A Feature-Oriented Requirements Modelling Language (FORML)

Pourya Shaker and Joanne M. Atlee
{p2shaker, jmatlee}@uwaterloo.ca
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada

Tech. Rep. CS-2012-05

Contents

1 Introduction

Telephony software is often thought of in terms of its constituent features, where each *feature* is "an optional or incremental unit of functionality" [10]. There are a number of advantages to taking a feature-oriented view of software. For one, features serve as a shared vocabulary among customers, engineers, and marketers for discussing software functionality. Second, *feature modularity* eases system development and evolution because features can be developed in isolation, in parallel, and by third-party vendors. Feature orientation is particularly relevant in the context of telephony software, which is developed as product line whose individual products are understood, constructed, managed, and evolved in terms of their features. In fact, there is an increasingly popular software-development paradigm, called *feature-oriented software development (FOSD)* [1], that advocates the use of features as first-class entities throughout a system's lifecycle.

The downside of feature orientation is that engineers must consider how features interact when deriving a product from a selection of features. Two features interact with each other when "one feature affects the operation of [the other] feature" [3]. Some interactions are planned: call waiting is *designed* to interact with and extend basic call service. Other interactions are innocuous. However, some feature interactions are harmful. For example, most telephony features affect the processing of calls (e.g., basic call service, call waiting, call forwarding on busy, three-way calling, call transfer); in a given situation (e.g., the arrival of a new call when a user is already in a call), some subset of these features will react and attempt to end or otherwise process the new call. To avoid conflicts between such reactions, the telephony-software developer needs to understand which situations activate which features, and must determine and document the appropriate behaviour of all possible combinations of features.

The goal of this research is to support the modelling of requirements of the features in a software product line – including the requirements of intended interactions among the features. We focus on the requirements phase of feature development for two reasons: (1) the application of FOSD to requirements artefacts has to date been informal and less well studied, and (2) requirements models are relatively small and abstract, and thus are amenable to automated analyses[1].

A second major objective of this work is to provide explicit support for documenting intended feature interactions. We do this using *enhancements* (i.e., extensions) to the requirements of existing features. There are three advantages to modelling some requirements as enhancements: (1) Sometimes, it is easier to understand a new feature in terms of its incremental changes to existing features. For example, call waiting is best described as an optional enhancement to basic call service, as opposed to being specified as a distinct full-fledged feature. (2) Explicit enhancements to existing features (expressed as model fragments to be superimposed onto the requirements of other features) effectively introduce variation points and variant behaviours to the enhanced features: if the new feature is present in a product, then the new variant behaviour is present in the product; if the new feature is absent from the product, then the behaviour of the product defaults to the behaviour of the existing features. As a result, adding a new feature to the product line is always an *additive* modelling task. Even when the purpose of a

[1]Although this report does not address analysis of feature models, this is planned future work. As such, we are always cognizant of the analyzability of our models.

new feature is to restrict or override existing features, the task of the requirements engineer is to add behaviour – possibly variant behaviour – to the existing requirements. This additive nature of requirements modelling is important because it eases the task of evolving the requirements documentation whenever new, possibly interacting, features are added to the product line. (3) Explicit modelling of intended feature interactions means that, in the future when we investigate analyses for detecting interactions among features, the analyses can avoid "detecting" planned interactions and can focus on reporting only unexpected interactions.

In this paper, we propose a feature-oriented requirements modelling language (FORML) that decomposes requirements models along two dimensions: (1) requirements views, as in traditional requirements-engineering methods, and (2) features, as in FOSD methods:

- There is a **world model** that defines the world phenomena that are relevant to the requirements, a la an ontology. At the same time, the world model specifies the context in which the features operate: it specifies the set of possible *world states*, which the features monitor and control. *Domain knowledge* or *assumptions* restrict the set of valid world states; they are expressed as constraints on the world model.

- The **behaviour model** is an extended finite state-machine model whose input language is the set of possible events and conditions over the world state and whose output language is the set of possible changes to the world state. To delineate features, we decompose the behaviour model into separate feature modules. A *feature module* comprises feature machines, feature-machine fragments, or both: a feature that enacts stand-alone requirements adds a new feature machine to the behaviour model, and a feature that enacts enhancements to another feature adds model fragments to the behaviour model. Model fragments are superimposed onto existing feature machines at specified locations.

The execution semantics of a collection of features is the parallel execution of the feature machines after they have been superimposed with their enhancements.

The rest of this report is organized as follows. Section 2 summarizes the reference model used in requirements engineering to scope the information to be included in a requirements model. Sections 3 and 4 present the models of FORML, including a world model that depicts the context of the features, a feature model (part of the world model) that clusters features into product lines and that specifies allowable configurations, and a behaviour model. In our presentation of FORML, we provide some advice on how to structure a requirements model. We conclude and discuss future work in Section 5. Appendix A demonstrates FORML on a collection of 15 telephony features.

2 Requirements Reference Model

We adopt the Jackson and Zave view of software requirements [6]: a software project aims at constructing a *machine*, comprising a software system installed on a computer, to solve some *problem*. The machine's *requirements* are a description of the problem to be solved.

Figure 1 shows a requirements reference model based on the above definition that is used to scope the information included in a machine's requirements. The context in which the problem occurs is called the *problem world*, and is shown as the light-blue oval on the left. The phenomena that the machine can monitor or control, called *machine phenomena*, is shown as the pink

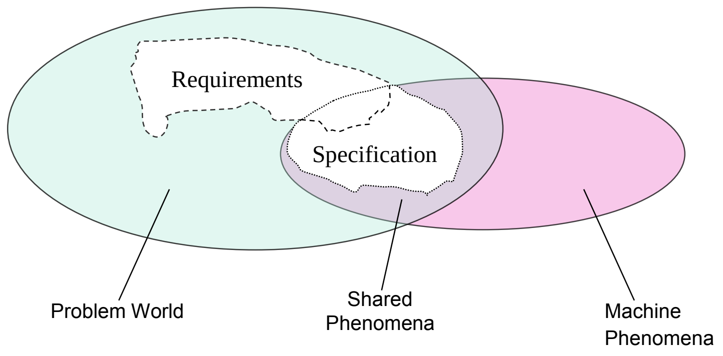


Figure 1: Requirements Reference Model

oval on the right. The intersection between the problem world and machine phenomena, called *shared phenomena*, are problem-world phenomena that the machine can directly monitor or control. Shared phenomena are at the machine's interface; that is, they are the inputs and outputs of the machine's sensors and actuators, respectively. In the requirements reference model, requirements are expressed solely in terms of problem-world phenomena (possibly including shared phenomena). The machine's *specification* is a description of the externally-observable behaviours that the machine should exhibit, so as to satisfy the requirements, and is expressed solely in terms of shared phenomena.

In the above reference model, the machine is normally thought of as containing a single software system. We propose extending the use of the reference model to software products lines (SPLs), in which the machine is thought of as comprising one or more SPLs. In general, the shared phenomena includes possible values input to the machine's configuration parameters by agents in the problem world. Therefore, in the proposed view, the shared phenomena includes possible values of the configuration parameters used to instantiate products of the SPL. One such parameter is the feature-configuration parameter of the SPL, a value of which is a set of features (aka a *feature configuration*) that characterizes a single product of the SPL.

3 FORML

The purpose of this report is to describe FORML as it is to be used to model the requirements of a system of related SPLs. Following the requirements reference model described in Section 2, a FORML model comprises a description of the problem world, called a *world model*, and a specification of the requirements in terms of the world model, called a *behaviour model*. The behaviour model, described in Section 3.3, is structured in terms of the types of features supported by the SPLs. Recall that in our extended view of the reference model, the problem world includes the possible values of the feature-configuration parameter of each SPL; this information is specified in a part of the world model, called a *feature model*, described in Section 3.1. Other phenomena in the problem world are described in terms of a concept model augmented with constraints, as described in Section 3.2.

3.1 Feature Model

A complex system can be decomposed into a set of sub-systems, each of which is a member of a different SPL. For example, a telephony system can be decomposed into a call-processing system and an administrative system, where each sub-system is instantiated from an SPL. A FORML feature model consists of a set of *feature diagrams*, one for each SPL. A feature diagram is the traditional method for specifying the space of products of an SPL, where each product is characterized in terms of its feature configuration.

A FORML feature diagram is expressed using the original feature-model notation [7]. The feature diagram for an SPL is a tree, where the tree root is the SPL, and every other tree node is a *feature type*. Figure 2b shows the feature model for the *TelSoft* SPL, which will be used as a running example in this document. A *TelSoft* product is a telephone service subscribed to by a user. The features types supported by the *TelSoft* SPL are the following:

- *Basic call service (BCS)*, which responds to the subscriber's requests for starting, accepting, and ending calls.
- *Call waiting (CW)*, which allows the subscriber to accept a second call and switch between the two calls.
- *Caller number delivery (CD)*, which deliver's a caller's number to the subscriber.
- *Caller number delivery blocking (CDB)*, which prevents CD from delivering the subscriber's number to a callee.

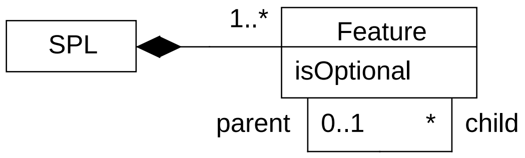
The feature model in Figure 2b specifies that a *TelSoft* product must have BCS, and can optionally have CW, CD, and CDB. Hence, the space of *TelSoft* products is the following: $\{BCS\}$, $\{BCS, CW\}$, $\{BCS, CD\}$, $\{BCS, CDB\}$, $\{BCS, CW, CD\}$, $\{BCS, CW, CDB\}$, $\{BCS, CD, CDB\}$, and $\{BCS, CW, CD, CDB\}$.

The abstract syntax for a feature diagram is given by the metamodel presented in Figure 2a. The feature types define the space of feature configurations within a product line. A feature type is either optional or mandatory. The optionality of feature types and the tree structure specify the *valid* feature configurations; that is, the allowable feature sets that characterize the allowable SPL products. In a valid feature configuration, a mandatory feature must be present, and an optional feature can be present, only if its parent feature is present (if it has a parent feature).

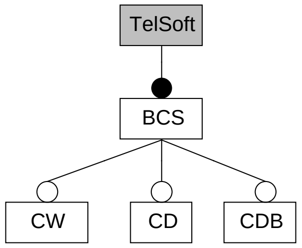
In the graphical representation of a feature model, SPL nodes have a gray background. A feature type topped with a filled circle denotes a mandatory feature type (e.g., BCS), and one topped with an empty circle denotes an optional feature type (e.g., CW).

3.2 World Model

A world model defines the ontology of the shared problem world of a set of SPLs. In FORML, the world model is expressed as a concept model augmented by constraints. Section 3.2.1 gives a brief impression of what a world model looks like, using the *TelSoft* problem world as an example. The complete world-model syntax is described in Section 3.2.2 and Section 3.2.4, with references to the *TelSoft* world model presented in Section 3.2.1.



(a) UML metamodel for feature diagrams



(b) *TelSoftFM* feature model

Figure 2: Feature Model

3.2.1 *TelSoft* Problem World

Figure 3 shows the world model for *TelSoft*, expressed as a UML class diagram. The problem world of *TelSoft* comprises the following phenomena:

- Users that subscribe to *TelSoft* products, modelled as a *concept User*. A user's *TelSoft* product has a feature configuration, modelled in Figure 2b.
- Calls between users, modelled as the *association Call*. Once a call is accepted, a voice connection is established, as modelled by the *attribute voice*.
- Commands issued to and notifications sent by *TelSoft* products are modelled as distinct *messages* associated with feature types. For example, there are input commands to start and accept calls (processed by BCS), and notifications regarding a caller's number (sent by CD).
- Features BCS and CW keep track of the call they are processing, as modelled by the associations *Processing* and *Second*, respectively.

3.2.2 Concept Model

Metamodel The abstract syntax for a conceptual model is given by the metamodel presented in Figure 4. A concept model describes a problem world as a set of *concepts* and the relationships between them, where each concept represents a type of object in the problem world (e.g., *User*). The instances (*objects*) of a concept are characterized by a set of properties, called *attributes*, each of which has a name, a type, and a multiplicity (e.g., attribute *voice* of *Call*, with type *bool*, and an implicit multiplicity of one). The type of an attribute is either an enumeration (i.e., a finite set of named values), an undefined type, or a type defined outside of the FORML model (e.g., type *bool*). Note than the type of an attribute is never a concept; such complex properties are modelled as associations, as described below. An attribute declaration may include a multiplicity, in which case the attribute's value is a set of values of the attribute's type. One concept may be a special case of another concept, in which case the first concept is a subtype of the second.

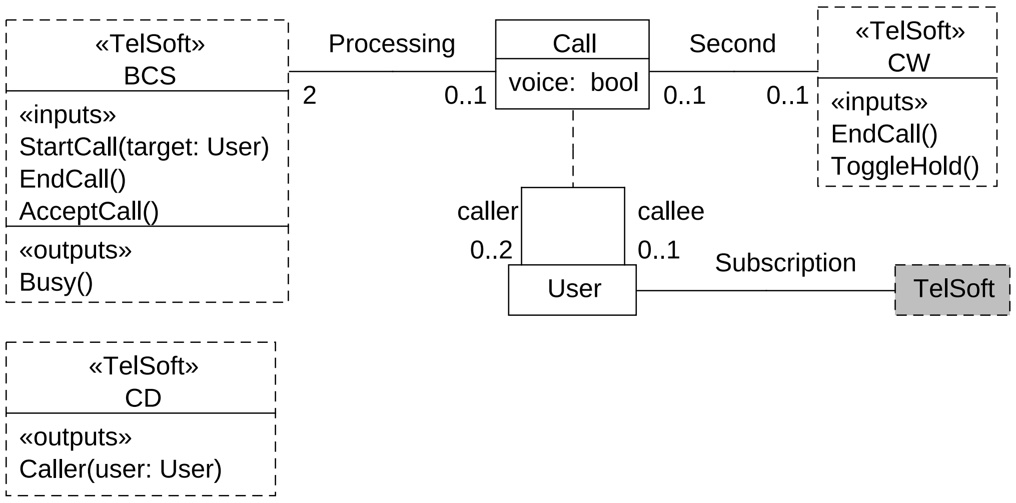


Figure 3: *TelSoft* world model

FORML distinguishes between the following special types of concepts, adapted from KAOS [9]:

- An *association* is a type of relationship that exists among two or more other objects, each of which takes some *role* in the relationship. An association cannot relate objects of its own type. An association instance is called a *link*. Each role in an association has a name, a type, and a multiplicity. The type of a role is a concept whose objects can take the role in an instance of the association. The multiplicity of a role constrains the number of objects that can take that role in links with a particular set of objects in the other roles. For example, the multiplicity *0..1* of role *callee* of association *Call* literally means that zero or one *User* objects can be in the *callee* role of *Call* links with a particular *User* object. If any of a link's related objects ceases to exist, so does the link.
  - An *aggregation* or *composition* is a special type of binary association. A link of an aggregation or association represents a weak or strong whole-part relationship between two objects, respectively. In an aggregation, a part belongs to zero or more whole objects, and can belong to different whole objects throughout its lifetime; whereas in a composition, a part belongs to exactly one whole object, and belongs to the same whole object throughout its lifetime. The roles of this latter composition have implicit multiplicities of one.
- A *message* is a type of communication to or from an SPL product. An instance of a message is primarily characterized by its parameters, which are modelled as either attributes (when the parameter types are simple) or as aggregations (when the parameter types are concepts)[2]. We use aggregation rather than association to model complex parameters because each parameter is a *part* of the message, and thus there is a whole-part relationship between the two. In fact, such aggregations are the only associations

[2]Note that *Message* is a type of *concept*, and thus can have attributes and associations.

