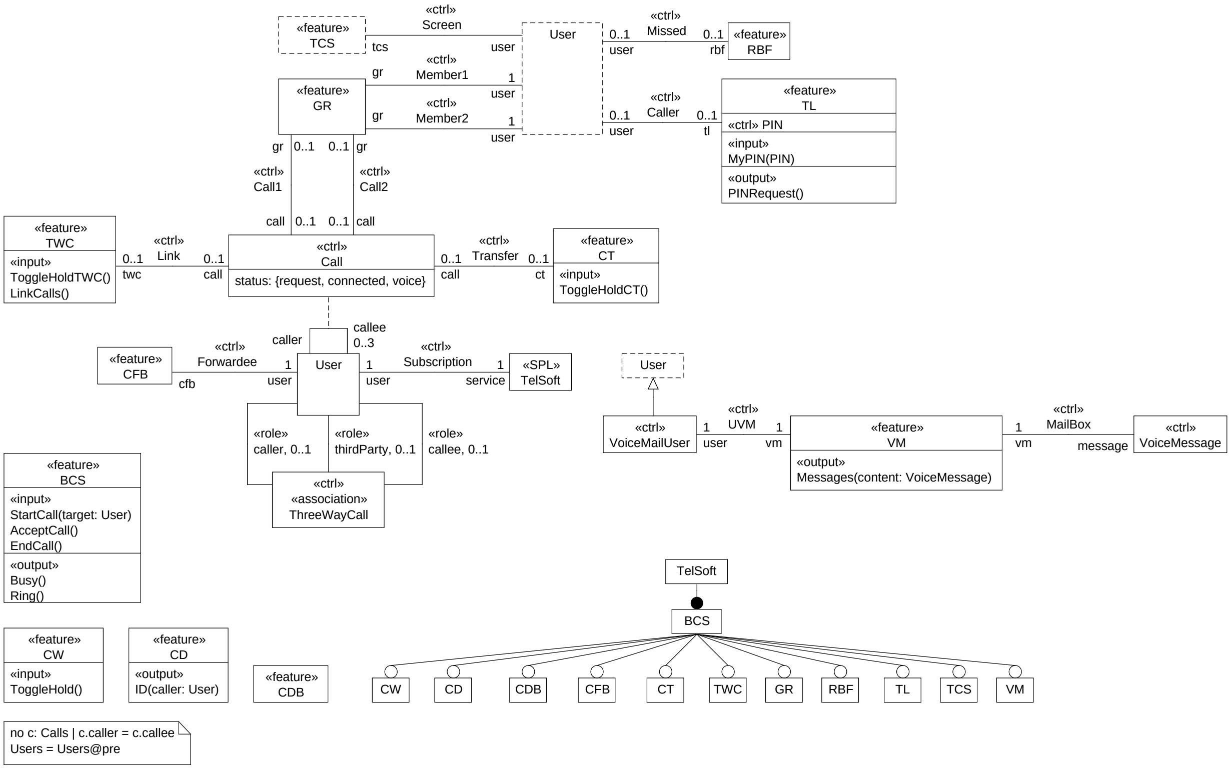
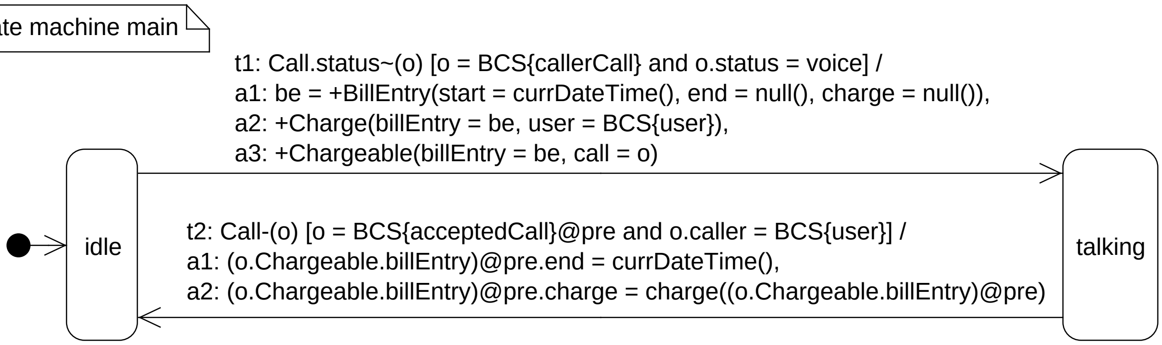


Figure A.24: The *TelSoft* world model as evolved by VM



t1: Call.status~(o) [o = BCS{callerCall} and o.status = voice] /
a1: be = +BillEntry(start = currDateTime(), end = null(), charge = null()),
a2: +Charge(billEntry = be, user = BCS{user}),
a3: +Chargeable(billEntry = be, call = o)

t2: Call-(o) [o = BCS{acceptedCall}@pre and o.caller = BCS{user}] /
a1: (o.Chargeable.billEntry)@pre.end = currDateTime(),
a2: (o.Chargeable.billEntry)@pre.charge = charge((o.Chargeable.billEntry)@pre)

Figure A.25: The feature module of Billing

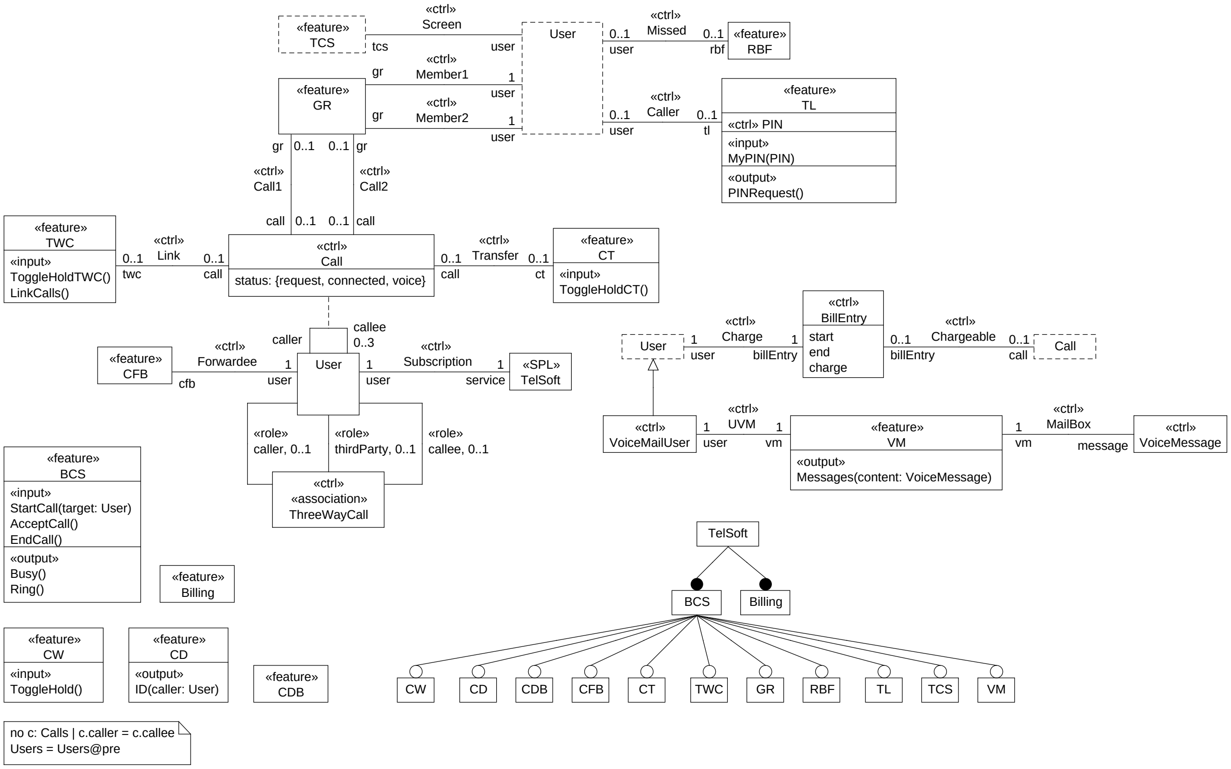


Figure A.26: The *TelSoft* world model as evolved by Billing

Billing{main.t1}: /
$a1 > Billing{a2}: [one o.callee.Subscription.service.RC] +Charge(BillEntry = be, User = o.callee)