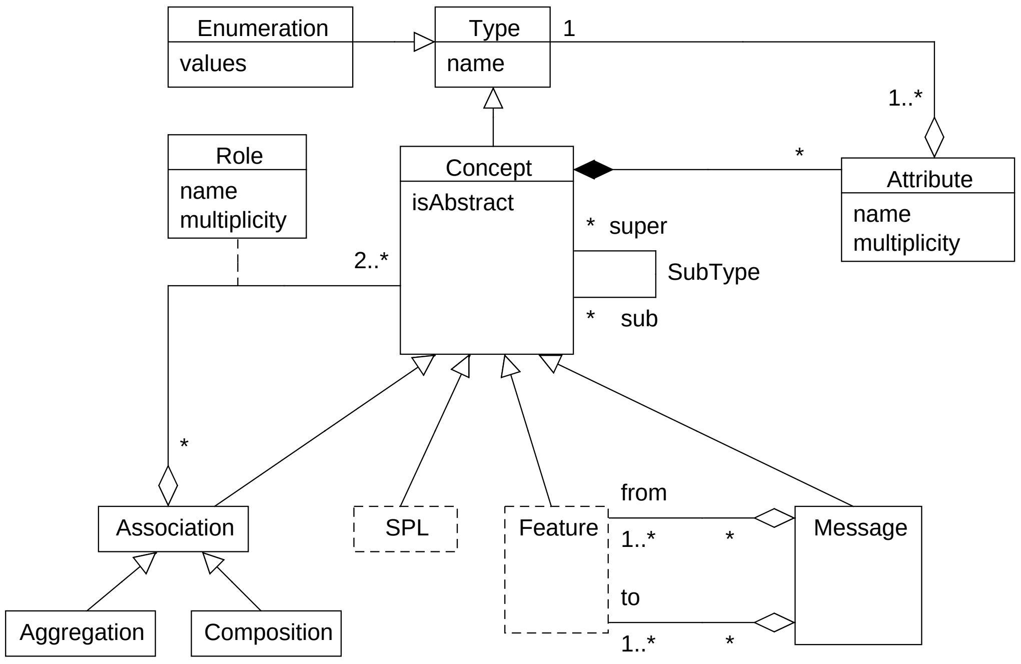


Figure 4: UML metamodel for conceptual models (The dashed border of the *SPL* and *Feature* classes indicates that these classes are defined in Figure 2a.)

that a message can participate in. Input message objects are further characterized by their destinations, and output messages are further characterized by their sources; the sources or destinations are the features that process or generate the messages, respectively. Because source and destinations are an integral part of the message, they are also modelled as aggregations. For example, a *StartCall* message object has a parameter *target* of type *User* and may be processed by a *BCS* object.

- An *SPL* or *feature type* defined in a feature model is a concept in the corresponding world model (e.g., *TelSoft* and its feature types, defined in Figure 2b, are *SPL* and *Feature* concepts in Figure 3).

We include SPLs and their feature types in the world model for the following reasons: First, a system often has a set of configuration parameters, the values of which affect the externally observable behaviours of instances of the system. We consider the possible values of such parameters to be among the shared phenomena of the problem world. An SPL is configured into a product, in part by selecting the feature configuration of the product. This choice affects the product's externally observable behaviours. Consequently, we consider the feature-configuration space of an SPL (i.e., the set of possible feature selections) to be part of the SPL's problem world. Second, there exist relationships between SPL products – and in some cases particular features of SPL products – and phenomena in the SPL's problem world. Such relationships can be described in a world model as attributes, associations, and messages that involve SPL and feature concepts. For example, the following relationships exist between *TelSoft* products and phenomena in the *TelSoft* problem world:

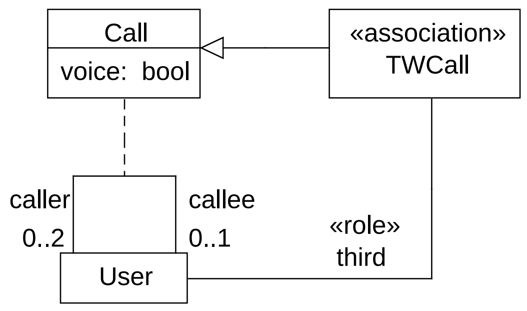


Figure 5: Graphical representation of higher-arity associations

- *TelSoft* products are subscribed to by users, modelled as a composition involving the *TelSoft* concept.
- Commands to *TelSoft* products are processed by particular types of *TelSoft* features, modelled by relating the corresponding messages and feature concepts, as described above.
- The processing of calls by BCS and CW is modelled as associations between *Call* and the *BCS* and *CW* concepts, respectively.

Graphical Syntax A concept model is graphically expressed using the UML class-diagram notation with the following conventions: a concept is normally shown as a UML class (e.g., *User*). An SPL or feature concept is distinguished with a dotted border (e.g., *BCS*). Only SPL and feature concepts that have attributes, take part in associations, or send or receive messages are shown. An SPL concept is further distinguished with a gray background (e.g., *TelSoft*). Attribute multiplicities have the same format as UML association multiplicities, and are specified in square brackets following the attribute name. For example, *a [1..*]: T* specifies an attribute *a* with multiplicity *1..**. The feature concepts shown have a stereotype indicating the SPL to which they belong (e.g., the stereotype «*TelSoft*» of *BCS*).

A binary association is shown as a UML association, aggregation, or composition (e.g., *Subscription*); if a binary association has attributes, or is a role type of another association, it is shown as a UML association class (e.g., *Call*). An association with a higher arity is shown as a UML class with the stereotype «*association*»; a role of such an association is shown as a UML association with the stereotype «*role*», which is between the UML classes corresponding to the association and role type, and is labelled with the name and multiplicity of the role (e.g., *TWCall* in Figure 5).

A message is shown as a *message signature* in a compartment of each of the UML classes representing a feature that sends or receives such messages. The UML class of a feature type includes a compartment listing the messages to which the feature responds (prefaced with the stereotype «*input*») and a compartment listing the messages that the feature generates (prefaced with the stereotype «*output*»). A message signature is of the form $M(p_1\ [m_1]{:}T_1,\ldots,p_n\ [m_n]{:}T_n)$, where M is the name of the message concept, and p_i, T_i, and m_i ($1 \leq i \leq n$) are names, types, and optional multiplicities of M's parameters, respectively (e.g., *StartCall(target: User)*). A non-*TelSoft* example of a parameter that has a multiplicity is the parameter *names*

[1..]: ID* of a registration request to a service, which literally means a set of one or more IDs to be registered with the service. In the case of a parameter that is the part role of an aggregation, the parameter's multiplicity corresponds to that of the part role; the multiplicity of the corresponding whole role (i.e., the message) is implicitly ***.

An enumeration E is defined in a UML note as *enum E = {v_1,...,v_m}*, where v_i $(1 \leq i \leq m)$ are enumeration values.

Multiplicities, role names, and aggregation and composition names can be left unspecified, in which case they take on the following default values: the default multiplicity value is *1* (e.g., the multiplicities of both roles of *Subscription*); the default name of a role is the role type (e.g., the name of the roles of *Subscription* are *User* and *TelSoft*); and the default name of an aggregation or composition is *W_P*, where W and P are the whole and part types, respectively. Macros can be used to simplify text in a world model (e.g., a list of message concepts). A macro m that stands for the text *txt* is defined in a UML note as *let m = txt*.

3.2.3 World State

A world model defines a space of *world states*, each representing a possible state of the products' problem world. A world state consists of the following:

- A set of objects (instances) of the concepts defined in the conceptual model
- For each object in the world state, its attribute values; in the case of an association, the objects that take each of its roles; in the case of a message, its parameters and the SPL product that sends or receives it; and in the case of an SPL product, its feature configuration

Figure 6 shows a possible world state of the *TelSoft* world model, expressed in the UML object-diagram notation[3]. The world state includes a single call with a voice connection between two users, each with a *TelSoft* product. The caller's *TelSoft* product has features BCS and CD, and that of the callee has BCS and CW. The BCS features of both products are processing the call. A command to end the call is sent to the callee's *TelSoft* product.

The world state changes over time. For example, the world state shown in Figure 6 will change in at least the following ways: The call and its associations with the BCS features will be removed in response to the end-call command. Moreover, message objects are transient: each object is present only in a single world state and is gone in the subsequent state. Thus, in Figure 6, the end-call command will not exist in the next world state.

When we say that an SPL's requirements monitor and control problem-world phenomena, we mean that they refer to and change the current world state. Thus all events and expressions in a FORML behaviour model refer to objects in the current world state of an executing product from the SPL; and all actions in the behaviour model are to be applied to objects in the current world state.

[3]FORML does not prescribe a notation for specifying individual world states. The UML object-diagram notation is used here for presentation purposes.

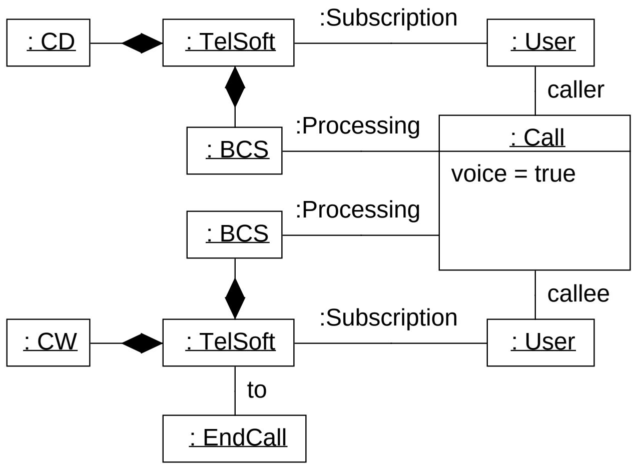


Figure 6: A *TelSoft* world state

3.2.4 Constraints

As mentioned above, one of the purposes of the world model is to specify the set of possible world states in which a product operates. As such, it is often necessary to augment the conceptual-world model with a set of constraints on allowable instantiations. There are several reasons why the world model may need to be constrained:

Domain knowledge represent facts, such as natural laws, about the world that are known to be true.

Assumptions are suppositions about the world that might be violated, but that have to be true in practice for our future product to work as advertised. To give an example from our *TelSoft* world: we may assume that commands to start and end calls cannot be simultaneously issued to a *TelSoft* product:

$$\textbf{all}\ sc : StartCalls,\ ec : EndCalls \mid sc.to \neq ec.to$$

Requirements are restrictions on the world that are to be imposed by our future products.

There are two types of constraints of interest: world-state constraints and world-state–transition constraints. *World-state constraints* restrict the world-state space defined by the conceptual world model. The three examples given above are world-state constraints. *World-state transition constraints* specify restrictions on consecutive world states; that is, they restrict how the world state can change on its own or be changed by our products. For example, we may specify an assumption that the set of *User*s does not change:

$$Users\textbf{@pre} = Users$$

where suffix "**@pre**" is used to refer to values in the previous world state. Graphically, constraints are specified in UML notes within the world model.

The following subsection describes the general language for writing expressions over world states. Constraints are predicate expressions: world-state constraints are predicate expressions over a single world state, and world-state–transitions are predicate expressions over a pair of consecutive world states. As in seen in Section 3.3, the expression language described below is used not only for writing constraints on the world model but also for writing expressions in the behaviour model.

3.2.4.1 Expression Language over World State

A FORML expression is an expression over the elements in one (or two consecutive) world state(s). In the following, an expression **e** in bold typeset refers to the expression itself, and $val(\mathbf{e})(WS)$ refers to the meaning of **e** in the world state *WS*.

3.2.4.1.1 Atomic Expressions

The FORML expression language borrows from Alloy the intuition that elements of the world state are *sets*. To simplify the expression language, an individual object (or value) is treated as a singleton set of objects (or values). In this manner, the engineer does not have to keep track of whether he is writing an expression over an individual object (or value) or over a set of objects (or values).

In FORML, an atomic expression refers to a single set of objects (or values) in a single world state. There are five types of atomic expressions: the empty set **none**, type **T**, variable **v**, constant **c**, and set **Cs** where *C* is a Concept. The meanings of the atomic expressions with respect to a world state *WS* are given below:

$val(\mathbf{none})(WS)$	= the empty set
$val(\mathbf{T})(WS)$	= the set of possible values in type **T**
$val(\mathbf{Cs})(WS)$	= the set of objects of concept **C** in *WS*
$val(\mathbf{v})(WS)$	= the set of objects or values of variable **v** in *WS*
$val(\mathbf{c})(WS)$	= the "set" of constant value **c**

For example, **Users** is the set of objects in the world state of concept *User*; and **true** is the "set" of constants *true*.

3.2.4.1.2 Set Expressions

Starting from an object or set of objects in a world state, it is possible to derive sets of related objects, links or attribute values by *navigating* the relationships among elements in the world state. For example, from a particular *User* **v** in a world state, one can derive the set of *TelSoft* products that **v** subscribes to:

v.Subscription.TelSoft

Literally, the above is a navigation expression that (1) starts with a single *User* **v**, (2) collects in a set all of the *Subscription* links that involve **v**, the (3) collects in a set all of the *TelSoft* objects the play role *TelSoft* in these links.

More generally, a navigation expression starts with a (possibly singleton) set of objects or links and traverses a sequence of relationships, ending with a set of objects or links or values that are related (via the sequence of relationships) to the original entity. Below are definitions of the possible navigation operations, each representing the traversal of a single relationship. In the expressions below, **O** is a (possibly singleton) set of objects, **a** is an attribute, **A** is an association, **r** is a role of an association, **L** is a (possibly singleton) set of links, **C** is a concept, **S** is a (possibly singleton) set of products from a software product line, **F** is the name of a feature type, and **M** is a (possibly singleton) set of message objects:

$val(\mathbf{O.a})(WS)$	= the set of attribute values: $\bigcup\{val(o.a)(WS) \mid o \in val(\mathbf{O})(WS)\}$
$val(\mathbf{O.A})(WS)$	= the set of links that involve objects $o \in val(\mathbf{O})(WS)$
$val(\mathbf{O.A\text{-}r})(WS)$	= the set of links in which objects $o \in val(\mathbf{O})(WS)$ play the role **r**
$val(\mathbf{L.r})(WS)$	= the set of objects of type C that play role **r** in links $l \in val(\mathbf{L})(WS)$
$val(\mathbf{S.F})(WS)$	= the set of feature instances of type **F** in products $s \in val(\mathbf{S})(WS)$
$val(\mathbf{M.to})(WS)$	= the set of products that receive some message $m \in val(\mathbf{M})(WS)$
$val(\mathbf{M.from})(WS)$	= the set of products that sent some message $m \in val(\mathbf{M})(WS)$

There are standard operations on sets, including set union, set intersection, set difference, a selection operation that returns a subset of elements that satisfy some predicate **P**, and a conditional operation whose value depends on some predicate **P**. Lastly, set cardinality returns the number of elements of a set. In the following, **S**, $\mathbf{S_1}$, and $\mathbf{S_2}$ are sets and **P** is a predicate:

$val(\mathbf{S_1}+\mathbf{S_2})(WS)$	$= val(\mathbf{S_1})(WS) \cup val(\mathbf{S_2})(WS)$
$val(\mathbf{S_1}\&\mathbf{S_2})(WS)$	$= val(\mathbf{S_1})(WS) \cap val(\mathbf{S_2})(WS)$
$val(\mathbf{S_1}-\mathbf{S_2})(WS)$	$= val(\mathbf{S_1})(WS) \setminus val(\mathbf{S_2})(WS)$
$val(\mathbf{S[v \mid P]})(WS)$	$= \{\ \mathrm{v} \in val(\mathbf{S})(WS) \mid val(\mathbf{P(v)})(WS)=true\ \}$
$val(\textbf{if P then } \mathbf{S_1} \textbf{ else } \mathbf{S_2})(WS)$	$= val(\mathbf{S_1})(WS)$ if $val(\mathbf{P})(WS)$; otherwise $val(\mathbf{S_2})(WS)$
$val(\#\mathbf{S})(WS)$	= the number of elements of set **S**

For example, the following expression uses the selection operation to refer to the subset of calls in the world state that have a voice connection: **Calls [v | v.voice=true]**.

3.2.4.1.3 Functions

FORML allows simple arithmetic operations (e.g., $+$, $-$, $\times$) over (singleton sets of) integers and real numbers. In addition, *Type* definitions in the world model may introduce operations on specific types (e.g., comparison or arithmetic operators over *Time* values). More complicated operations are named and used in models, but their definition is left unspecified. For example, the *Billing* feature module in Appendix A refers to a function *charge()* without specifying how the charge for a call is computed.

3.2.4.1.4 Predicates

Predicates are expressions that evaluate to *True* or *False*. As mentioned above, predicates are widely used as guard conditions on actions and as constraints on the world model or on behaviour.

Predicates include comparison operations over expressions $\mathbf{E_1}$ and $\mathbf{E_2}$ of the same type:

$$
\begin{array}{lcl}
\mathit{val}(\mathbf{E_1}{=}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){=}\mathit{val}(\mathbf{E_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{E_1}{\neq}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){\neq}\mathit{val}(\mathbf{E_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{E_1}{<}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){<}\mathit{val}(\mathbf{E_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{E_1}{\leq}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){\leq}\mathit{val}(\mathbf{E_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{E_1}{>}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){>}\mathit{val}(\mathbf{E_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{E_1}{\geq}\mathbf{E_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{E_1})(\mathit{WS}){\geq}\mathit{val}(\mathbf{E_2})(\mathit{WS})
\end{array}
$$

There are special predicates on sets, such as whether one set is a subset of another, and predicates about set cardinality. In the following, **S, $\mathbf{S_1}$,** and $\mathbf{S_2}$ are sets:

$$
\begin{array}{lcl}
\mathit{val}(\mathbf{S_1}\ \textbf{in}\ \mathbf{S_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{S_1})(\mathit{WS}) \subset \mathit{val}(\mathbf{S_2})(\mathit{WS}) \\
\mathit{val}(\textbf{no S})(\mathit{WS}) & = & \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ is empty} \\
\mathit{val}(\textbf{lone S})(\mathit{WS}) & = & \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ has zero or one element} \\
\mathit{val}(\textbf{one S})(\mathit{WS}) & = & \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ has exactly one element} \\
\mathit{val}(\textbf{some S})(\mathit{WS}) & = & \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ has one or more elements}
\end{array}
$$

There are standard boolean operations for writing compound expressions over predicates, such as negation, conjunction, disjunction, implication and equivalence. In the following, **P,** $\mathbf{P_1}$, and $\mathbf{P_2}$ are predicates:

$$
\begin{array}{lcl}
\mathit{val}(\textbf{not P})(\mathit{WS}) & = & \neg\ \mathit{val}(\mathbf{P})(\mathit{WS}) \\
\mathit{val}(\mathbf{P_1}\ \textbf{and}\ \mathbf{P_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{P_1})(\mathit{WS}) \wedge \mathit{val}(\mathbf{P_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{P_1}\ \textbf{or}\ \mathbf{P_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{P_1})(\mathit{WS}) \vee \mathit{val}(\mathbf{P_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{P_1}\ \textbf{implies}\ \mathbf{P_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{P_1})(\mathit{WS}) \Rightarrow \mathit{val}(\mathbf{P_2})(\mathit{WS}) \\
\mathit{val}(\mathbf{P_1}\ \textbf{iff}\ \mathbf{P_2})(\mathit{WS}) & = & \mathit{val}(\mathbf{P_1})(\mathit{WS}) \Leftrightarrow \mathit{val}(\mathbf{P_2})(\mathit{WS})
\end{array}
$$

Lastly, there are *quantified predicates* that express predicates about the members of a set. A quantified predicate introduces a variable **v** that refers to an arbitrary member of a specified set **S**, and specifies a predicate **P(v)** over variable **v**. Depending on the type of the quantifier, the quantified predicate is *True* or *False* depending on the number of members of **S** that satisfy **P**:

$$
\begin{array}{ll}
\mathit{val}(\textbf{no v: S} \mid \mathbf{P})(\mathit{WS}) & = \text{there is } \textit{no}\ \mathbf{v} \in \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ such that } \mathit{val}(\mathbf{P(v)})(\mathit{WS}) \text{ is true} \\
\mathit{val}(\textbf{lone v: S} \mid \mathbf{P})(\mathit{WS}) & = \text{there is } \textit{at most one}\ \mathbf{v} \in \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ such that } \mathit{val}(\mathbf{P(v)})(\mathit{WS}) \text{ is true} \\
\mathit{val}(\textbf{one v: S} \mid \mathbf{P})(\mathit{WS}) & = \text{there is } \textit{exactly one}\ \mathbf{v} \in \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ such that } \mathit{val}(\mathbf{P(v)})(\mathit{WS}) \text{ is true} \\
\mathit{val}(\textbf{some v: S} \mid \mathbf{P})(\mathit{WS}) & = \text{there is } \textit{one or more}\ \mathbf{v} \in \mathit{val}(\mathbf{S})(\mathit{WS}) \text{ such that } \mathit{val}(\mathbf{P(v)})(\mathit{WS}) \text{ is true} \\
\mathit{val}(\textbf{all v: S} \mid \mathbf{P})(\mathit{WS}) & = \textit{for all}\ \mathbf{v} \in \mathit{val}(\mathbf{S})(\mathit{WS}),\ \mathit{val}(\mathbf{P(v)})(\mathit{WS}) \text{ is true}
\end{array}
$$

Below are three example quantified predicates that assert that (1) every user has subscribes to exactly one *TelSoft* product, and (2) no two users subscribe to the same *TelSoft* product:

$$\textbf{all v:Users} \mid \textbf{one v.Subscription.TelSoft}$$

$$\textbf{all } \mathbf{u_1,u_2}\textbf{:Users} \mid (\mathbf{u_1} \neq \mathbf{u_2})\ \textbf{implies}\ (\mathbf{u_1}\textbf{.Subscription.TelSoft} \neq \mathbf{u_2}\textbf{.Subscription.TelSoft})$$

The last example exhibits a common pattern, in which a quantifier introduces two variables $\mathbf{v_1}$ and $\mathbf{v_2}$ that represent *distinct* arbitrary members of a set. In such cases, we use keyword **disj** to indicate that the variables are disjoint:

$$\textbf{all disj } \mathbf{u_1,u_2}\textbf{:Users} \mid \mathbf{u_1}\textbf{.Subscription.TelSoft} \neq \mathbf{u_2}\textbf{.Subscription.TelSoft}$$

3.2.4.1.5 @pre

A *world-state transition constraint* is a predicate over a pair of consecutive world states. The language for such predicates is that described above, plus syntax for distinguishing between values in the *before* versus *after* world states of the world-state transition. Specifically, suffix **@pre** is appended to subexpressions of a *before* world state. The **@pre** tag is useful when expressing the current value of a variable relative to its previous value. For example, the following expression states that a *StartCall* message cannot persist over two consecutive world states:

no StartCalls & StartCalls@pre

3.3 Behaviour Model

A FORML behaviour model operationally describes the requirements of a set of SPLs. The requirements of each SPL are specified as a set of *feature modules*, each of which specifies the requirements associated with a single feature type of the SPL.

A feature module can specify the requirements of a feature type as *standalone*, as *enhancements*, or a combination of both. A standalone specification does not refer to the requirements of other feature types whereas an enhancement is specified in the context of the requirements of another feature type. Some requirements can be specified either as stand-alone or as enhancements, but they are easier understood and can be more succinctly expressed if specified as enhancements (examples are given in Section 3.3.2). Intended interactions are associated with two feature types, and therefore, can only be understood and expressed as enhancements. Section 3.3.1 and 3.3.2 describe how a feature module specifies requirements as stand-alone and as enhancements, respectively. Like the description of the world model in Section 3.2, these sections begin with an *TelSoft* example, followed by a complete description of the FORML syntax that makes reference to the example.

The requirements of an SPL are described by the composition of its feature modules. The composition of feature modules is described in Section 4.

3.3.1 Specifying Requirements as a Standalone Model

3.3.1.1 BCS Requirements

Figure 7 shows the feature module of the *BCS* feature type. The UML note at the top declares the feature module by naming the feature type being modelled (*feature BCS*) and the SPL that it belongs to (*SPL TelSoft*). The requirements of *BCS* are shown as a single *feature-machine* declared in a UML note (*feature-machine main*). The input and output language of *main* is based on the *TelSoft* world model shown in Figure 3. The requirements of *BCS* are as follows. A call is initiated upon a *StartCall* request (transitions *t1* and *t4*); if the callee is not available, the call is removed (transition *t7*). A voice connection is established when the callee accepts the call (transitions *t5* and *t6*). At any point in a call, the call is removed upon an *EndCall* request (transitions *t2* and *t3*).

3.3.1.2 Syntax

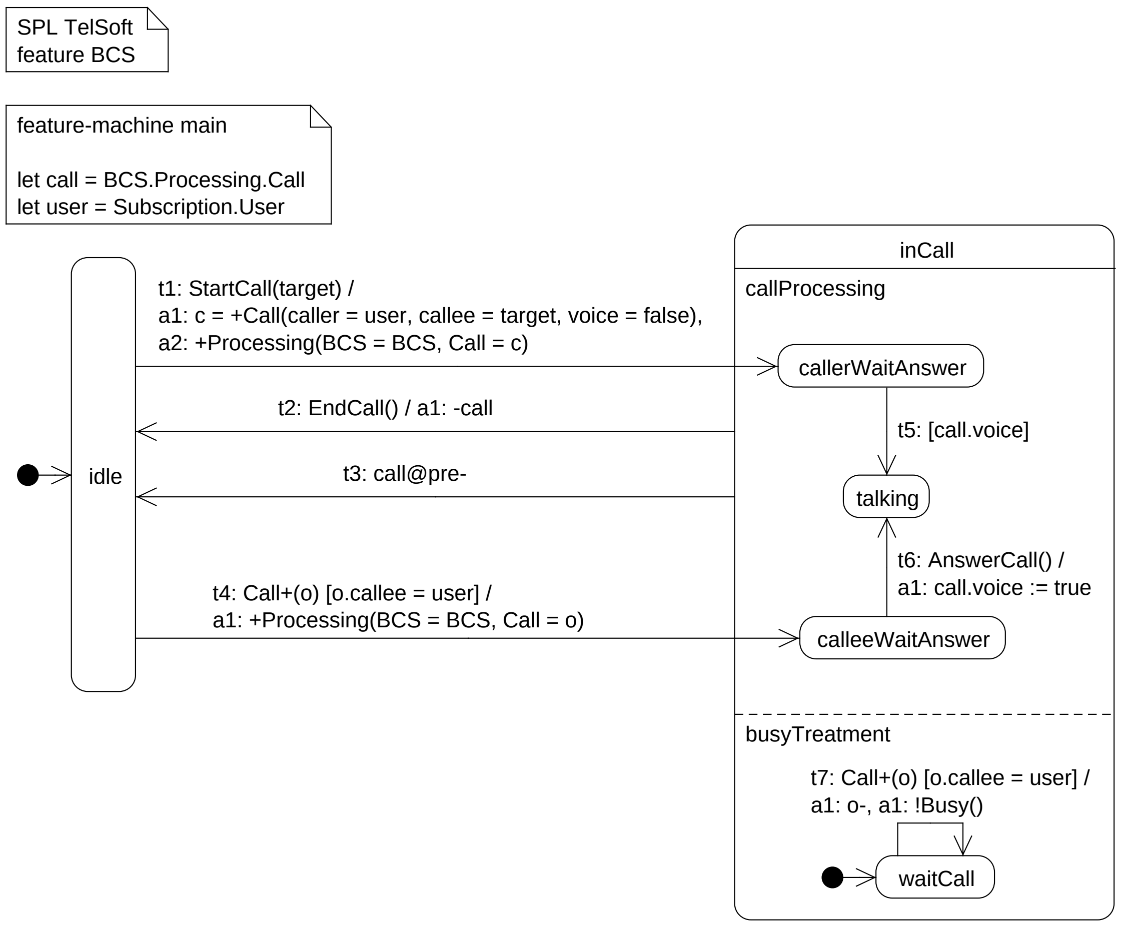

SPL TelSoft
feature BCS
feature-machine main
let call = BCS.Processing.Call
let user = Subscription.User
inCall
callProcessing
t1: StartCall(target) /
a1: c = +Call(caller = user, callee = target, voice = false),
a2: +Processing(BCS = BCS, Call = c)
callerWaitAnswer
t2: EndCall() / a1: -call
t5: [call.voice]
idle
t3: call@pre-
talking
t6: AnswerCall() /
a1: call.voice := true
t4: Call+(o) [o.callee = user] /
a1: +Processing(BCS = BCS, Call = o)
calleeWaitAnswer
busyTreatment
t7: Call+(o) [o.callee = user] /
a1: o-, a1: !Busy()
waitCall

Figure 7: BCS feature module

A set of feature-type requirements are modelled standalone, as one or more concurrent state machines, called *feature machines* (e.g., *BCS* requirements are modelled as a single feature-machine *main*). A feature machine is shown as a UML state machine, prefaced with a UML note that contains the feature-machine's declaration (e.g., *feature-machine main*) and the definition of any macros used to simplify text in the feature machine. There is an implicit variable named *me* that refers to the feature's corresponding SPL product. A feature machine consists of

- A set of states, one of which is designated as the initial state by being the destination of an arrow from a small filled circle (e.g., the initial state of feature-machine *main* is *idle*). A state can be basic or composite. A composite state consists of one or more concurrent regions (e.g., state *inCall* comprises region *callProcessing*), where each region consists of a set of states. A region may also have an initial state.