Figure A.27: The feature module of RC

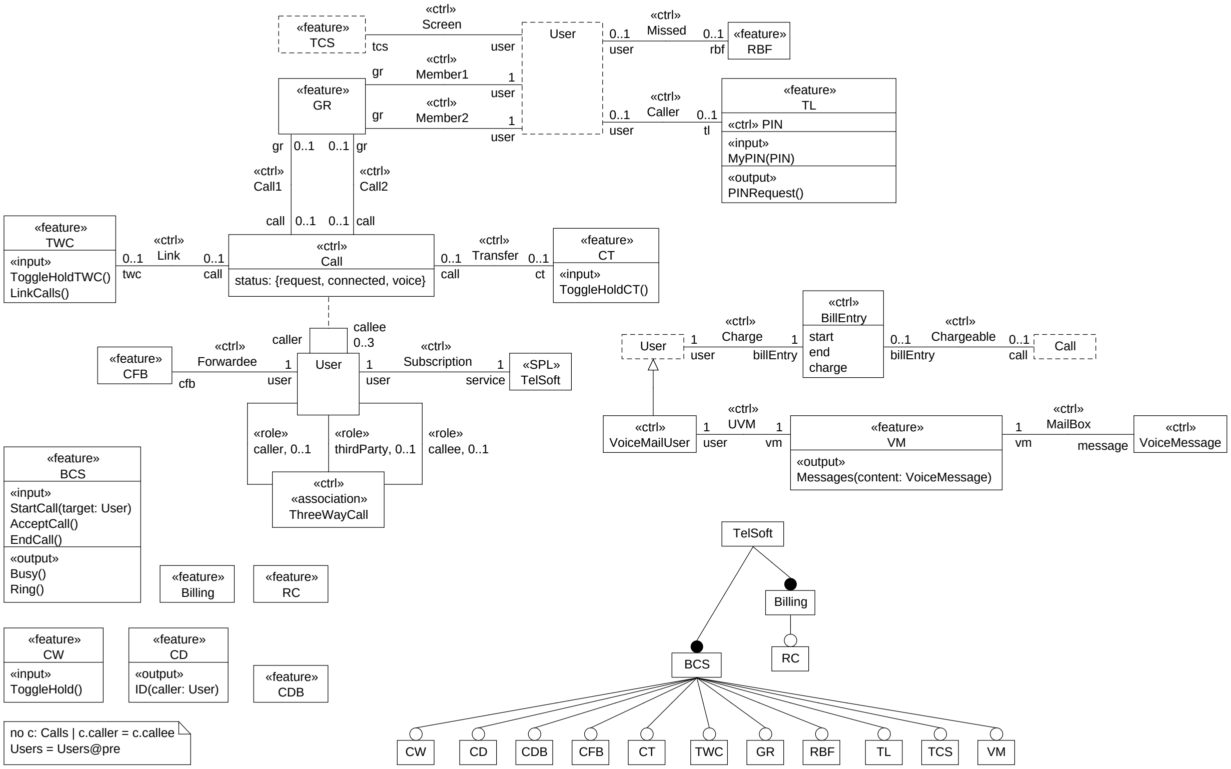


Figure A.28: The *TelSoft* world model as evolved by RC

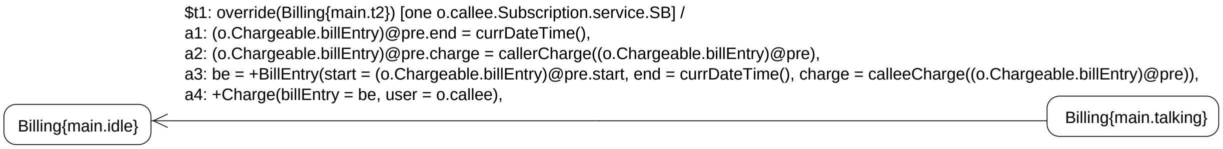

```
$t1: override(Billing{main.t2}) [one o.callee.Subscription.service.SB] /
a1: (o.Chargeable.billEntry)@pre.end = currDateTime(),
a2: (o.Chargeable.billEntry)@pre.charge = callerCharge((o.Chargeable.billEntry)@pre),
a3: be = +BillEntry(start = (o.Chargeable.billEntry)@pre.start, end = currDateTime(), charge = calleeCharge((o.Chargeable.billEntry)@pre)),
a4: +Charge(billEntry = be, user = o.callee),
```

Figure A.29: The feature module of SB

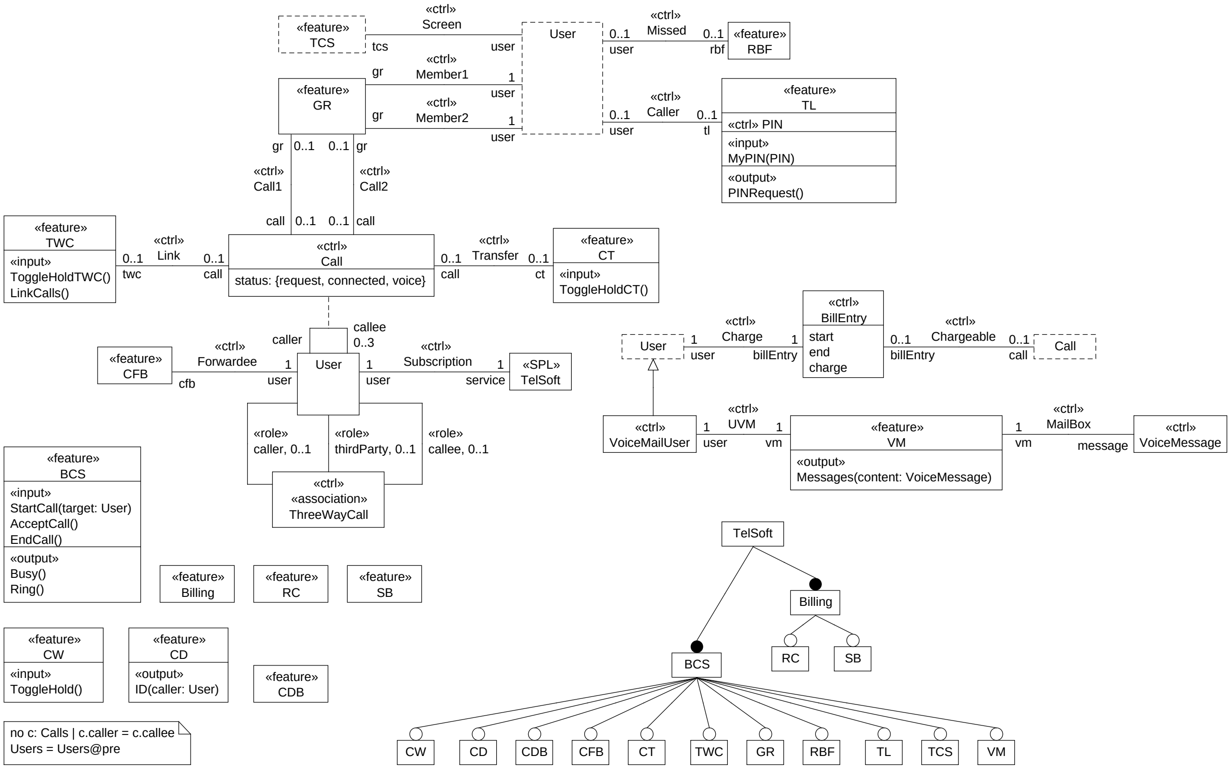


Figure A.30: The *TelSoft* world model as evolved by SB

A.2 Automotive Case Study

Figures A.31 to A.52 show, in order, the evolution steps of a FORML model of *AutoSoft* .

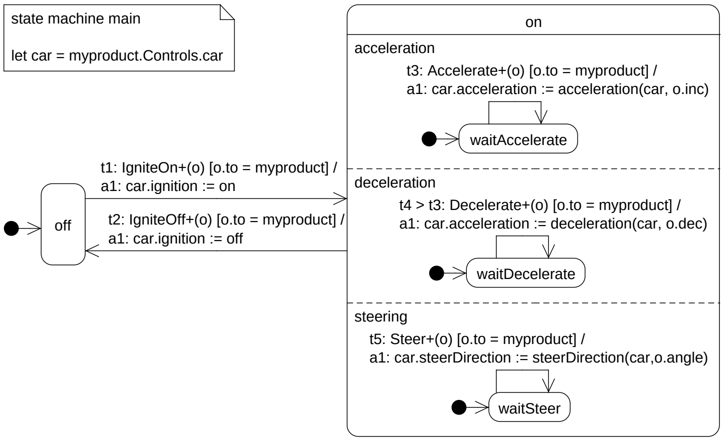

state machine main
let car = myproduct.Controls.car
on
acceleration
t3: Accelerate+(o) [o.to = myproduct] /
a1: car.acceleration := acceleration(car, o.inc)
waitAccelerate
t1: IgniteOn+(o) [o.to = myproduct] /
a1: car.ignition := on
off
t2: IgniteOff+(o) [o.to = myproduct] /
a1: car.ignition := off
deceleration
t4 > t3: Decelerate+(o) [o.to = myproduct] /
a1: car.acceleration := deceleration(car, o.dec)
waitDecelerate
steering
t5: Steer+(o) [o.to = myproduct] /
a1: car.steerDirection := steerDirection(car,o.angle)
waitSteer

Figure A.31: The feature module of BDS

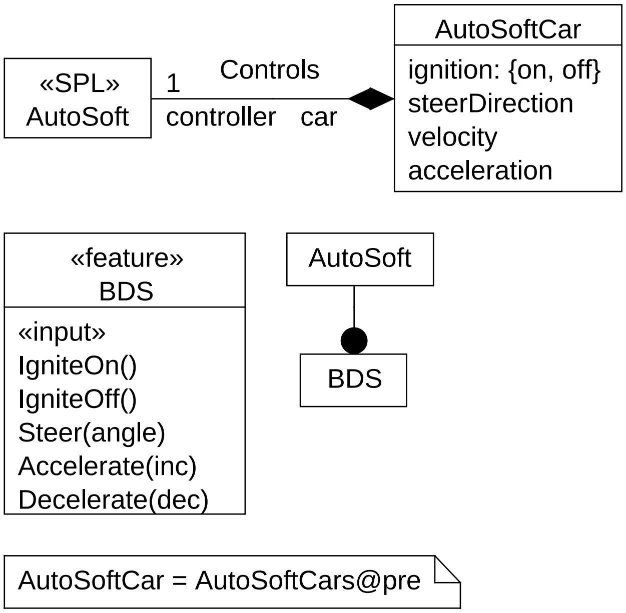


Figure A.32: The *AutoSoft* world model as evolved by BDS

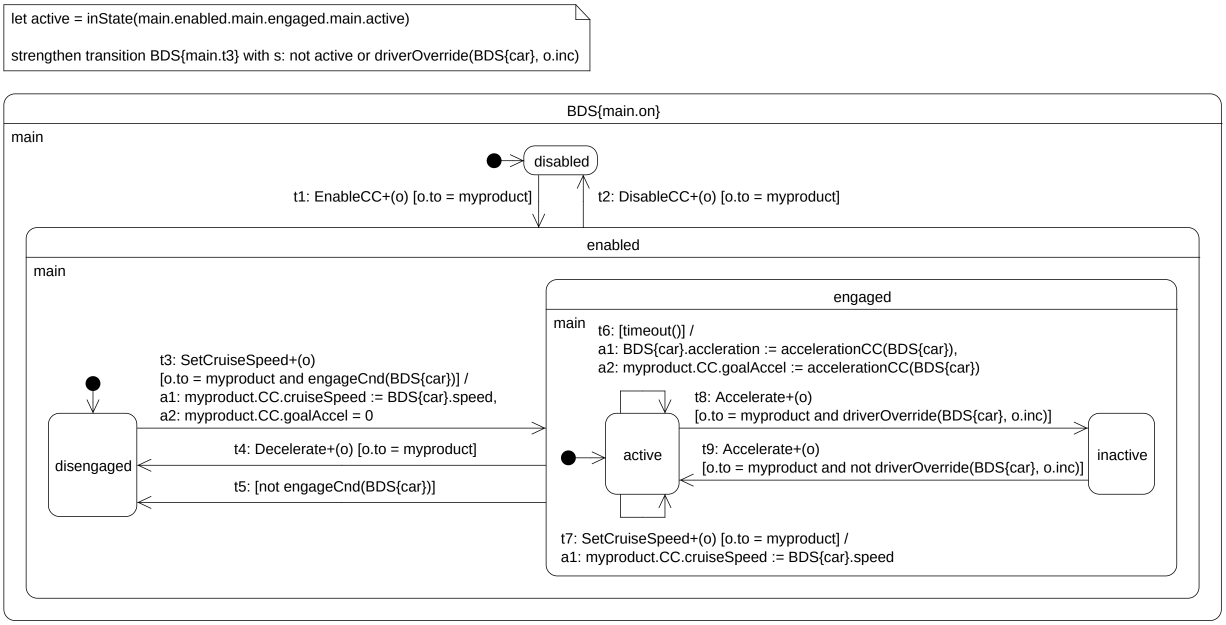

let active = inState(main.enabled.main.engaged.main.active)
strengthen transition BDS{main.t3} with s: not active or driverOverride(BDS{car}, o.inc)
BDS{main.on}
main
disabled
t1: EnableCC+(o) [o.to = myproduct]
t2: DisableCC+(o) [o.to = myproduct]
enabled
main
engaged
main
t6: [timeout()] /
a1: BDS{car}.accleration := accelerationCC(BDS{car}),
a2: myproduct.CC.goalAccel := accelerationCC(BDS{car})
t3: SetCruiseSpeed+(o)
[o.to = myproduct and engageCnd(BDS{car})] /
a1: myproduct.CC.cruiseSpeed := BDS{car}.speed,
a2: myproduct.CC.goalAccel = 0
t8: Accelerate+(o)
[o.to = myproduct and driverOverride(BDS{car}, o.inc)]
disengaged
t4: Decelerate+(o) [o.to = myproduct]
active
t9: Accelerate+(o)
[o.to = myproduct and not driverOverride(BDS{car}, o.inc)]
inactive
t5: [not engageCnd(BDS{car})]
t7: SetCruiseSpeed+(o) [o.to = myproduct] /
a1: myproduct.CC.cruiseSpeed := BDS{car}.speed