- A set of transitions between states. A transition cannot connect states that are in different concurrent regions. A transition has a *label* of the form

  $$\mathbf{id:\ e\ [c]\ /\ id_1:\ [c_1]\ a_1,\ \cdots,\ id_n:\ [c_n]\ a_n}$$

  where *id* is the transition name (e.g., *t1*), *e* is the optional triggering event (e.g., triggering event *StartCall(target)* of transition *t1*), c is the optional guard condition, and a_i ($1 \leq i \leq n$) are concurrent actions named id_i with optional guard conditions c_i, respectively. A transition t executes when the state machine is in t's source state, t's triggering event is occuring, and t's guard condition is true. The execution of t takes the state machine to t's destination state, and executes those actions of t whose guard conditions are true. The languages for triggering events, guard conditions, and actions are described below. The basic constructs are introduced first, followed by some abbreviations.

3.3.1.2.1 Triggering Events

The triggering event of a transition is either a *world-change event* (WCE) or a time event. A WCE reflects a primitive change to the world state, such as the addition or removal of an object, or a change in the value of an object's attribute. A WCE can be expressed in one of the following basic forms, where **C** is a concept in the world model, and the WCE parameter **o** is an object instance of type **C**:

- $val(\mathbf{C+(o)})(WS)$ = object o has just been added to the world state (e.g., *Call+(o)*).
- $val(\mathbf{C\text{-}(o)})(WS)$ = object o has just been removed from the world state (e.g., *Call-(o)*).
- $val(\mathbf{C.a{\sim}(o)})(WS)$ = o's attribute a has just changed (e.g., *call.voice~(o)*).

A time event is of the form **after(t)**, which means that duration t has passed since the transition's source state was entered.

3.3.1.2.2 Guard Conditions

The guard condition, of a transition or action, is a predicate over the world state and any parameters of the transition's triggering event (see Section 3.2.4.1.4). For example, transition *t4* of feature *BCS* in Figure 7 has a guard that is based on the value of the triggering event's parameter o.

3.3.1.2.3 Actions

An action of a transition is a *world-change action* (WCA). A WCA reflects a feature's command to make a primitive change to the world state, such as adding or removing an object from the world state[4], or changing the value of an object attribute. A WCA can be expressed in one of the following basic forms. The expressions below refer to concepts **A** (an association), **M** (a message), and **C** (a concept that is not an association, message, feature, or SPL); they also refer to attributes ($\mathbf{a_i}$), roles ($\mathbf{r_i}$), parameters ($\mathbf{p_i}$), expressions ($\mathbf{exp_i}$), and object expressions ($\mathbf{o_i}$):

- $\mathbf{+C(a_1 = exp_1, ..., a_n = exp_n)}$ adds to the world state a **C** object whose attributes $\mathbf{a_i}$ have corresponding values $val(\mathbf{exp_i})(WS)$ (e.g., **+User()**).

[4]A WCA cannot add or remove an SPL or feature object. All constraints on the presence of SPL and feature objects in the world state are specified solely in the feature and world models.

- **+A($\mathbf{a_1}$ = $\mathbf{exp_1}$, ..., $\mathbf{a_n}$ = $\mathbf{exp_n}$, $\mathbf{r_1}$ = $\mathbf{o_1}$, ..., $\mathbf{r_m}$ = $\mathbf{o_m}$)** adds to the world state an **A** link whose attributes $\mathbf{a_i}$ have corresponding values *val*($\mathbf{exp_i}$)(*WS*) and relates objects *val*($\mathbf{o_j}$)(*WS*) ($1 \leq j \leq m$) in roles $\mathbf{r_j}$ (e.g., **+Call(caller = user, ...)**).

- **!M($\mathbf{p_1}$ = $\mathbf{exp_1}$, ..., $\mathbf{p_n}$ = $\mathbf{exp_n}$)** adds to the world state an **M** message object from *me* whose parameters $\mathbf{p_i}$ have corresponding values *val*($\mathbf{exp_i}$)(*WS*).

- **-exp** removes the objects **exp** and their dependent links from the world state. The dependent links of an object **o** are the links L that relate o, the links that relate the links in L, and so on. For example, in a world state of the world model in Figure 3, the dependent links of a *User* include all of the *Call* links that the *User* participates in, and all of the *Processing* links that that relate those *Call* links.

- **o.a := exp** changes the value of *val*(**o**)(*WS*)'s attribute **a** to value *val*(**exp**)(*WS*) (e.g., **call.voice := true**).

An object-adding WCA (i.e., the first three WCAs in the above list) can have a prefix of the form '**v** = ', which declares a variable **v** that stands for the added object; e.g.,

c = +Call(caller = user, ...)

Expressions in the other WCAs of the same transition can contain this variable.

3.3.1.2.4 Abbreviations

A transition label can be simplified using the following abbreviations:

- Navigation expressions that start from *me* need not explicitly include the prefix *'me.'*. For example, in the macro *user*, **Subscription.User** simplifies **me.Subscription.User**.

- WCE expressions can be simplified if they apply to specific objects in the world state (as opposed to applying to some arbitrary object that is passed in as the WCE parameter). For example, **v.voice~** simplifies **Call.voice~(o) [o = v]**.

- A message WCE can directly list its data parameters (rather than referring to the message object), and the transition's guard condition and actions can directly refer to those parameter values (rather than object o's parameters). For example,

  StartCall(target) [target = v] / +Call(callee = target, ...)

  simplifies

  StartCall+(o) [o.to = me and o.target = v] / +Call(callee = o.target, ...)

3.3.1.2.5 Presence Conditions

The presence or absence of optional features in a product's feature configuration affects the behaviours that exist in the product. For example, an *TelSoft* product will only have behaviours for accepting a second call, if the product has CW. In FORML, the dependence of product behaviours on the presence of features is captured by *presence conditions*. For example, in the composed model of *TelSoft*'s requirements, many transitions are guarded by the presence-condition *CW*, which means that the behaviours specified by those transitions only exist in an *TelSoft* product, if the product has CW. Presence conditions are specified per optional feature type. The presence condition P of a feature-type F is specified in F's feature module as a predicate that constrains the presence of F features in *me*; P is specified following the feature module's declaration in the form **feature F [P]** (e.g., **feature CW [CW]** in Figure 8).

3.3.2 Specifying Requirements as Enhancements

3.3.2.1 *TelSoft* Enhancements

We start by describing two *TelSoft* features, as a way of introducing some of the types of enhancement behaviour that we want to be able to model in FORML.

Figure 8 shows the feature module of the CW feature type (CW for short). CW can become active only when the subscriber is in a call with a voice connection. We capture this condition by modelling CW as new transitions to and from BCS's state *talking*. The second UML note in Figure 8 specifies the context of these transitions — namely, the region *BCS{main.inCall.callProcessing}* (where prefix *BCS{}* qualifies names that are local to the *BCS* feature module, *main* is the name of *BCS*'s top-level machine, *inCall* is the name of a top-level state within *main*, and *callProcessing* is the name of a region within *inCall*). The transitions specify the conditions for activating and deactivating the CW feature (all transitions besides *t2*), and specifies that when there is a call waiting, the user can toggle the voice connection between the two calls (transition *t2*).

In addition, there are a few planned interactions between CW and *BCS* that need to be modelled: CW overrides *BCS* when a second call arrives, by accepting the second call rather than removing it (transition *t1*); and when the voice-connected call is removed or when the subscriber hangs up, by establishing a voice connection with the second call (transitions *t4* and *t5*).

Figure 9 shows the feature module of the *CD* feature type (*CD* for short). *CD* delivers a caller's number to the subscriber. When capture this behaviour as a new action *a1* added to transition *t4* in *BCS*'s feature module.

Figure 10 shows the feature module of the *CDB* feature type (*CDB* for short). *CD* which prevents *CD* from delivering the subscriber's number to a callee. Whe capture this intended feature interaction by strengthening the guard of *a1* added by *CD* (as described above) with a condition stating that *CD*'s behaviour only applies if the caller is not subscribed to *CDB*.

3.3.2.2 Syntax

A set of feature-type requirements can be modelled as enhancements: that is, as structural extensions of other feature modules. For example, the requirements of CW are modelled

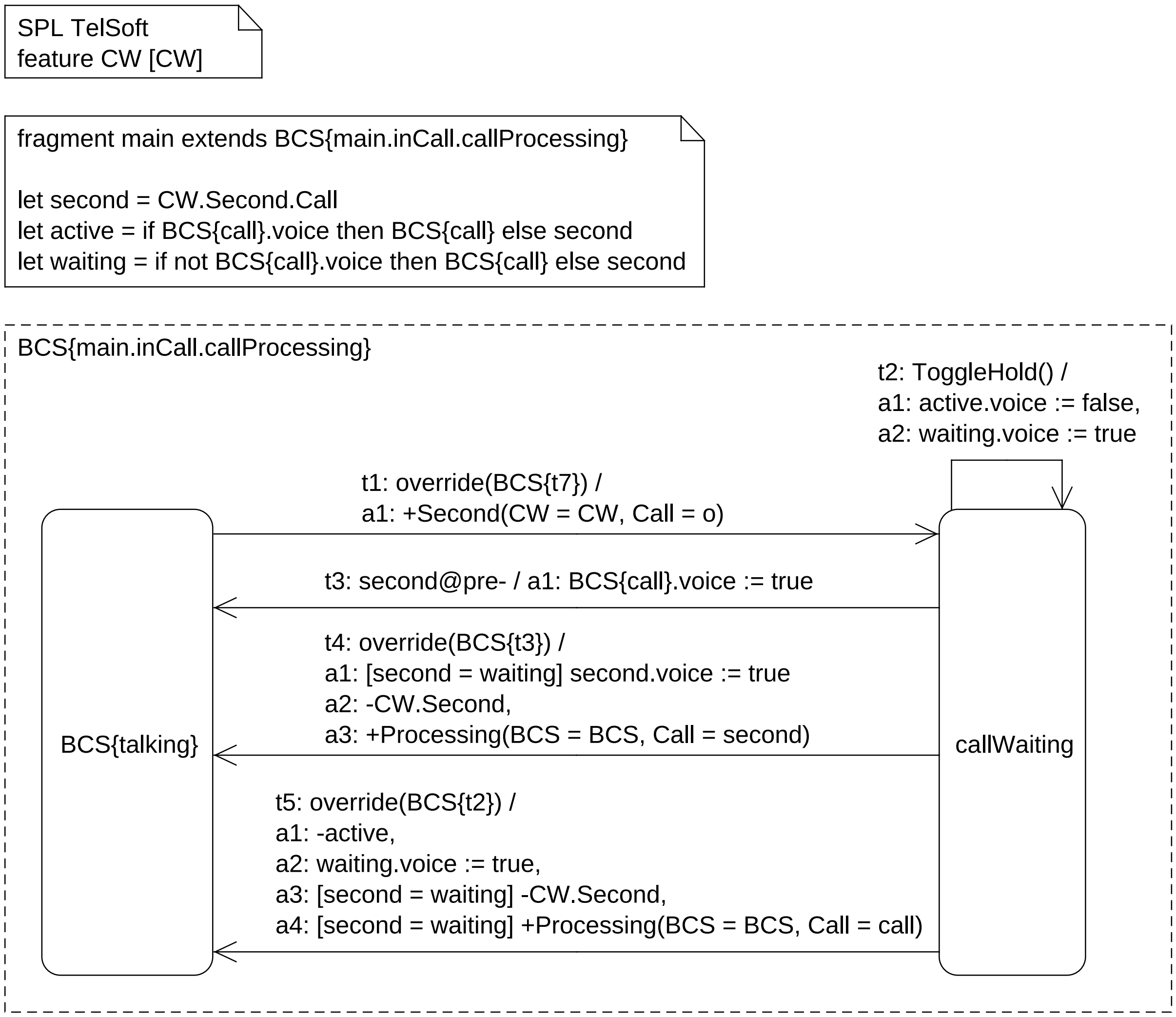

SPL TelSoft
feature CW [CW]
fragment main extends BCS{main.inCall.callProcessing}
let second = CW.Second.Call
let active = if BCS{call}.voice then BCS{call} else second
let waiting = if not BCS{call}.voice then BCS{call} else second
BCS{main.inCall.callProcessing}
t2: ToggleHold() /
a1: active.voice := false,
a2: waiting.voice := true
t1: override(BCS{t7}) /
a1: +Second(CW = CW, Call = o)
t3: second@pre- / a1: BCS{call}.voice := true
t4: override(BCS{t3}) /
a1: [second = waiting] second.voice := true
a2: -CW.Second,
a3: +Processing(BCS = BCS, Call = second)
BCS{talking}
callWaiting
t5: override(BCS{t2}) /
a1: -active,
a2: waiting.voice := true,
a3: [second = waiting] -CW.Second,
a4: [second = waiting] +Processing(BCS = BCS, Call = call)

Figure 8: CW feature module

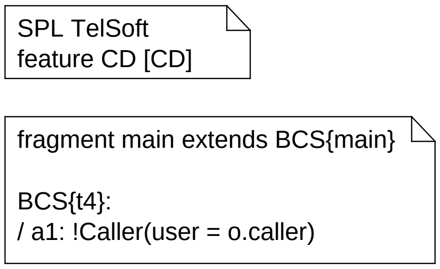

SPL TelSoft
feature CD [CD]
fragment main extends BCS{main}
BCS{t4}:
/ a1: !Caller(user = o.caller)

Figure 9: CD feature module

as extensions of *BCS*'s feature module. It is even possible to have extensions of extensions (e.g., the requirements of *CDB* are specified as extensions of *CD*'s extensions of *BCS*'s feature

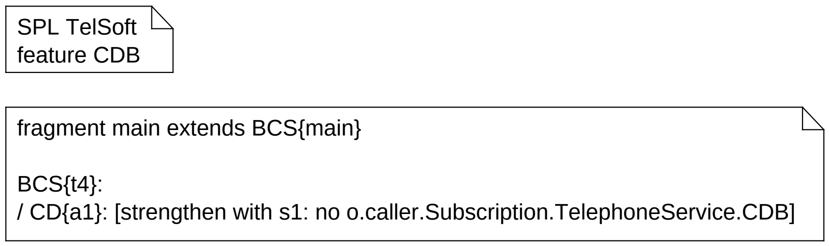

SPL TelSoft
feature CDB
fragment main extends BCS{main}
BCS{t4}:
/ CD{a1}: [strengthen with s1: no o.caller.Subscription.TelephoneService.CDB]

Figure 10: CDB feature module

module). Possible extensions include adding a transition to a feature machine, adding an action to a transition, and adding a condition to a transition or action guard. A new condition can be a *weakening* or *strengthening* clause: a strengthening clause is in conjunction with the clause that it strengthens, and a weakening clause is in disjunction with the clause that it weakens. As described below, strengthening clauses model intended interactions. A condition c with weakening clauses $\{w_1, \cdots, w_n\}$ and strengthening clauses $\{s_1, \cdots, s_m\}$ results in the expression $(c \wedge s_1 \wedge \cdots \wedge s_m) \vee w_1 \vee \cdots \vee w_n$. As mentioned above, enhancements can have enhancements. Suppose that an enhancement weakens a condition c by clause w, and that another enhancement strengthens clause w introduced by the first enhancement by clause s. The resulting expression is $c \vee (w \wedge s)$.