Figure A.33: The feature module of CC

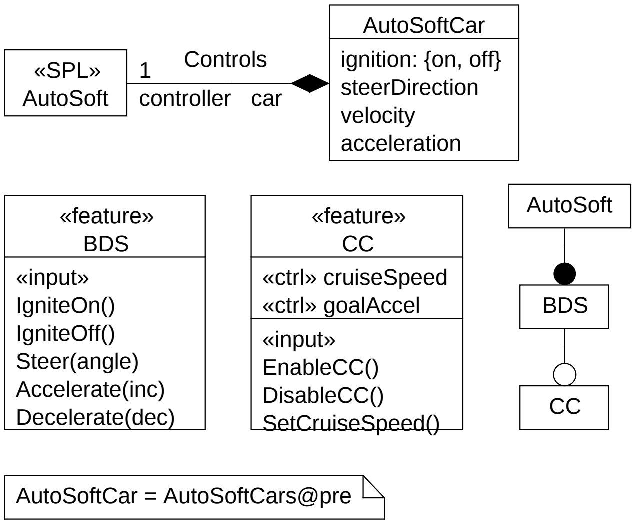


Figure A.34: The *AutoSoft* world model as evolved by CC

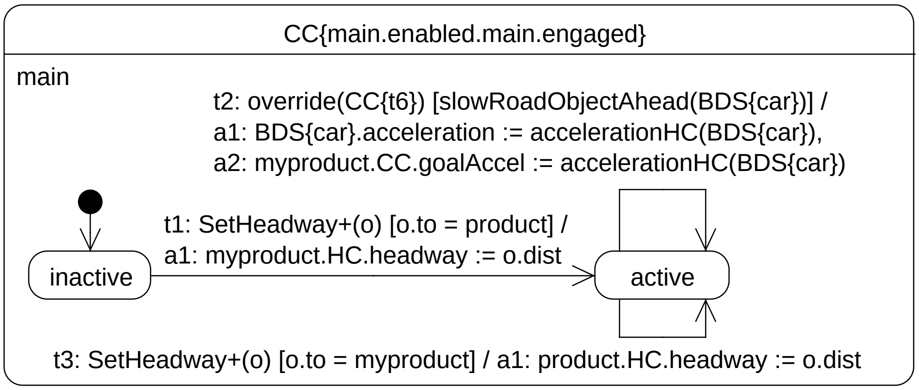

CC{main.enabled.main.engaged}
main
t2: override(CC{t6}) [slowRoadObjectAhead(BDS{car})] /
a1: BDS{car}.acceleration := accelerationHC(BDS{car}),
a2: myproduct.CC.goalAccel := accelerationHC(BDS{car})
t1: SetHeadway+(o) [o.to = product] /
a1: myproduct.HC.headway := o.dist
inactive
active
t3: SetHeadway+(o) [o.to = myproduct] / a1: product.HC.headway := o.dist

Figure A.35: The feature module of HC

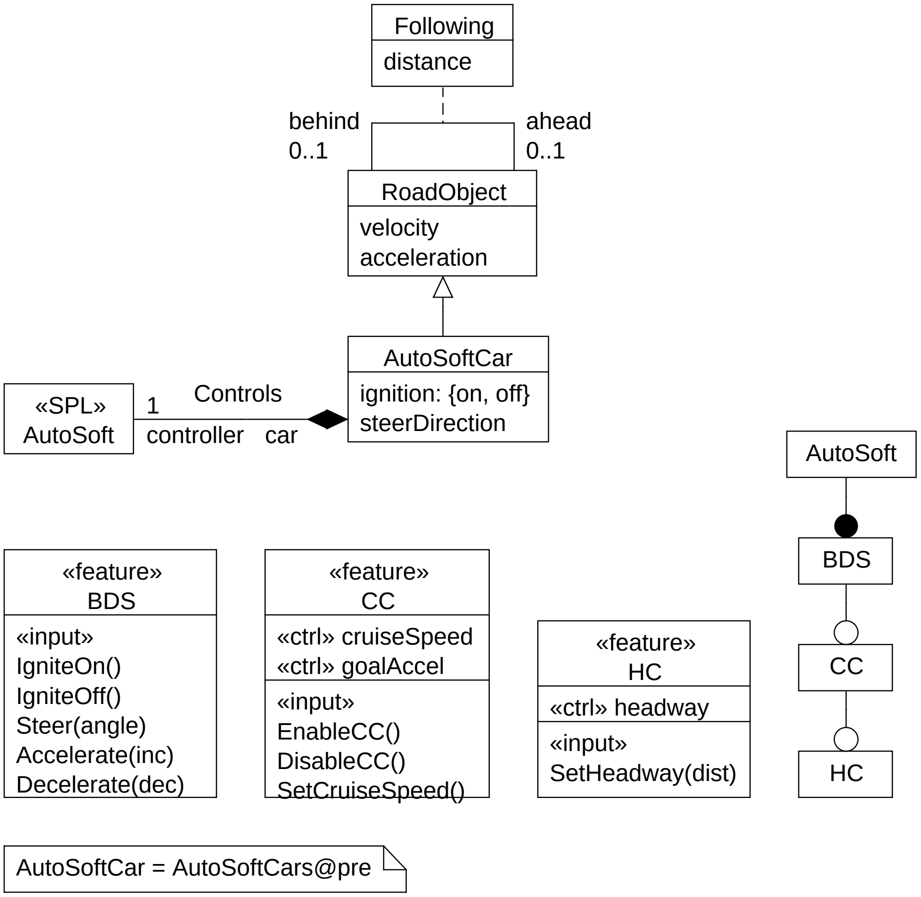


Figure A.36: The *AutoSoft* world model as evolved by HC

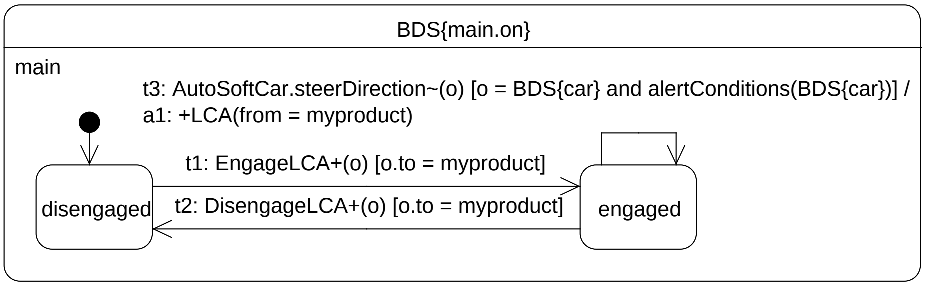

BDS{main.on}
main
t3: AutoSoftCar.steerDirection~(o) [o = BDS{car} and alertConditions(BDS{car})] /
a1: +LCA(from = myproduct)
t1: EngageLCA+(o) [o.to = myproduct]
t2: DisengageLCA+(o) [o.to = myproduct]
disengaged
engaged

Figure A.37: The feature module of LCA

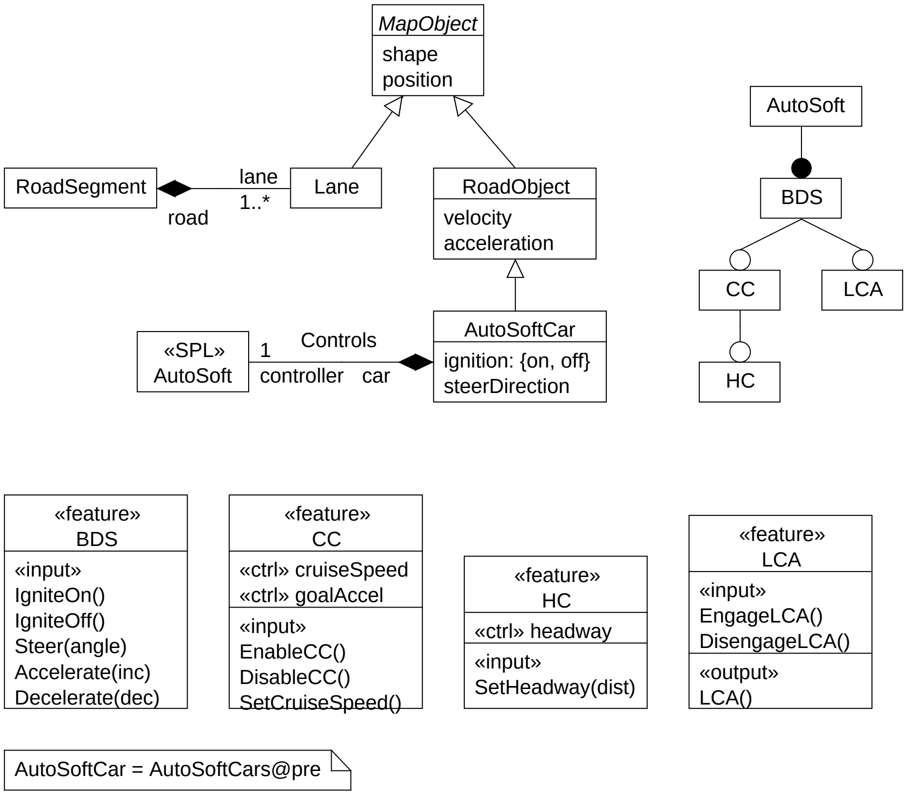


Figure A.38: The *AutoSoft* world model as evolved by LCA

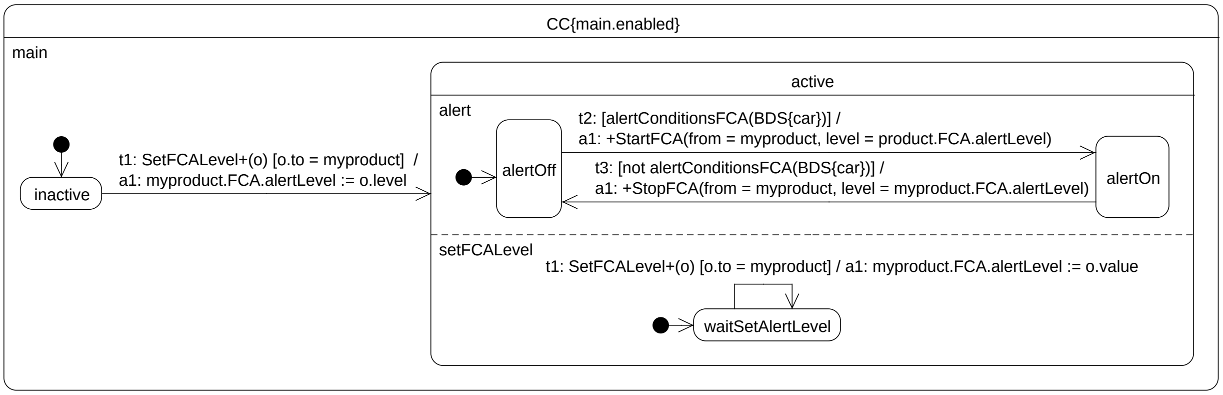

CC{main.enabled}
main
inactive
t1: SetFCALevel+(o) [o.to = myproduct] /
a1: myproduct.FCA.alertLevel := o.level
active
alert
alertOff
t2: [alertConditionsFCA(BDS{car})] /
a1: +StartFCA(from = myproduct, level = product.FCA.alertLevel)
t3: [not alertConditionsFCA(BDS{car})] /
a1: +StopFCA(from = myproduct, level = myproduct.FCA.alertLevel)
alertOn
setFCALevel
t1: SetFCALevel+(o) [o.to = myproduct] / a1: myproduct.FCA.alertLevel := o.value
waitSetAlertLevel

Figure A.39: The feature module of FCA

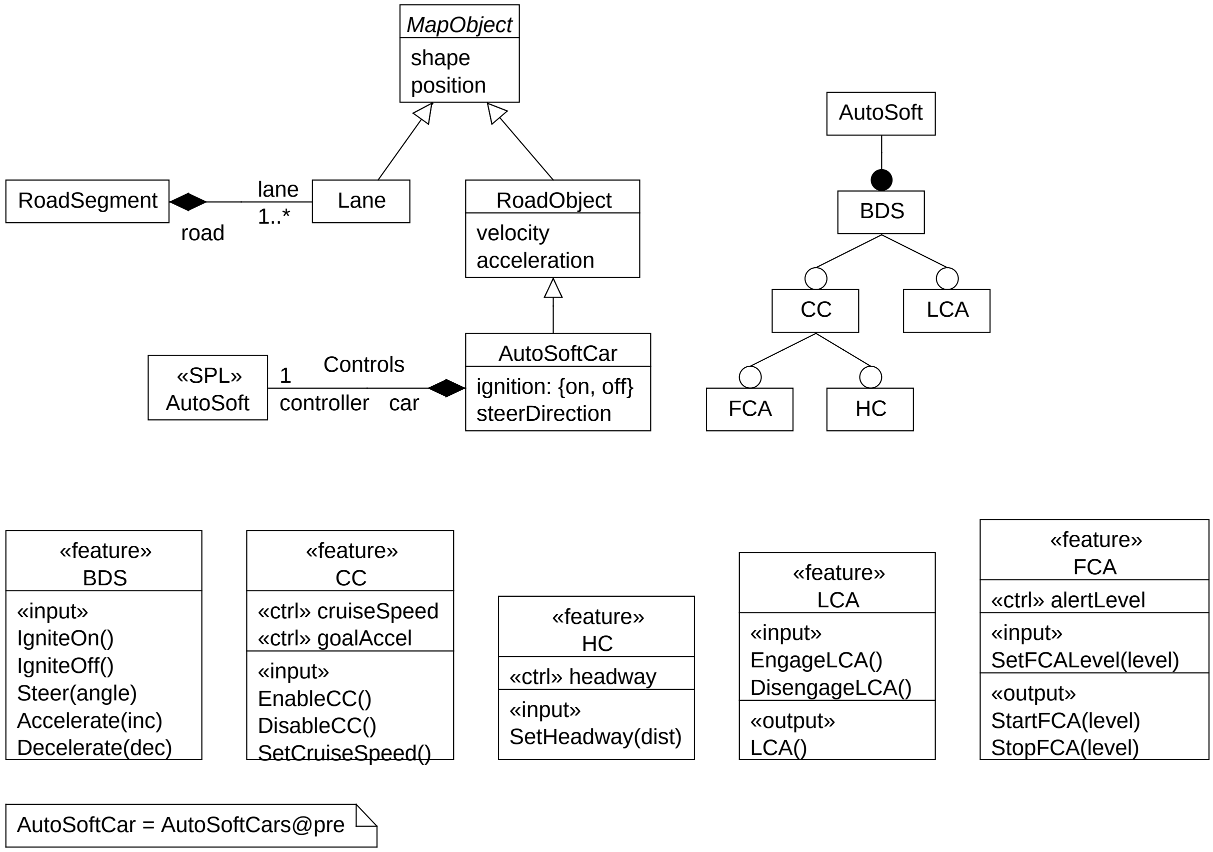
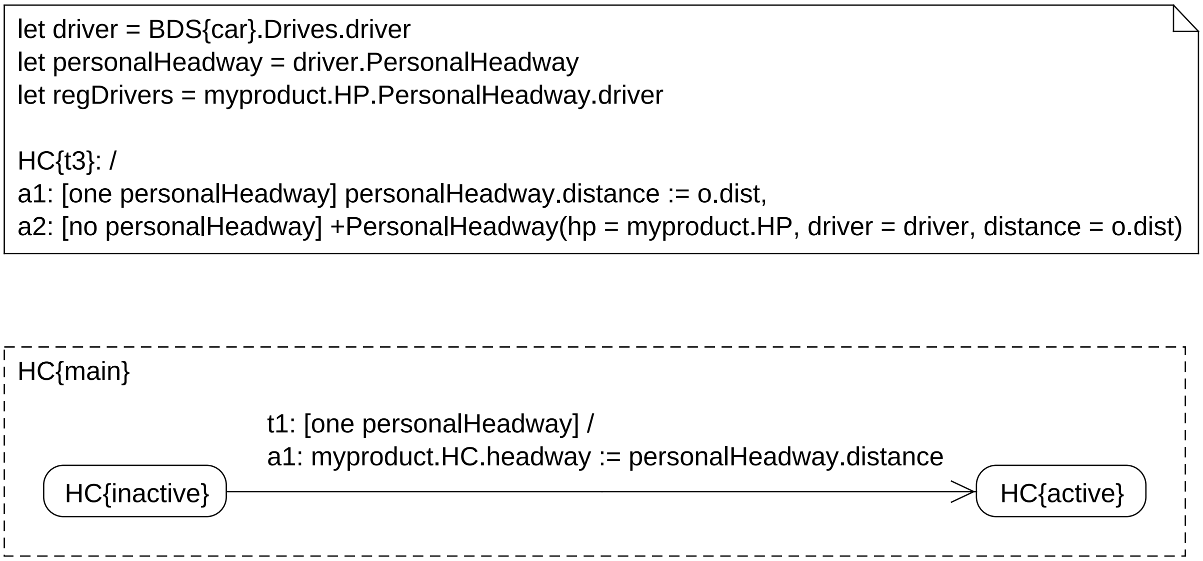


Figure A.40: The *AutoSoft* world model as evolved by FCA



```
let driver = BDS{car}.Drives.driver
let personalHeadway = driver.PersonalHeadway
let regDrivers = myproduct.HP.PersonalHeadway.driver

HC{t3}: /
a1: [one personalHeadway] personalHeadway.distance := o.dist,
a2: [no personalHeadway] +PersonalHeadway(hp = myproduct.HP, driver = driver, distance = o.dist)
```

HC{main}
t1: [one personalHeadway] /
a1: myproduct.HC.headway := personalHeadway.distance
HC{inactive}
HC{active}