Enhancements in a feature module are structured into *fragments*, where each fragment specifies extensions in a particular context. The context of a fragment may be a feature machine, state, or region in another feature module. In order to specify the context of a fragment, we need to refer to elements in other feature modules[5]. The identifiers in a feature module F can be referenced by an expression of the form **F{txt}**, in which all identifiers in *txt* are interpreted to be within F. For example, CW's feature module has a single fragment, the context of which is *BCS{main.inCall.callProcessing}*: i.e., region *callProcessing* of state *inCall* of feature-machine *main* in *BCS*'s feature module. A fragment is prefaced with a UML note that contains its declaration (e.g., *fragment main extends BCS{main.inCall.callProcessing}*).

Enhancements can specify the addition, retraction, or replacement[6] of behaviours associated with the requirements of other feature types. The feature-module extensions that model each type are described separately in Sections 3.3.2.2.1, 3.3.2.2.2, and 3.3.2.2.3.

An intended interaction of a feature A with a feature B is modelled in A's feature module as an enhancement that removes or replaces behaviours associated with B's requirements. For example, CW interacts with BCS in that it overrides BCS's removal of an incoming call, while the subscriber in already in a call. This interaction is modelled in CW's feature module as

[5]The element names in a feature module are local to that feature module. The named elements of a feature module are feature machines, enhancement fragments, states, regions, transitions, actions, strengthening/weakening clauses, unspecified functions, constants, and macros. The name of an element must be unique in its scope: The scope of a feature machine or enhancement fragment is a feature module; the scope of a state is a feature machine or a region; the scope of a transition, unspecifed function, constant, or macro is a feature machine; the scope of an action is a transition; and the scope of a weakening/strengthening clause is a guard condition.

[6]Although behaviour replacement is essentially a combination behaviour retraction and addition, we consider it separately due to the close coupling between its components.

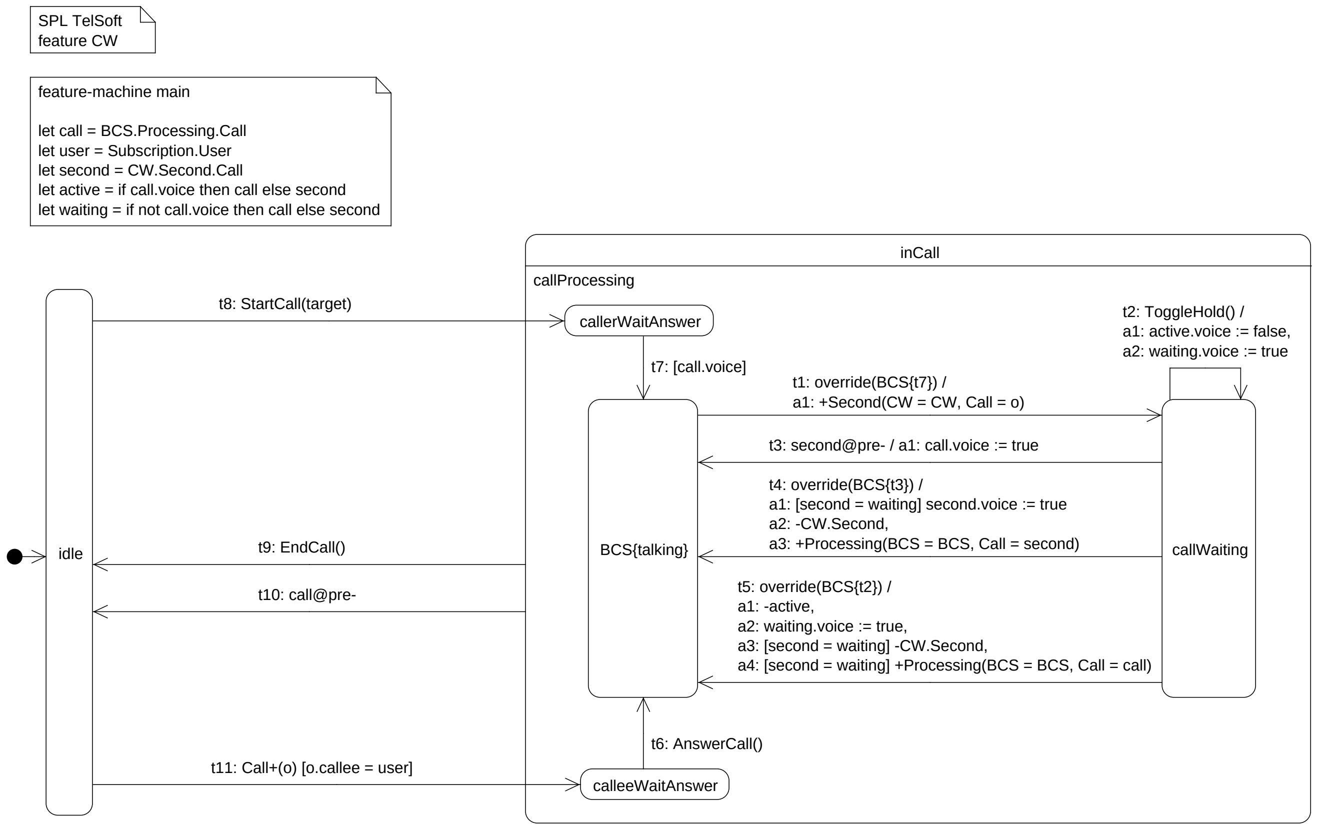

SPL TelSoft
feature CW
feature-machine main
let call = BCS.Processing.Call
let user = Subscription.User
let second = CW.Second.Call
let active = if call.voice then call else second
let waiting = if not call.voice then call else second
inCall
callProcessing
idle
callerWaitAnswer
BCS{talking}
callWaiting
calleeWaitAnswer
t8: StartCall(target)
t7: [call.voice]
t1: override(BCS{t7}) /
a1: +Second(CW = CW, Call = o)
t2: ToggleHold() /
a1: active.voice := false,
a2: waiting.voice := true
t3: second@pre- / a1: call.voice := true
t4: override(BCS{t3}) /
a1: [second = waiting] second.voice := true
a2: -CW.Second,
a3: +Processing(BCS = BCS, Call = second)
t5: override(BCS{t2}) /
a1: -active,
a2: waiting.voice := true,
a3: [second = waiting] -CW.Second,
a4: [second = waiting] +Processing(BCS = BCS, Call = call)
t9: EndCall()
t10: call@pre-
t6: AnswerCall()
t11: Call+(o) [o.callee = user]

Figure 11: CW feature module (modelled as a standalone machine)

an enhancement that overrides behaviours specified in *BCS*'s feature module. As a second example, *CDB* interacts with *CD* in that it removes *CD*'s behaviour of delivering the caller's number to a *CD* subscriber, if the caller is subscribed to *CDB*. This interaction is modelled in *CDB*'s feature module as an enhancement that removes behaviours specified in *CD*'s feature module.

In most cases, an enhancement can alternatively be specified as one or more feature machines that synchronize with machines in other feature modules. The constructs used for synchronizing feature machines and the cases where this alternative approach is applicable are described in Section 3.3.2.2.4.

3.3.2.2.1 Requirements Additions

Requirements additions add new behaviours (i.e., execution traces) to the behaviour model. New behaviours can always be modelled as a standalone machine, but this sometimes results in a larger and less-focused feature module than if the new behaviours are modelled as extensions to an existing feature's machine. For example, our model of feature *CW* in Figure 8 expresses the requirements of *CW* as enhancements to *BCS*. In contrast, Figure 11 shows a standalone version of *CW*, in which, the new behaviours are modelled by transitions *t1-t5*. However, even the discerning reader cannot distinguish between these transitions (which model new behaviours) from the other transitions (which model behaviours leading to the state in which the two new transitions are enabled). In general, whether to model a feature's behavioural requirements as standalone machines or as enhancements depends on whether a standalone representation emphasizes the new behaviours: if a standalone representation must also model the context of the new behaviours and if that part of the model is substantial, then it is better to model the new behaviours as enhancements.

Syntactically, requirements additions that are modelled as enhancements are expressed as model fragments (e.g., actions, transitions, regions) that are superimposed onto an existing feature module. A UML note specifies the feature machine(s) being enhanced and the *context* of the enhancements within those machines. Specifically, a context specification helps to indicate where in the enhanced machine the enhancements are to be attached. A context might be a transition (to which actions are added), or a state (to which a region is added), or a machine (to which a transition is added). Multiple enhancements can be modelled together, if they are expressed with respect to the same context. For an example, see the model of feature *voice mail* in Appendix A.

Requirements additions have one of the following forms:

- **A new region** that extends a state in the enhanced feature machine. In this case, new behaviours are expressed as a sub-machine that executes concurrently with sub-machines in the other regions of the extended state.

- **A new transition** that extends (possibly a sub-machine of) the enhanced feature machine. Adding a transition may also add new states as its source or destination states. A new transition t is represented graphically as a sub-machine fragment that includes t and its source and destination states (expressing as much of the state and region hierarchy as is necessary to represent the source and destination states within the specified context). For example, *CW* is specified as several transition extentions to *BCS*.

- **A new action** that extends a transition in the enhanced feature machine. Specifically, a new action id_1: $[c_1]$ a_1 that extends transition t is represented textually within a UML note:

  transition t: / $\mathbf{id_1}$: $[\mathbf{c_1}]$ $\mathbf{a_1}$

  For example, the *CD* extends a *BCS* transition with a new action.

- **A weakening clause** that extends the guard condition c of a transition or of an action. Weakening a condition results in the guarded transition or action being executed more frequently, and therefore leads to added behaviours.

  1. If the original c is a guard condition of a transition t, then the weakening clause is expressed as follows

     transition t: [weaken with id: d]

     where *id* is a name assigned to the weakening clause *d*, and *d* is a predicate expression over the world state.

  2. If the original c is a guard on an action a in transition t, then the weakening clause is expressed as follows

     transition t: / a: [weaken with id: d]

3. It is possible that a weakening clause weakens another (weakening or strengthening) clause, as opposed to weakening the full guard condition. In this case, the individual clause to be weakened must be specified. Consider the following series of weakening and strengthening clauses that extend the guard condition c of transition t:

   transition t: [weaken with id$_1$: k]
   transition t: [weaken with id$_2$: m]
   transition t: [strengthen id$_1$ with id$_3$: n]
   transition t: [weaken id$_3$ with id$_4$: p]

   The result is a guard $[c \vee (k \wedge (n \vee p)) \vee m]$ where the original condition c is weakened with clauses k and m; clause k is strengthened with clause n; and clause n is weakened with clause p. To ensure that the specification of the clause to be extended is unambiguous, it can be prefaced with the name of the feature that added that clause[7]. Thus, if the above four weakening clauses are added by features F, G, H, and J, respectively, then their full expressions would be:

   transition t: [weaken with id$_1$: k]
   transition t: [weaken with id$_2$: m]
   transition t: [strengthen F{id$_1$} with id$_3$: n]
   transition t: [weaken H{id$_3$} with id$_4$: p]

3.3.2.2.2 Requirements Retractions

Requirements retractions remove from the behaviour model behaviours associated with the requirements of other feature types. For example, *CDB* removes *CD*'s behaviours. Requirements retractions are realized by adding additional criteria to the preconditions of transitions or guards, so that they execute in fewer conditions. Because the behaviours to be removed are specified in other feature modules, a requirements retraction is necessarily specified as an enhancment.

Requirements retractions can be modelled as **A strengthening clause** that extends the guard condition of a transition or of an action. Such an extension is specified in the same form as for a weakening clause, except that the keyword **weaken** is replaced with **strengthen**. For example, *CDB* strengthens an action guard of *CD*.

3.3.2.2.3 Requirements Replacements

Requirements replacements replace behaviours associated with the requirements of other feature types. For example, *CW* replaces some *BCS* behaviours that concern processing a second call. Because the behaviours to be replaced are specified in other feature modules, a requirements retraction is necessarily specified as an enhancment.

Syntactically, requirements replacements can be modelled as the following extensions of an existing feature machine:

- **A new transition** that *overrides* a transition in the enhanced feature machine. It is possible but cumbresome to specify overrides without introducing new syntax. Consider

[7]Clause identifiers must be unique within a feature.

a new transition *t2* that overrides transition *t1* under condition c. First, *t2*'s enabling condition should include that of *t1*, so that *t2* is enabled whenever *t1* would be. Thus, if *t1* has a source state *s1* that is different from *t2*'s source state, a triggering event e, and a guard g, then *t2*'s label would be

t2: e [inState(s1) $\wedge$ g $\wedge$ c] / ...

where *inState(s1)* means that the feature machine is in state *s1* (discussed in Section 3.3.2.2.4). Second, *t1*'s guard should be strengthened, so that it never executes when *t2* is enabled. Thus, *t1*'s guard should be strengthened to $g \wedge \neg(inState(s2) \wedge c)$ where *s2* is *t2*'s source state. If *t1* and *t2* have the same source state, then the *inState* conjuncts are not needed. There are two problems with this approach to specifying overrides: (1) It is repetitious in that *t1*'s enabling condition is repeated within *t2*'s enabling condition. Repetition hinders modifiability: if *t1*'s enabling condition changes, that of *t2* will also have to be changed. (2) It does not explicate the intent of one transition overriding another. In the above specification, it is not clear that *t2* is intended to override *t1*.

To avoid these problems, we propose new constructs for specifying transition overrides:

- The **transition override** construct is used specify that one transition overrides another under some world-state condition. Syntactically, the label (up to the actions) of a new transition *t1* that overrides transition *t2* under condition c is expressed as follows:

  override(t2) [c] /

  Literally, this means that *t1* executes instead of t_2, provided that the feature machine is in *t1*'s source state and c is true. For example, one intended interaction of *CW* with *BCS* is specified by *CW{t1}* overriding *BCS{t7}*.
- The **transition priority** construct is used to specify a weaker form of override: given two transitions with independent enabling conditions, we can specify that one overrides the other when they are both enabled. This case differs from the above scenario, in that the overriding transtion's enabling condition does not necessarily include that of the overriden transition. Syntactically, the label (up to the triggering event) of a transition *t1* that overrides transition *t2* when they are both enabled is expressed as follows:

  t2 $>$ t1

- **A new action** that *overrides* an existing action in the same transition. Due to problems analogous to those for specifying transition overrides, we propose a new construct for specifying action overrides: A new action *a2* that overrides action *a1* under condition c is expressed as follows, up to *a2*'s WCA:

  a2: override(a1) [c]

  Literally, this means that *a2* executes instead of *a1*, provided c is true.