Figure A.41: The feature module of HP

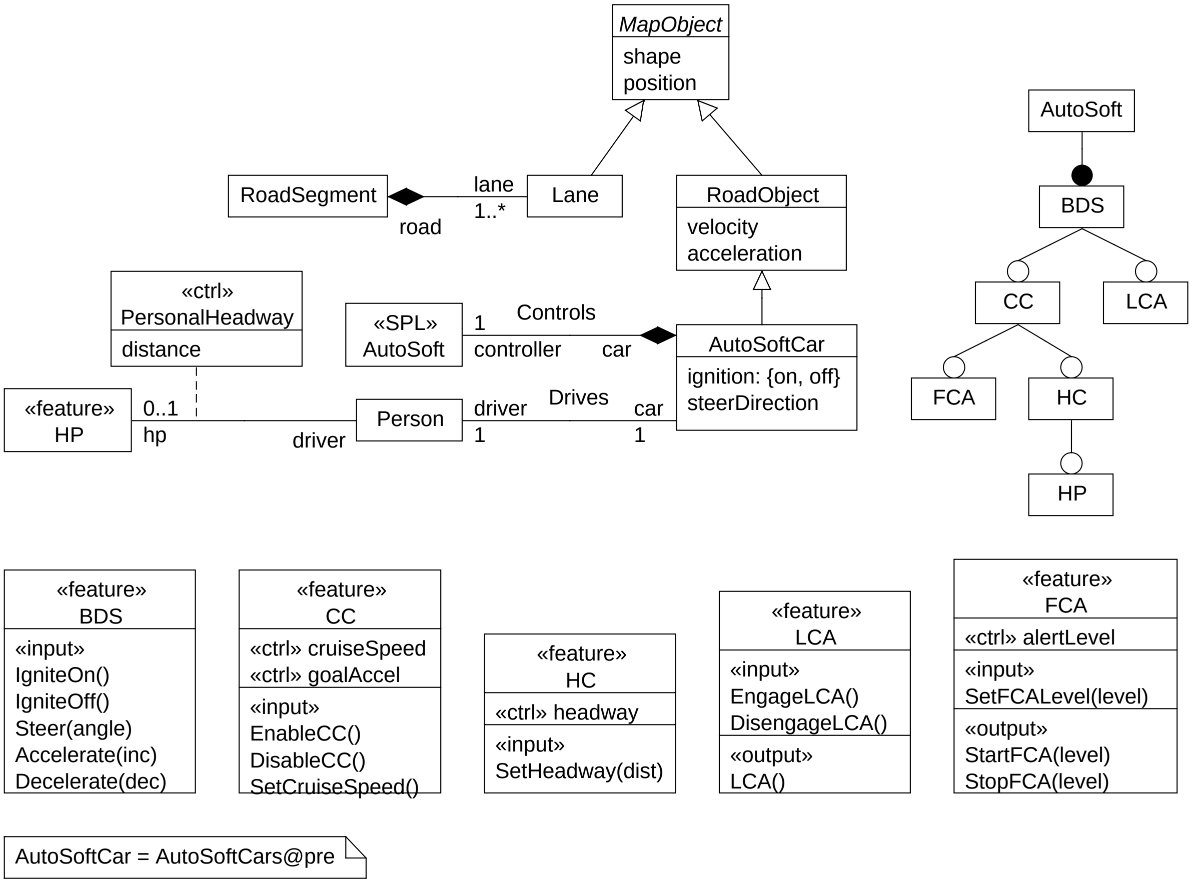


Figure A.42: The *AutoSoft* world model as evolved by HP

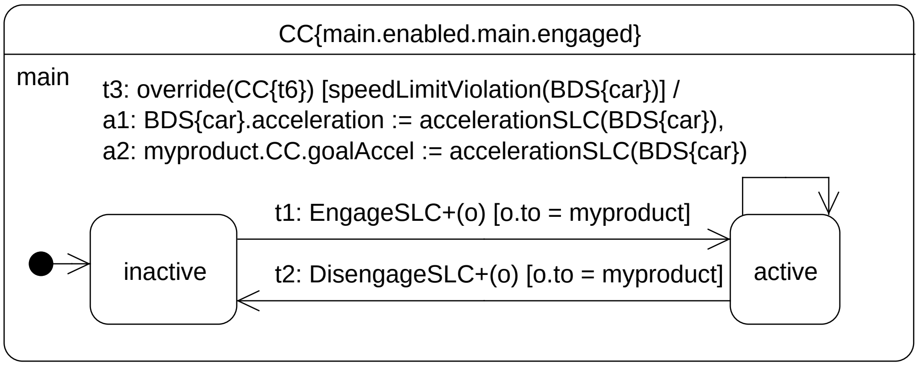

CC{main.enabled.main.engaged}
main
t3: override(CC{t6}) [speedLimitViolation(BDS{car})] /
a1: BDS{car}.acceleration := accelerationSLC(BDS{car}),
a2: myproduct.CC.goalAccel := accelerationSLC(BDS{car})
t1: EngageSLC+(o) [o.to = myproduct]
inactive
t2: DisengageSLC+(o) [o.to = myproduct]
active

Figure A.43: The feature module of SLC

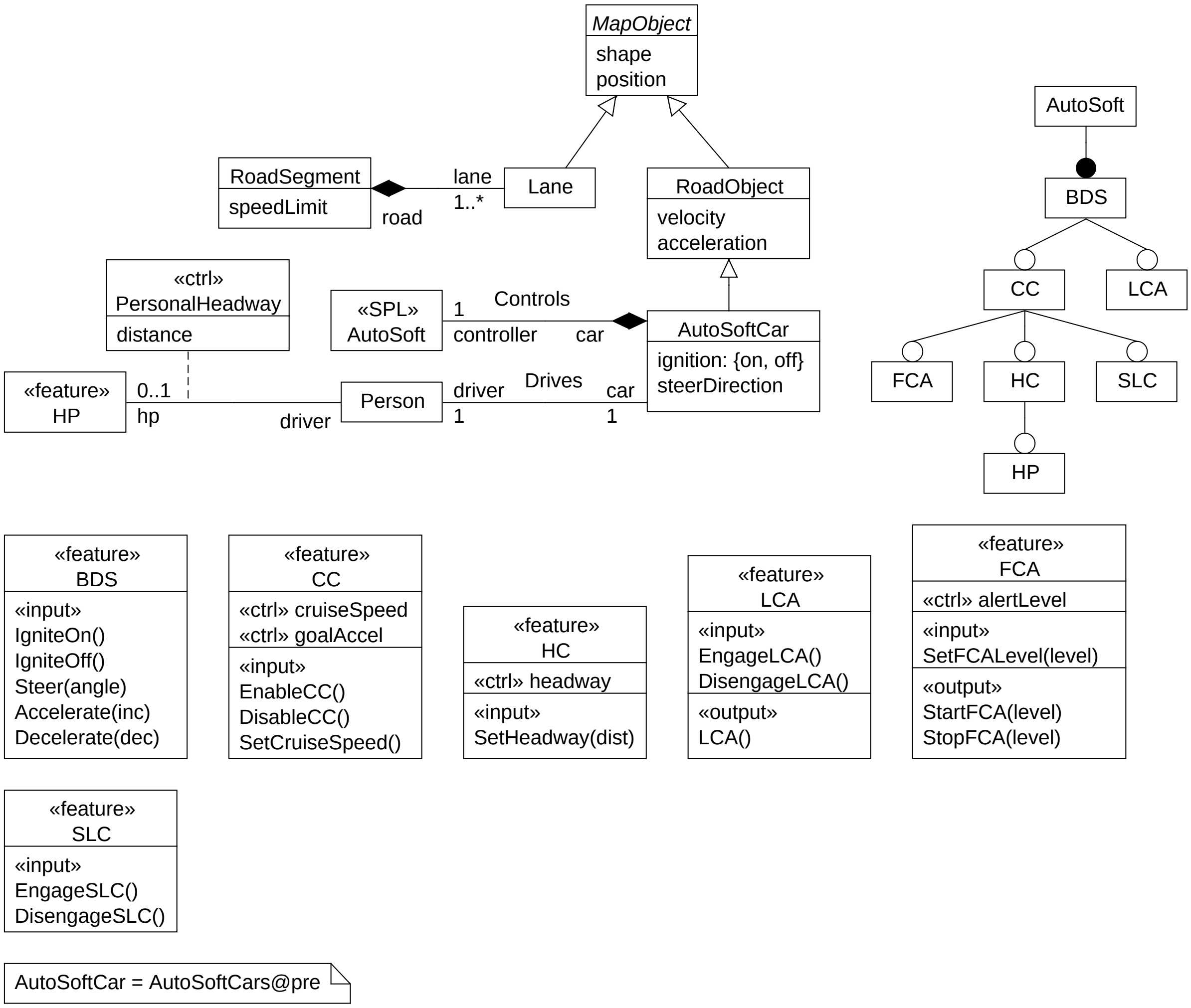


Figure A.44: The *AutoSoft* world model as evolved by SLC

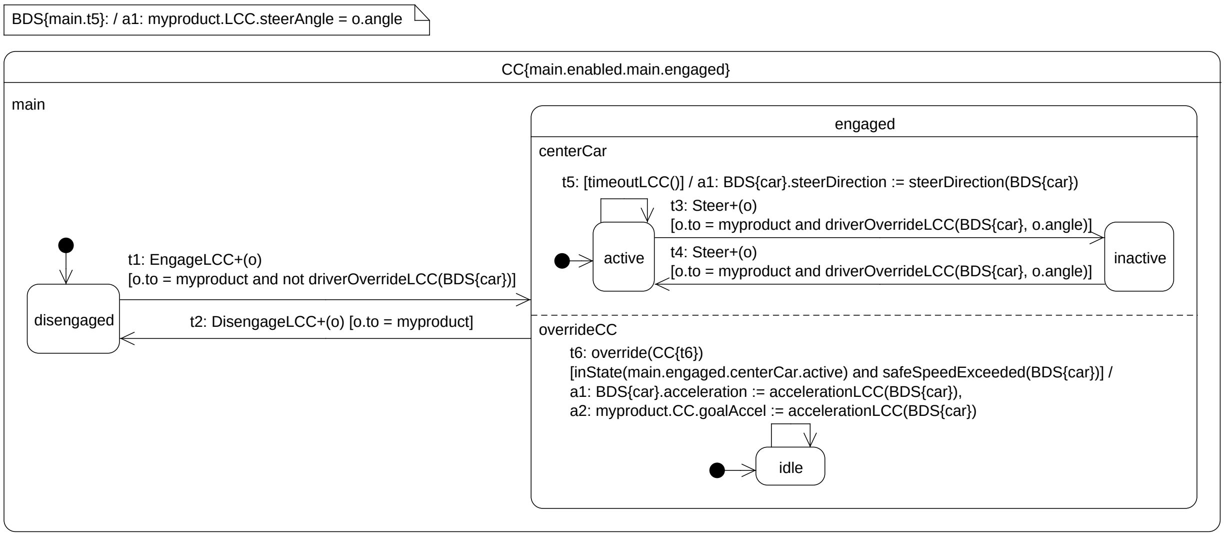

BDS{main.t5}: / a1: myproduct.LCC.steerAngle = o.angle
CC{main.enabled.main.engaged}
main
engaged
centerCar
t5: [timeoutLCC()] / a1: BDS{car}.steerDirection := steerDirection(BDS{car})
t3: Steer+(o)
[o.to = myproduct and driverOverrideLCC(BDS{car}, o.angle)]
active
t4: Steer+(o)
[o.to = myproduct and driverOverrideLCC(BDS{car}, o.angle)]
inactive
t1: EngageLCC+(o)
[o.to = myproduct and not driverOverrideLCC(BDS{car})]
disengaged
t2: DisengageLCC+(o) [o.to = myproduct]
overrideCC
t6: override(CC{t6})
[inState(main.engaged.centerCar.active) and safeSpeedExceeded(BDS{car})] /
a1: BDS{car}.acceleration := accelerationLCC(BDS{car}),
a2: myproduct.CC.goalAccel := accelerationLCC(BDS{car})
idle

Figure A.45: The feature module of LCC

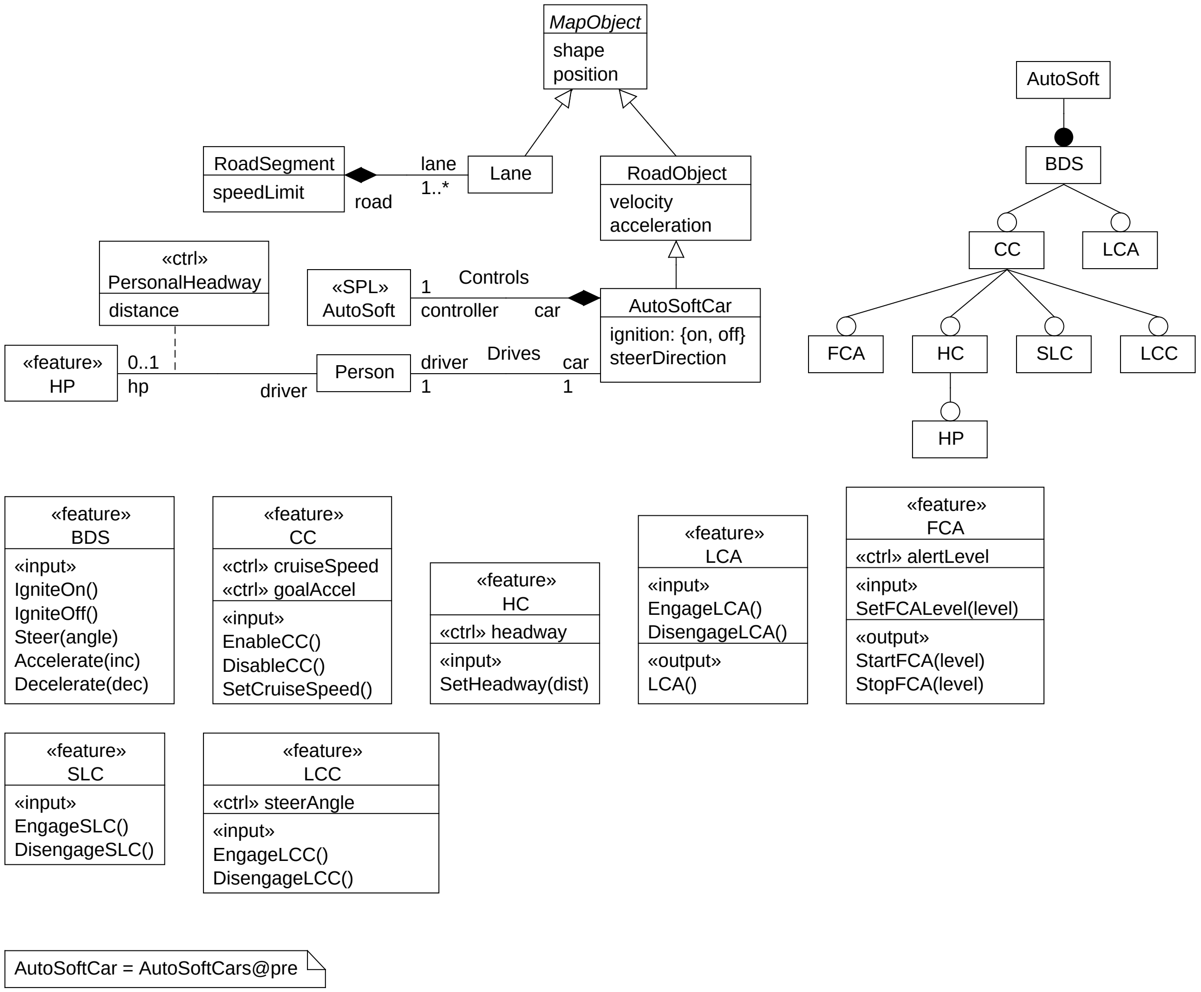


Figure A.46: The *AutoSoft* world model as evolved by LCC

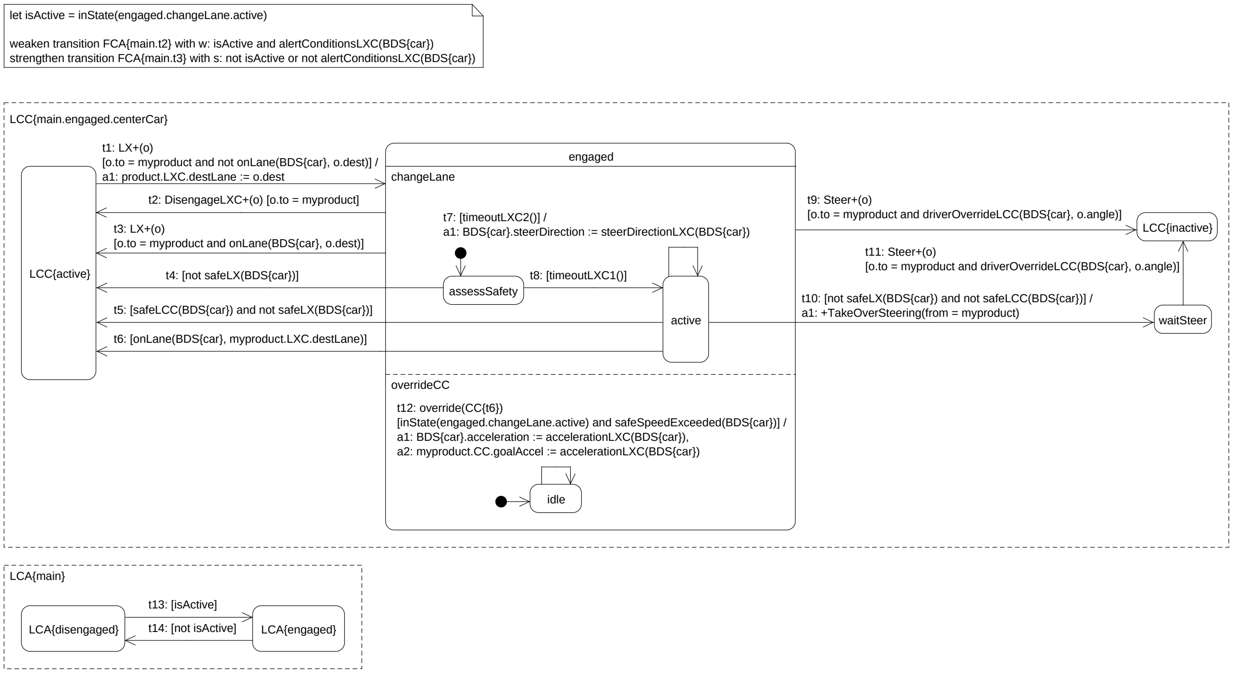

let isActive = inState(engaged.changeLane.active)
weaken transition FCA{main.t2} with w: isActive and alertConditionsLXC(BDS{car})
strengthen transition FCA{main.t3} with s: not isActive or not alertConditionsLXC(BDS{car})
LCC{main.engaged.centerCar}
t1: LX+(o)
[o.to = myproduct and not onLane(BDS{car}, o.dest)] /
a1: product.LXC.destLane := o.dest
engaged
changeLane
t2: DisengageLXC+(o) [o.to = myproduct]
t3: LX+(o)
[o.to = myproduct and onLane(BDS{car}, o.dest)]
LCC{active}
t4: [not safeLX(BDS{car})]
t5: [safeLCC(BDS{car}) and not safeLX(BDS{car})]
t6: [onLane(BDS{car}, myproduct.LXC.destLane)]
t7: [timeoutLXC2()] /
a1: BDS{car}.steerDirection := steerDirectionLXC(BDS{car})
assessSafety
t8: [timeoutLXC1()]
active
t9: Steer+(o)
[o.to = myproduct and driverOverrideLCC(BDS{car}, o.angle)]
LCC{inactive}
t11: Steer+(o)
[o.to = myproduct and driverOverrideLCC(BDS{car}, o.angle)]
t10: [not safeLX(BDS{car}) and not safeLCC(BDS{car})] /
a1: +TakeOverSteering(from = myproduct)
waitSteer
overrideCC
t12: override(CC{t6})
[inState(engaged.changeLane.active) and safeSpeedExceeded(BDS{car})] /
a1: BDS{car}.acceleration := accelerationLXC(BDS{car}),
a2: myproduct.CC.goalAccel := accelerationLXC(BDS{car})
idle
LCA{main}
t13: [isActive]
t14: [not isActive]
LCA{disengaged}
LCA{engaged}