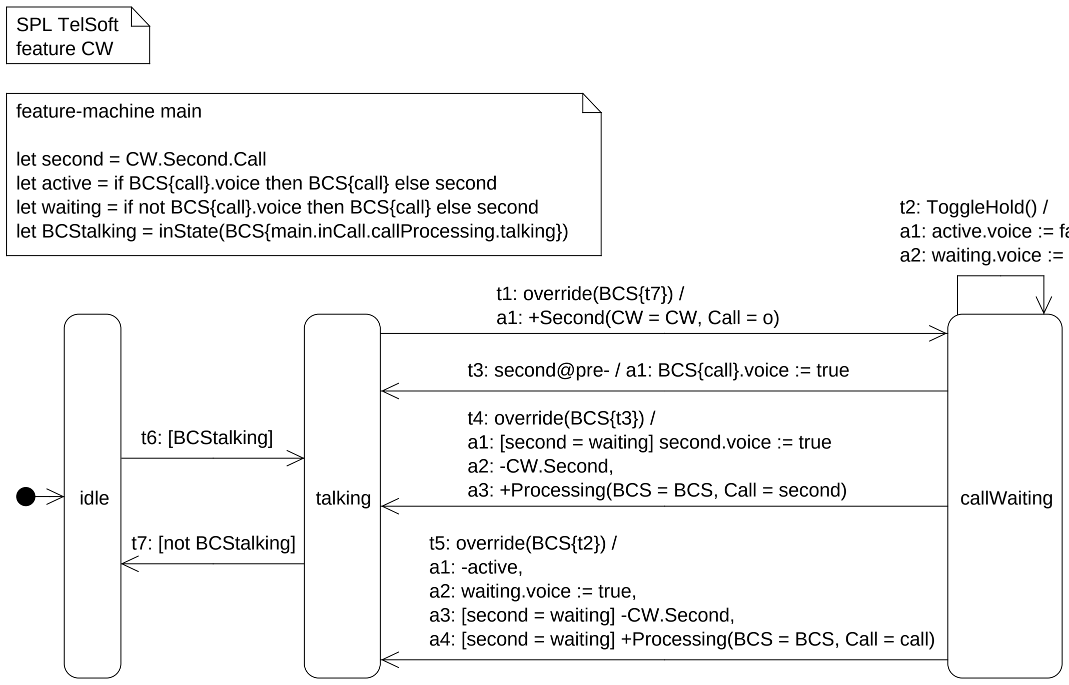

SPL TelSoft
feature CW
feature-machine main
let second = CW.Second.Call
let active = if BCS{call}.voice then BCS{call} else second
let waiting = if not BCS{call}.voice then BCS{call} else second
let BCStalking = inState(BCS{main.inCall.callProcessing.talking})
t2: ToggleHold() /
a1: active.voice := false,
a2: waiting.voice := true
t1: override(BCS{t7}) /
a1: +Second(CW = CW, Call = o)
t3: second@pre- / a1: BCS{call}.voice := true
t4: override(BCS{t3}) /
a1: [second = waiting] second.voice := true
a2: -CW.Second,
a3: +Processing(BCS = BCS, Call = second)
t6: [BCStalking]
idle
talking
callWaiting
t7: [not BCStalking]
t5: override(BCS{t2}) /
a1: -active,
a2: waiting.voice := true,
a3: [second = waiting] -CW.Second,
a4: [second = waiting] +Processing(BCS = BCS, Call = call)

Figure 12: CW feature module (modelled as an enhancement in terms of a feature machine)

3.3.2.2.4 Specifying Enhancements as Feature Machines

Most enhancements can be equivalently specified as one or more concurrent feature machines that synchronize with the feature modules being enhanced. For example, Figure 12 shows a feature module that expresses the requirements of *CW* as a single feature-machine *CW{main}* that synchronizes with feature-machine *BCS{main}*: the state *CW{main.talking}* is entered/exited whenever state *BCS{main.inCall.callProcessing.talking}* is entered/exited. The rest of *CW*'s requirements (including its retraction of *BCS*'s requirements) are modelled the same as that shown in Figure 8. In general, the size and focus of models of enhancements expressed as synchronizing feature machines are comparable to those expressed as feature-module extensions. Which approach is better depends on which model is clearer.

FORML introduces two synchronizing constructs:

- **Synchronization by state** makes behaviour in one feature machine conditional upon the state of another feature machine. To specify that a transition or action with guard g is performed only when another feature machine is in state s, we strengthen g with the predicate **inState(s)**; e.g.,

  inState(BCS{main.inCall.callProcessing.talking})

  .

- **Synchronization by transition** synchronizes one transition with a transition of another feature machine. To specify that transition *t1* synchronizes with a transition *t2* in another feature machine, we give *t1* the triggering event **when(t2)**, which means that *t1* is enabled when *t2* executes.

In most cases, requirements additions can be specified in terms of synchronizing feature machines, except for the weakening of conditions. A requirements replacement can be specified using the transition/action override constructs described in Section 3.3.2.2.3. A requirements retraction can also be specified as an override, where the overriding transitions/actions make no changes to the world state.

Although the override constructs were introduced as a means to synchronize feature machines, they can also be used within a singe machine to coordinate the behaviours in concurrent regions. The *inState* and *when* constructs can be used within a feature machine as well. Finally, all of the synchronization constructs above can be used in the specification of standalone requirements.

4 Composing Feature Modules

The requirements for a SPL are derived by composing the feature modules in the SPL's behaviour model. The composition yields an *integrated model* comprising a set of parallel feature machines that have been extended with fragments. Figure 13 shows part of the integrated model for the *TelSoft* SPL: *BCS*'s feature-machine *main* is extended with *CW*'s state *callWaiting* and transitions *t1-t5*; and *CD*'s action *a1*, which itself is strengthened by feature *CDB* with clause *s1*. The state and transitions extensions and the strengthening clause are shown in grey. Note that the integrated model uses global names (e.g., *CW*'s transition *t1* is named *CW{t1}*).

The presence condition of a feature module is manifested in the composed model in one of two ways: (1) The presence condition is added as a conjunct to all transitions and action guards introduced in the feature module; if a transition or action has no guard, the presence condition is added as the guard to the transition or action. For example, the presence condition *CW* of the *CW* feature module is added as a guard to transition *CW{t1}*, and the presence condition *CD* of *CD*'s feature module is added as a conjunct to the guard of action *CD{a1}*. (2) Any condition *c* (i.e., a guard or weakening/strengthening clause) that is introduced in the feature module is prefaced with the presence condition as the antecedent. The above rules can be overriden by adding a *$* symbol before the name of a transition, action, or weakening/strengthening clause (e.g., strengthening clause *$s1* in Figure 32 in Appendix A).

More precisely, the semantics of the composition of a pair of feature modules is the *superimposition* of the *feature-structure trees* (FSTs) of the feature modules. The structuring of the behaviour model as a composition of feature-modules with the above composition semantics is based on the feature-oriented software development algebra by Apel et al. [2]. Section 4.1 describes feature-module FSTs, and Section 4.2 describes how FSTs are superimposed.

4.1 Feature-Module FSTs

The FST of a feature module is an abstract-syntax tree of the feature module. Figure 14 shows a partial FST of *CW*. Each FST node specifies the global name (if any) and the type of feature-module elements, such as a state, a transition, an action, a condition, a clause within a condition, and so on. A *non-terminal* node represents a composite element whose structure is exposed as subtrees of the node. For example, the non-terminal node representing state-machine *BCS{main}* has separate subtrees that represent the state hierarchy and transitions of

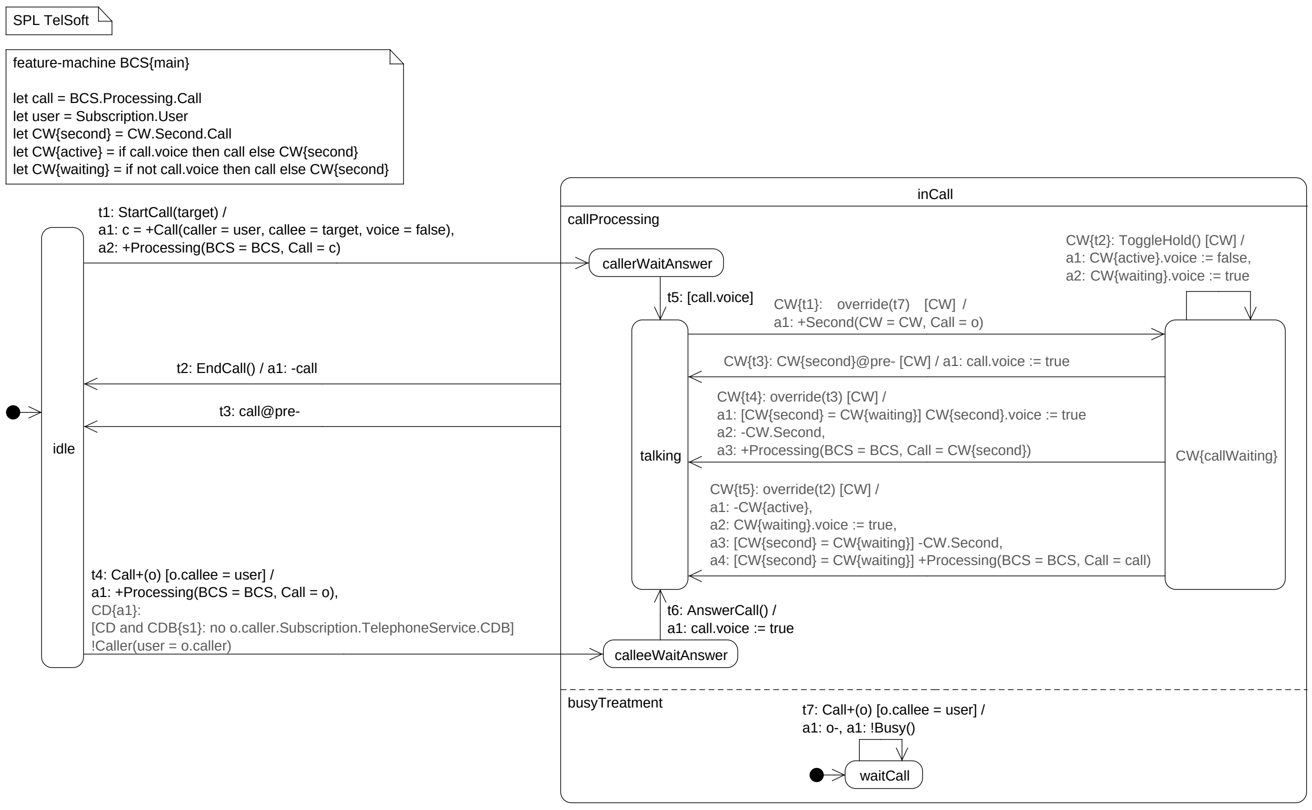

SPL TelSoft
feature-machine BCS{main}
let call = BCS.Processing.Call
let user = Subscription.User
let CW{second} = CW.Second.Call
let CW{active} = if call.voice then call else CW{second}
let CW{waiting} = if not call.voice then call else CW{second}
t1: StartCall(target) /
a1: c = +Call(caller = user, callee = target, voice = false),
a2: +Processing(BCS = BCS, Call = c)
inCall
callProcessing
callerWaitAnswer
CW{t2}: ToggleHold() [CW] /
a1: CW{active}.voice := false,
a2: CW{waiting}.voice := true
t5: [call.voice]
CW{t1}: override(t7) [CW] /
a1: +Second(CW = CW, Call = o)
CW{t3}: CW{second}@pre- [CW] / a1: call.voice := true
t2: EndCall() / a1: -call
t3: call@pre-
idle
CW{t4}: override(t3) [CW] /
a1: [CW{second} = CW{waiting}] CW{second}.voice := true
a2: -CW.Second,
a3: +Processing(BCS = BCS, Call = CW{second})
talking
CW{callWaiting}
CW{t5}: override(t2) [CW] /
a1: -CW{active},
a2: CW{waiting}.voice := true,
a3: [CW{second} = CW{waiting}] -CW.Second,
a4: [CW{second} = CW{waiting}] +Processing(BCS = BCS, Call = call)
t4: Call+(o) [o.callee = user] /
a1: +Processing(BCS = BCS, Call = o),
CD{a1}:
[CD and CDB{s1}: no o.caller.Subscription.TelephoneService.CDB]
!Caller(user = o.caller)
t6: AnswerCall() /
a1: call.voice := true
calleeWaitAnswer
busyTreatment
t7: Call+(o) [o.callee = user] /
a1: o-, a1: !Busy()
waitCall

Figure 13: Integrated *TelSoft* behaviour model ("..." in transition labels and regions elides portions of the model)

BCS{main}. A *terminal* node (which is shaded for easy identification) represents an element whose structure is atomic in the FST; the element's content is shown as text below the node (e.g., the triggering event of transition *CW{t1}*).

Note that a feature-module's FST captures not only elements introduced in the feature module, but also the context of such elements. The context of an element is represented by the path of nodes from the FST's root up to the node that represents the element. For example, the context of transition *CW{t1}* is the *BCS{main}* state machine, within the *TelSoft* SPL, within the behaviour model (named "machine").

The grammar for feature-module FSTs is shown in Figure 15. The *behaviour-model* and *SPL* nodes represent the grouping of state machines by SPL in the composite behaviour model. All of the remaining non-terminals are effectively *extension points* of a behaviour model; that is, they correspond to feature-module elements that can be extended by enhancements (e.g., states can be enhanced with new regions). Feature-module elements that cannot be enhanced are represented by terminal nodes (e.g., WCAs and predicates). The terminal **clause-type** represents the type of an enhancement clause, which can be either *weakening* or *strengthening*.