Figure A.47: The feature module of LXC

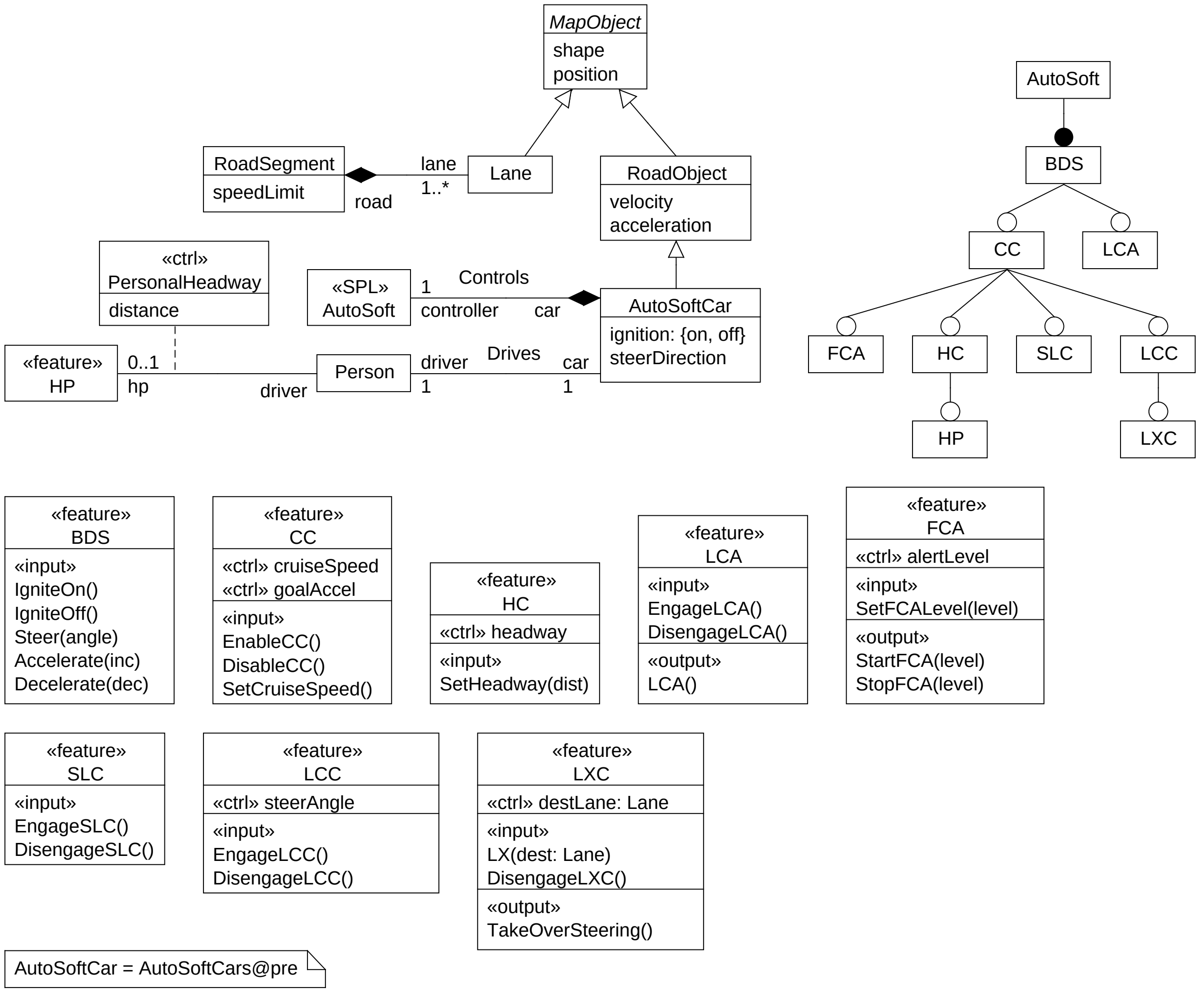


Figure A.48: The *AutoSoft* world model as evolved by LXC

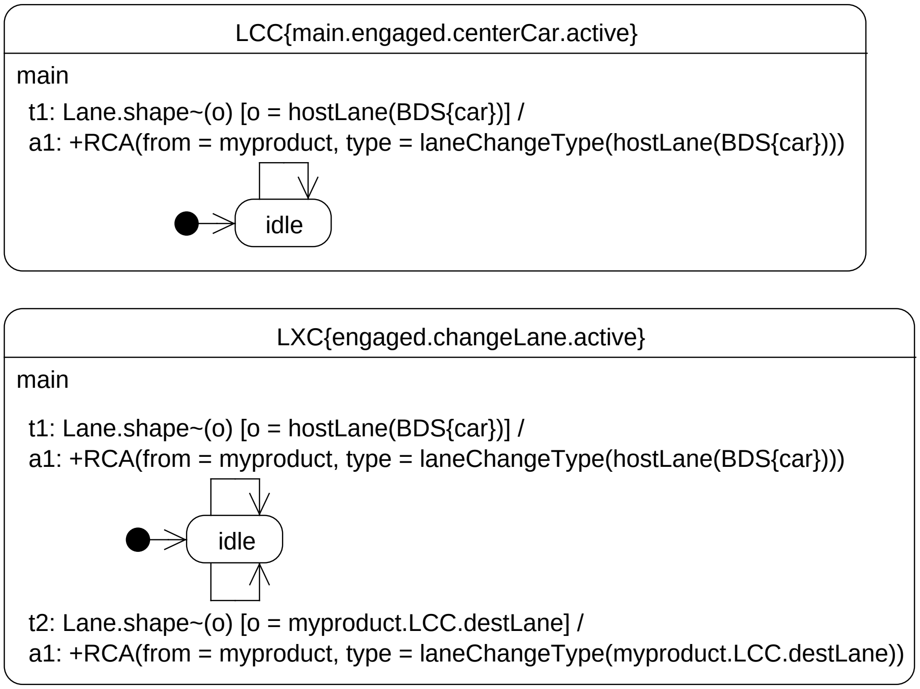

LCC{main.engaged.centerCar.active}
main
t1: Lane.shape~(o) [o = hostLane(BDS{car})] /
a1: +RCA(from = myproduct, type = laneChangeType(hostLane(BDS{car})))
idle
LXC{engaged.changeLane.active}
main
t1: Lane.shape~(o) [o = hostLane(BDS{car})] /
a1: +RCA(from = myproduct, type = laneChangeType(hostLane(BDS{car})))
idle
t2: Lane.shape~(o) [o = myproduct.LCC.destLane] /
a1: +RCA(from = myproduct, type = laneChangeType(myproduct.LCC.destLane))

Figure A.49: The feature module of RCA

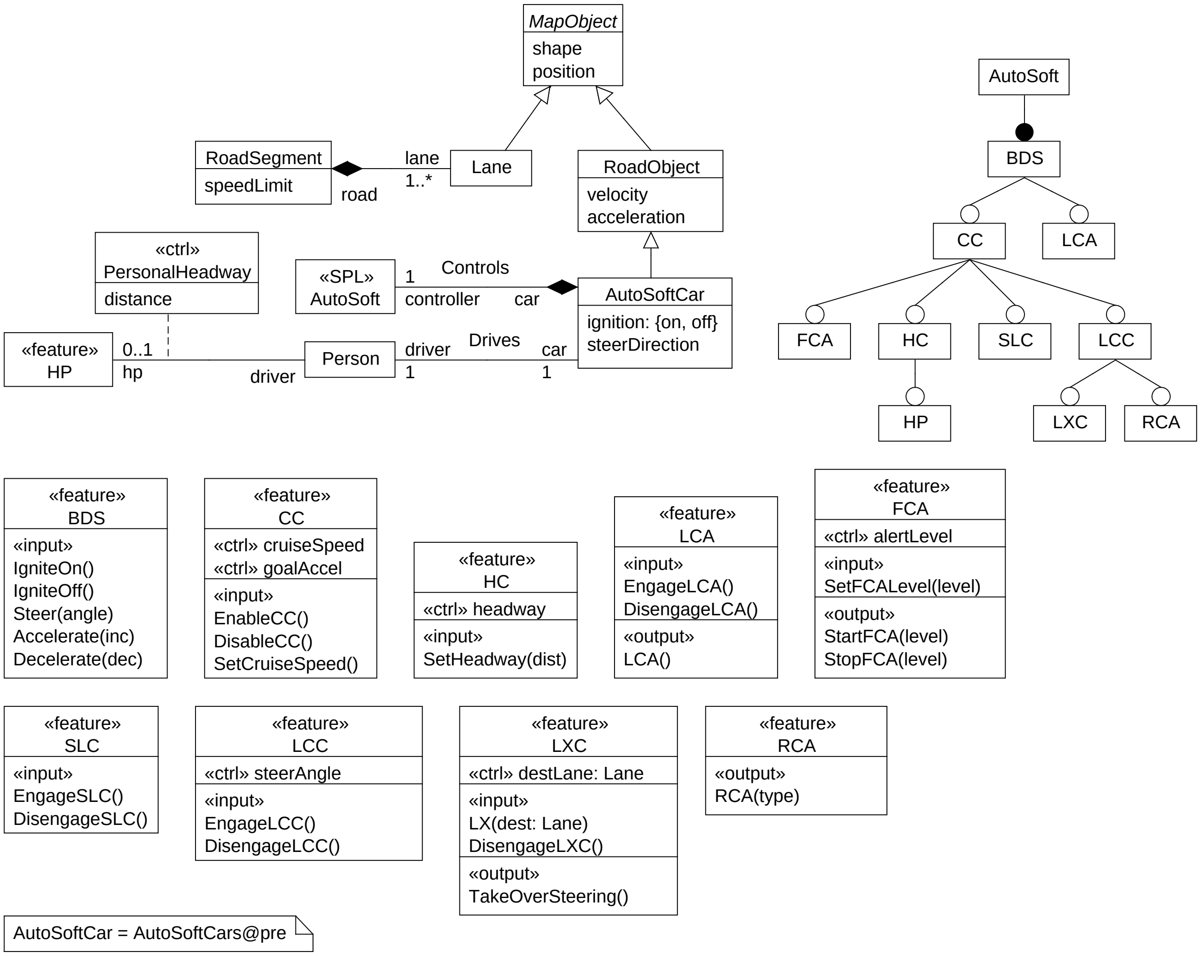


Figure A.50: The *AutoSoft* world model as evolved by RCA

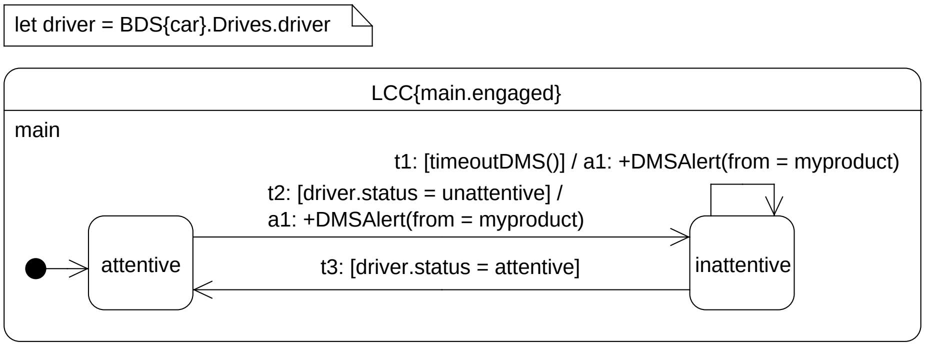

let driver = BDS{car}.Drives.driver
LCC{main.engaged}
main
t1: [timeoutDMS()] / a1: +DMSAlert(from = myproduct)
t2: [driver.status = unattentive] /
a1: +DMSAlert(from = myproduct)
attentive
inattentive
t3: [driver.status = attentive]

Figure A.51: The feature module of DMS

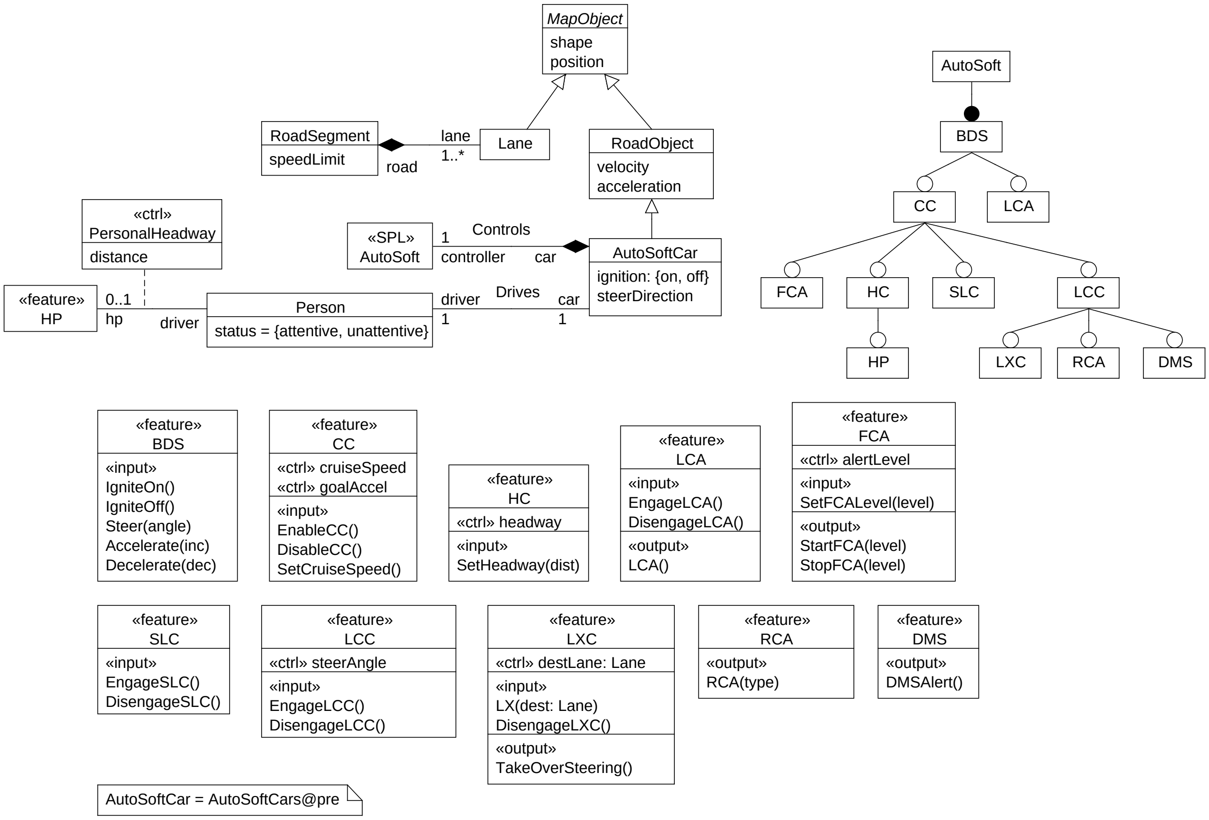


Figure A.52: The *AutoSoft* world model as evolved by DMS

References

[1] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *ICMT*, pages 4–19, 2009.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *JOT*, 8 (5):49–84, 2009.

[3] S. Apel, T. Leich, M. Rosenmï£¡ller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE*, pages 125–140, 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *TSE*, 34 (2):162–180, 2008.

[5] Sven Apel, Christian Kästner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, pages 221–231, 2009.

[6] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An algebra for feature-oriented software development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, 2007.

[7] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *ASE*, pages 372–375, 2011.

[8] Patrizia Asirelli, Maurice H. Ter Beek, Alessandro Fantechi, and Stefania Gnesi. A logical framework to deal with variability. In *IFM*, pages 43–58, 2010.

[9] Patrizia Asirelli, Maurice H. ter Beek, Stefania Gnesi, and Alessandro Fantechi. Formal description of variability in product families. In *SPLC*, pages 130–139, 2011.

[10] B. W. Bates, Jean-Michel Bruel, R. B. France, and M. M. Larrondo-petrie. Formalizing fusion object-oriented analysis models. *FMOODS*, pages 222–233, 1996.

[11] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30 (6):355–371, 2004.

[12] R.K. Berman and J.H. Brewster. Perspectives on the AIN architecture. *IEEE Communications Magazine*, 30 (2):27–32.

[13] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use case description of requirements for product lines. In *REPL*, pages 12–18, 2002.

[14] Armin Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[15] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Lin Y-J. The feature interaction problem in telecommunication systems. In *SETSS*, pages 59–62, 1989.

[16] K. H. Braithwaite and J. M. Atlee. Towards automated detection of feature interactions. In *Feature Interactions in Telecommunications Systems*, pages 36–59. IOS Press, 1994.

[17] G. Bruns. Foundations for features. In *ICFI*, pages 3–11, 2005.

[18] M Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41 (1):115–141, 2003.

[19] M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In *SPIN*, pages 143–162, 2001.

[20] E. J. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, 31 (8):18–23, 1993.

[21] E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin Margaret E. Nilson, Yow jian Lin, Margaret E. Nilson, William K. Schnure, and Hugo Velthuijsen. A feature interaction benchmark for IN and beyond. In *IEEE Communications Magazine*, volume 31 (3), pages 64–69, 1993.

[22] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, 2001.

[23] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE*, pages 335–344, 2010.

[24] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE*, pages 321–330, 2011.

[25] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[26] P. Combes and S. Pickin. Formalisation of a user view of network and services for feature interaction detection. In *Feature Interactions in Telecommunications Systems*, pages 120–135, 1994.

[27] CVL Submission Team. Common variability language (CVL), OMG revised submission. http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf, 2012.

[28] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, pages 422–437, 2005.

[29] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[30] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *REJ*, pages 235–265, 2010.

[31] Fathiyeh Faghih and Nancy A. Day. Mapping big-step modelling languages to SMV. Technical Report CS-2011-29, David R. Cheriton School of Computer Science, University of Waterloo, 2011.

[32] Alessandro Fantechi and Stefania Gnesi. A behavioural model for product families. In *ESEC-FSE*, pages 521–524, 2007.

[33] Alessandro Fantechi and Stefania Gnesi. Formal modeling for product families engineering. In *SPLC*, pages 193–202, 2008.

[34] A. P. Felty and K. S. Namjoshi. Feature specification and automatic conflict detection. In *Feature interactions in telecommunications and software systems VI*, pages 179–192, 2000.

[35] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA workshop in ISSTA*, pages 39–48, 2006.

[36] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *TSE*, 31 (7):529–536, 2005.

[37] A. Gammelgaard and J.E. Kristensen. Interaction detection, a logical approach. In *Feature Interactions in Telecommunications Systems*, pages 178–196, 1994.

[38] Martin Glinz, Stefan Berner, and Stefan Joos. Object-oriented modeling with adora. *Inf. Syst.*, 27(6):425–444, 2002.

[39] Stefania Gnesi and Marinella Petrocchi. Towards an executable algebra for product lines. In *SPLC*, pages 66–73, 2012.

[40] M.L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *ICSR*, pages 76–85, 1998.

[41] Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. Modeling and model checking software product lines. In *FMOODS*, pages 113–131, 2008.

[42] R. J. Hall. Feature combination and interaction detection via foreground/background models. In *FIW*, pages 449–469, 1998.

[43] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *LICS*, pages 54–64, 1987.

[44] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *FSE*, pages 110–119, 2000.

[45] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of ICSE*, pages 943–944, 2008.

[46] C. L. Heitmeyer and R. D. Jeffords. The SCR tabular notation: A formal foundation. Technical Report NLR/MR/5546-03-8678, Naval Research Lab, 2003. NLR/MR/5546-03-8678.

[47] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[48] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[49] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2001.

[50] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *ICSE*, pages 15–24, 1995.

[51] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process, and Organization for Business Success.* Addison-Wesley-Longman, 1997.

[52] Alma L. Juarez Dominguez. Detection of feature interactions in automotive active safety features. Ph.D. Thesis, 2012.

[53] Alma L. Juarez-Dominguez, Nancy A. Day, and Jeffrey J. Joyce. Modelling feature interactions in the automotive domain. In *MiSE*, pages 45–50, 2008.

[54] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.

[55] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *McGPLE workshop of GPCE*, pages 35–40, 2008.

[56] A. Khoumsi. Detection and resolution of interactions between services of telephone networks. In *Feature Interactions in Telecommunication Network IV*, 1997.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[58] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[59] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. *FIW*, pages 293–310, 2000.

[60] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. pages 177–186, 2009.

[61] R. C. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In *ICFI*, pages 129–144, 2007.

[62] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (3rd Edition)*. Prentice Hall, 2005.

[63] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.

[64] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[65] K. Lee, Kang. K. C., S. Kim, and Lee. J. Feature-oriented engineering of PBX software. *APSEC*, pages 394–403, 1999.

[66] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented software designs. In *FIW*, pages 178–197, 2005.

[67] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model-based testing of delta-oriented software product lines. In *TAP*, pages 67–82, 2012.

[68] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. 1969.

[69] Silvio Meier, Tobias Reinhard, Christian Seybold, and Martin Glinz. Aspect-oriented modeling with integrated object models. In *Modellierung*, volume 82 of *LNI*, pages 129–144, 2006.

[70] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction as a context sharing problem. In *ICFI*, pages 133–148, 2009.

[71] OMG. *Unified Modelling Language (UML) Specification, Version 2.0*, 2005.

[72] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006.

[73] OMG. *Object Constraint Language (OCL) 2.0 Specification*, 2006.

[74] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, pages 167–183, 1981.

[75] S. Park, M. Kim, and V. Sugumaran. A scenario, goal and feature-oriented domain analysis approach for developing software product lines. *IMDS*, 104 (4):296–308, 2004.

[76] David L. Parnas. Functional documents for computer systems. *Science of Computer Programming*, 25 (1):41–61, 1995.

[77] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.

[78] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[79] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code generation to support static and dynamic composition of software product lines. In *GPCE*, pages 3–12, 2008.

[80] Pourya Shaker and Joanne M. Atlee. Behaviour interactions among product-line features (submitted to Modularity), 2013.

[81] Pourya Shaker, Joanne M. Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *RE*, pages 151–160, 2012.

[82] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[83] Reinhard Stoiber, Silvio Meier, and Martin Glinz. Visualizing product line domain variability by aspect-oriented modeling. *REV*, 0:8–13, 2007.

[84] Ali Taleghani and Joanne M. Atlee. Semantic variations among uml statemachines. In *MoDELS*, pages 245–259, 2006.

[85] Maurice H. ter Beek, Stefania Gnesi, Carlo Montangero, and Laura Semini. Detecting policy conflicts by model checking uml state machines. In *ICFI*, pages 59–74, 2009.

[86] Maurice H. ter Beek, Franco Mazzanti, and Aldi Sulova. VMC: A tool for product variability analysis. In *FM*, pages 450–454, 2012.

[87] M. Thomas. Modelling and analysing user views of telecommunications services. In *Feature Interactions in Telecommunications Systems*, pages 168–182, 1997.

[88] S. Trujillo, D. Batory, and O. Diaz. Feature oriented model driven development: A case study for portlets. *International Conference on Software Engineering*, pages 44–53, 2007.

[89] Carlton Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *JSS*, 49(1):3–15, 1999.

[90] Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Jianxiong Pang, Tom Gray, Peter Perry, and Joe Ireland. Policy support for call control. *CSI*, pages 635–649, 2006.

[91] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[92] H. Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In *FIW*, pages 31–42, 1995.

[93] P. Zave. Requirements for evolving systems: a telecommunications perspective. In *RE*, pages 2–9, 2001.

[94] P. Zave and M. Jackson. A component-based approach to telecommunication software. *IEEE Software*, 15 (5):70–78, 1998.