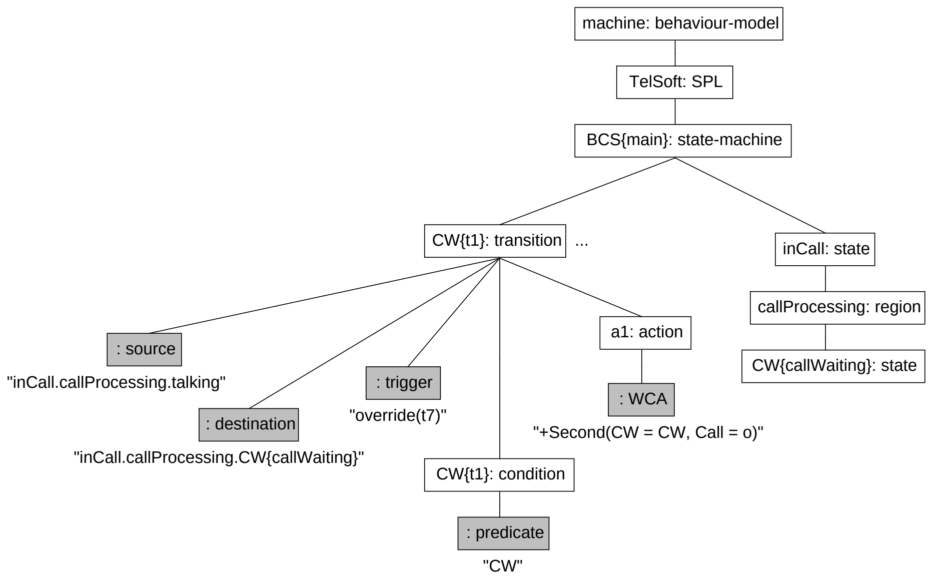


Figure 14: Partial FST of the *CW* feature module

behaviour-model ::= *SPL*+
SPL ::= *state-machine*+
state-machine ::= *state** **initial-state**? *transition*+ **macro***
state ::= *region**
region ::= *state*+ **initial-state**?
transition ::= **source**? **destination**? **trigger**? *condition*? *action**
action ::= *condition*? **WCA**?
condition ::= **predicate**? *condition** **clause-type**?

Figure 15: Abstract syntax of feature-module FSTs

4.2 Superimposition of Feature-Module FSTs

Two feature-module FSTs are superimposed by recursively merging their common non-terminal nodes, starting from the FSTs' root nodes; two nodes are merged if they have the same name and type[8]. When two nodes are merged, so are their non-terminal child nodes, where possible; both the merged and unmerged child nodes (including terminal child nodes) are added as children of the merged parent node.

The superimposition of feature-module FSTs is commutative and associative. A commutative and associative composition operator is desirable, becuase it avoids a class of unintended feature interactions that arise due to different composition orders yielding unexpectedly different behaviours [8, 5, 4]. Commutativity and associativity is ensured due to the fact that

[8]Because non-terminal nodes are matched in part by their names, it follows that all non-terminal nodes in an FST must be named. All extensible feature-module elements are explicitly named, except for the guard conditions of transitions and actions. To support composition, a guard condition is implicitly named after its transition or action; for example, the guard of transition *CW{t1}* is also named *CW{t1}*.

feature-module FSTs are inherently unordered[9]: the state-machines in a composite behaviour model, the regions of a composite state, and the actions of a transition are concurrent and are therefore unordered; the sibling states of a state-machine or region are inherenetly unordered; the order in which transitions execute is not given by their relative position in the FST; and finally, the order of the strengthening/weakening clauses applied to a condition does not matter, because the strengthening clauses are composed with the condition by conjunction, and the result is composed with weakening clauses by disjunction.

5 Conclusions and Future Plans

We have presented FORML, a language for modelling the requirements for features in a set of SPLs. A FORML model has interrelated views for describing the problem world and the requirements, in accordance with the Jackson and Zave RE reference model [6]. In addition, each feature's requirements are described separately in a feature module. The contribution of FORML is to integrate and adapt best practices for modelling behavioural requirements with techniques for decomposing artefacts into feature modules. The distinguishing aspects of FORML are (1) the inclusion of feature phenomena and feature configurations in an SPL's problem world; (2) a systematic treatment of how incremental features evolve a requirements model, with respect to added, removed, or replaced behaviours; (3) language constructs for explicitly modelling intended interactions among state-machine models of features; and (4) an operator for composing feature modules that preserves intended feature interactions, yet is commutative and associative.

Other advantageous properties of FORML include a UML-like syntax, to ease adoption; and the ability to model incremental features as model fragments, in order to focus on the feature's essential requirements.

We are currently investigating analyses of FORML models. We are interested both in detecting unintended interactions among features and in exploring the configuration space of a SPL. This work entails determining how unintended feature interactions manifest themselves in FORML models, so that we know what properties and patterns the analyses should look for. It also means modifying analysis tools to so that they adhere to our semantics of feature composition.

References

[1] S. Apel and C. Kastner. An overview of feature-oriented software development. *Journal of Object Technology*, 8, 2009.

[9]Superimposition of FSTs in general merges nonterminal nodes. However, it is some times desirable to merge terminal nodes. Two terminal nodes are merged using a rule defined for their node type. In these cases, the commutativity and associativity of FST composition depends on whether such merging rules are commutative and associative. However, in the superimposition of FORML's feature-module FSTs, merging terminal nodes is not allowed (matching terminal nodes signals an error in the model), and so commutativity and associativity is guaranteed by the fact that feature-module FSTs are unordered.

[2] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. An algebra for feature-oriented software development, 2007.

[3] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and L. Y-J. The feature interaction problem in telecommunication systems. In *Int. Conf. on Soft. Eng. for Telecommunication Switching Systems*, 1989.

[4] G. Bruns. Foundations for features. In *Feature Interactions in Telecommunications and Software Systems VIII*, 2005.

[5] 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.

[6] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *International Conference on Software Engineering*, pages 15–24, 1995.

[7] 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, Carnegie-Mellon University Software Engineering Institute, 1990.

[8] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction as a context sharing problem. In *Feature Interactions in Software and Communication Systems X*, 2009.

[9] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[10] P. Zave. Requirements for evolving systems: a telecommunications perspective. In *Proc. Fifth IEEE International Symposium on Requirements Engineering*, pages 2–9, 27–31 Aug. 2001.

A Extended FORML Model of *TelSoft*

This section presents the FORML model of an extended *TelSoft*. The feature model for extended *TelSoft* is shown in Figure 16. The additional feature types supported are *call-forwarding on busy (CFB)*, *call transfer (CT)*, *three-way calling (TWC)*, *group ringing (GR)*, *ringback when free (RBF)*, *teenline (TL)*, *terminating call screening (TCS)*, *voice mail (VM)*, *billing*, *reverse charging (RC)*, *split billing (SB)*. The world model for extended *TelSoft* is shown in Figure 17. The feature modules of *BCS*, *CW*, *CD*, and *CDB* are repeated in Figure 18, Figure 19, Figure 20, and Figure 21 for ease of reference.

Billing For every call, billing charges the caller's account, based on the designated billing rate and the duration of the call. The Billing feature module is shown in Figure 22.

Call forwarding on busy (CFB) CFB forwards calls received while the subscriber is in a call, to a designated user. The CFB feature module is shown in Figure 23.

Call transfer (CT) CT enables the subscriber to transfer an ongoing call to a second remote party by putting the first remote party on hold, establishing a second call, and then establishing a call between the first and second remote parties. The CT feature module is shown in Figure 24.

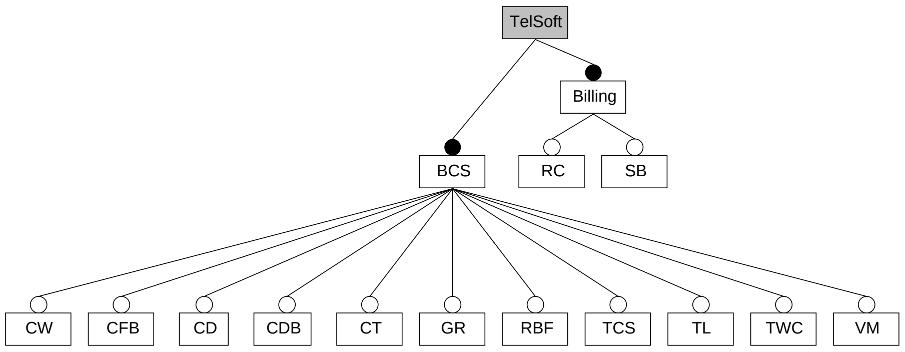


Figure 16: Extended *TelSoft* feature model

Group ringing (GR) GR replicates a call to the subscriber, to two designated users. As soon as the subscriber or either of the designated users accepts his/her call, the remaining two calls are dropped. The GR feature module is shown in Figure 25.

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. The GR feature module is shown in Figure 26.

Reverse charging (RC) RC charges the subscriber for incoming calls. The RC feature module is shown in Figure 27.

Split billing (SB) SB splits the charge between the subscriber and his/her callers based on a designated percentage. The SB feature module is shown in Figure 28.

Terminating-call screening (TCS) TCS blocks calls to the subscriber from designated users. The TCS feature module is shown in Figure 29.

Teenline (TL) If the subscriber makes a call request within a designated curfew period, TL requires authentication before starting the call. The TL feature module is shown in Figure 30.

Three-way calling (TWC) TWC allows the subscriber to bring a second remote party into a call by putting the first remote party on hold, connecting a second call, and then linking the two calls. The TWC feature module is shown in Figure 31.

Voice mail (VM) VM allows callers to leave messages for the subscriber when he/she does not accept the call within some timeout period, and allows the subscriber to check his/her messages. The VM feature module is shown in Figure 32.

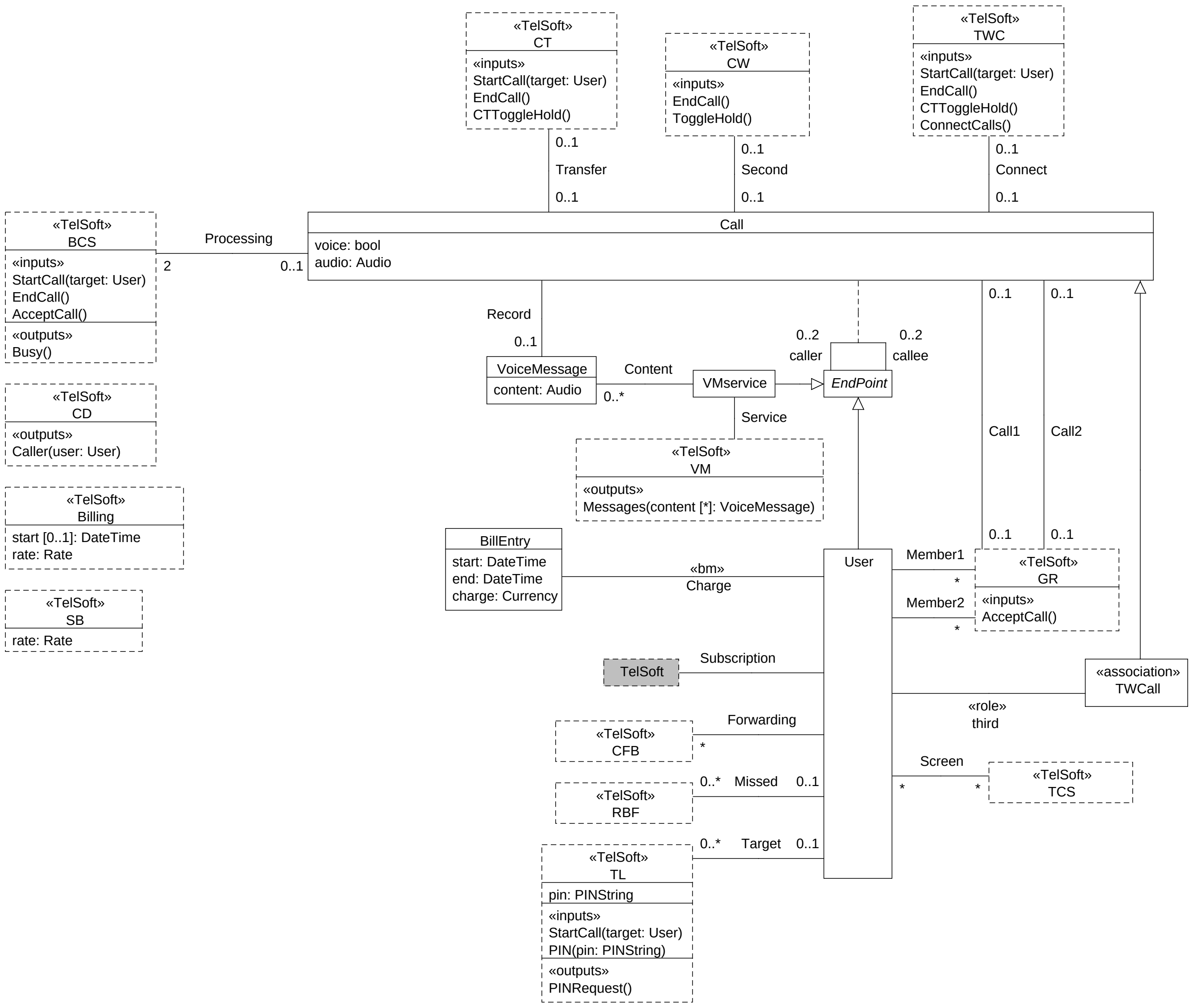


Figure 17: Extended *TelSoft* world model

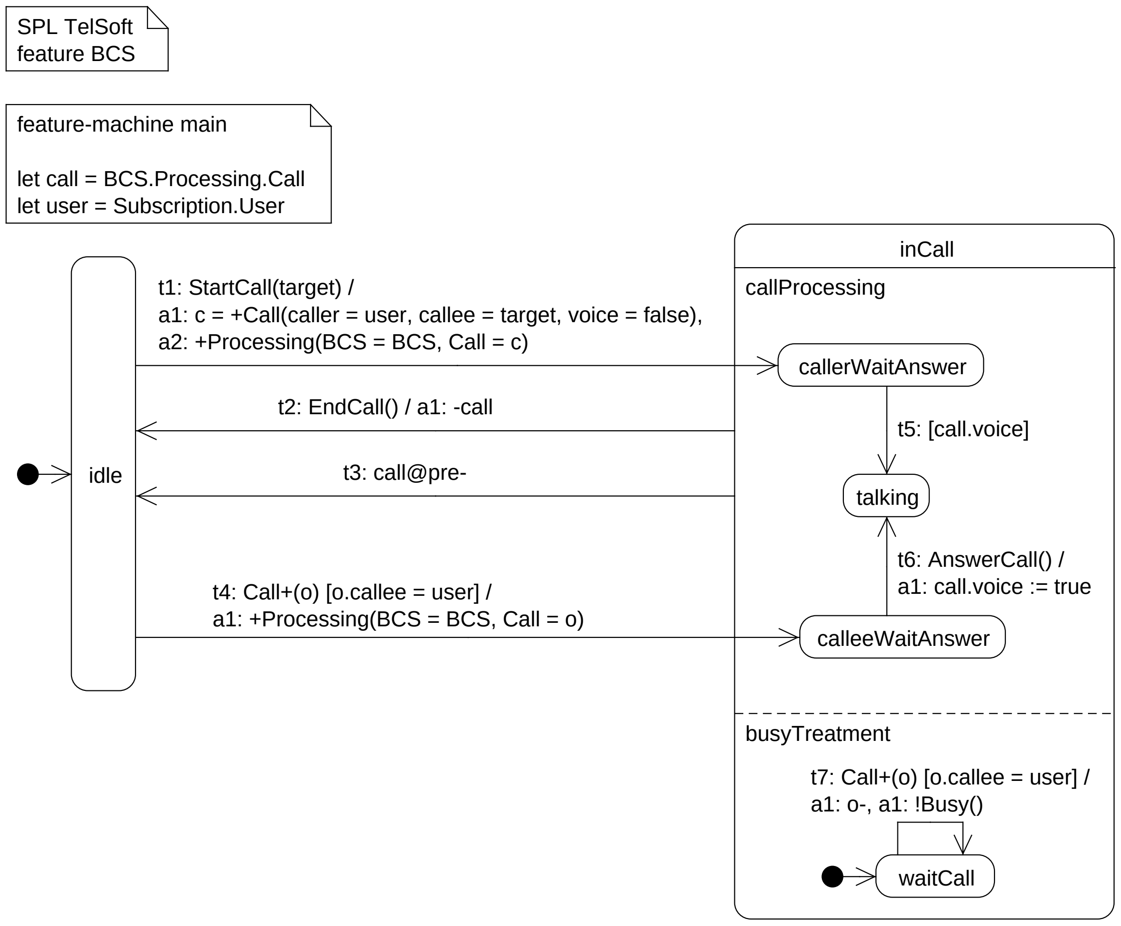
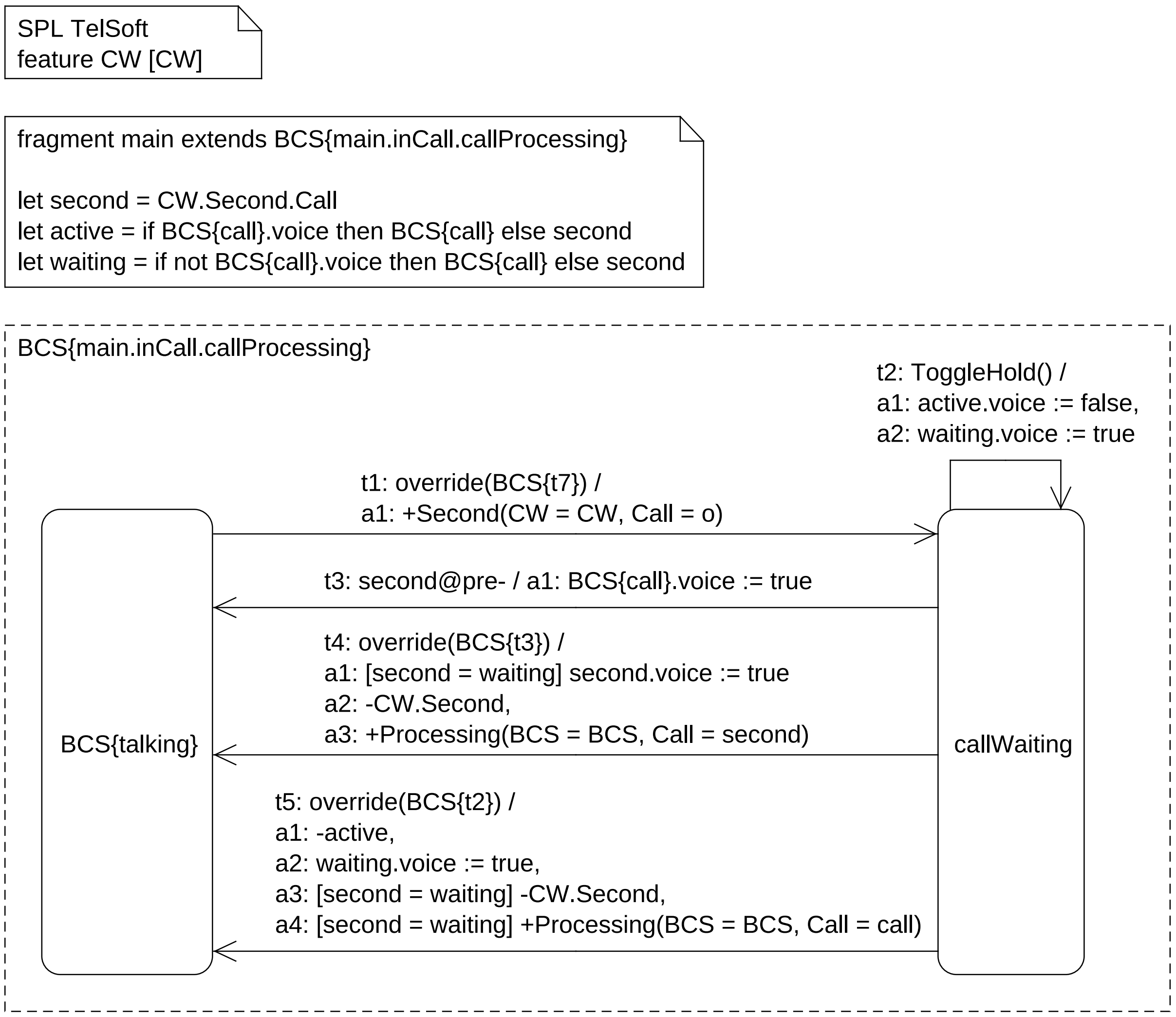

SPL TelSoft
feature BCS
feature-machine main
let call = BCS.Processing.Call
let user = Subscription.User
inCall
callProcessing
idle
t1: StartCall(target) /
a1: c = +Call(caller = user, callee = target, voice = false),
a2: +Processing(BCS = BCS, Call = c)
callerWaitAnswer
t2: EndCall() / a1: -call
t5: [call.voice]
t3: call@pre-
talking
t6: AnswerCall() /
a1: call.voice := true
t4: Call+(o) [o.callee = user] /
a1: +Processing(BCS = BCS, Call = o)
calleeWaitAnswer
busyTreatment
t7: Call+(o) [o.callee = user] /
a1: o-, a1: !Busy()
waitCall

Figure 18: BCS feature module

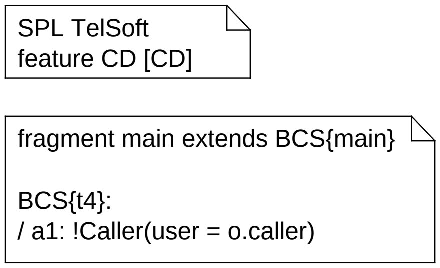


```
SPL TelSoft
feature CW [CW]
```

```
fragment main extends BCS{main.inCall.callProcessing}

let second = CW.Second.Call
let active = if BCS{call}.voice then BCS{call} else second
let waiting = if not BCS{call}.voice then BCS{call} else second
```

BCS{main.inCall.callProcessing}
t2: ToggleHold() /
a1: active.voice := false,
a2: waiting.voice := true
t1: override(BCS{t7}) /
a1: +Second(CW = CW, Call = o)
t3: second@pre- / a1: BCS{call}.voice := true
t4: override(BCS{t3}) /
a1: [second = waiting] second.voice := true
a2: -CW.Second,
a3: +Processing(BCS = BCS, Call = second)
BCS{talking}
callWaiting
t5: override(BCS{t2}) /
a1: -active,
a2: waiting.voice := true,
a3: [second = waiting] -CW.Second,
a4: [second = waiting] +Processing(BCS = BCS, Call = call)

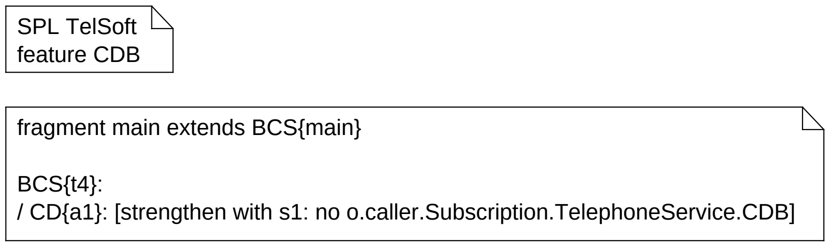
Figure 19: CW feature module



```
SPL TelSoft
feature CD [CD]
```

```
fragment main extends BCS{main}

BCS{t4}:
/ a1: !Caller(user = o.caller)
```

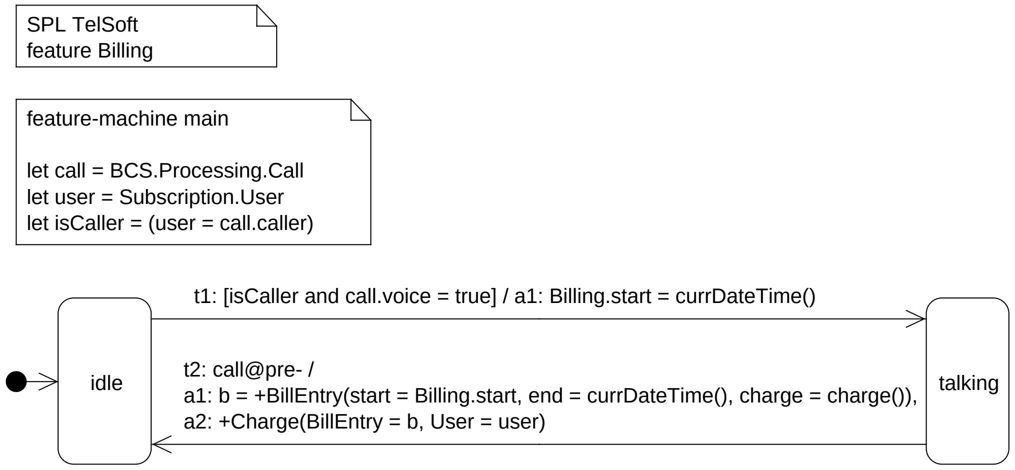
Figure 20: CD feature module



```
SPL TelSoft
feature CDB
```

```
fragment main extends BCS{main}

BCS{t4}:
/ CD{a1}: [strengthen with s1: no o.caller.Subscription.TelephoneService.CDB]
```

Figure 21: CDB feature module

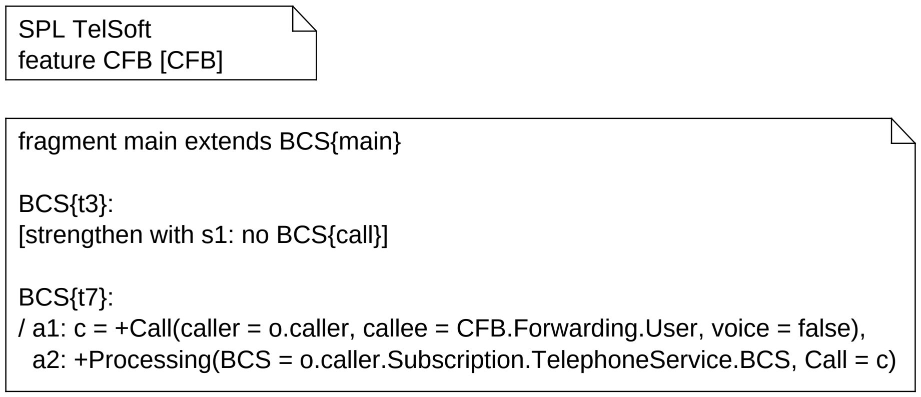


```
SPL TelSoft
feature Billing
```

```
feature-machine main

let call = BCS.Processing.Call
let user = Subscription.User
let isCaller = (user = call.caller)
```

idle
talking
t1: [isCaller and call.voice = true] / a1: Billing.start = currDateTime()
t2: call@pre- /
a1: b = +BillEntry(start = Billing.start, end = currDateTime(), charge = charge()),
a2: +Charge(BillEntry = b, User = user)

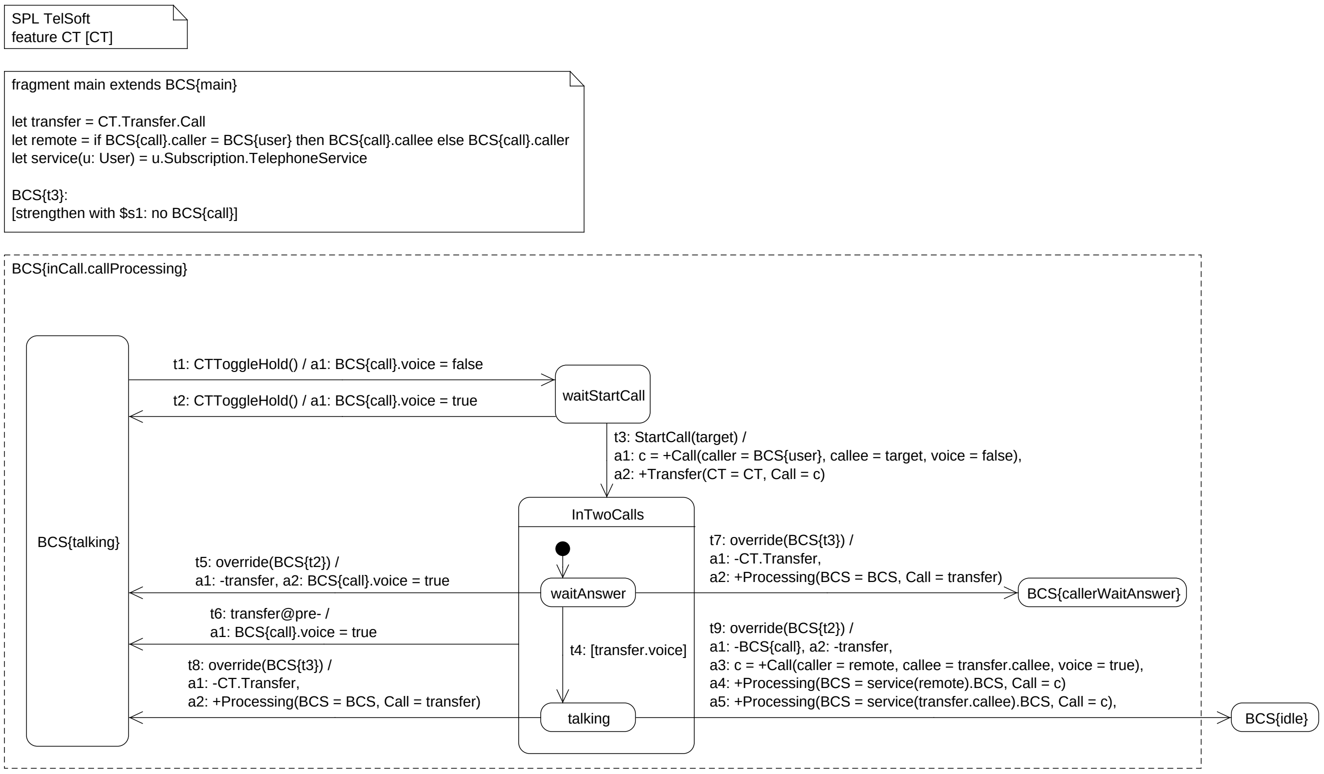
Figure 22: Billing feature module



```
SPL TelSoft
feature CFB [CFB]
```

```
fragment main extends BCS{main}

BCS{t3}:
[strengthen with s1: no BCS{call}]

BCS{t7}:
/ a1: c = +Call(caller = o.caller, callee = CFB.Forwarding.User, voice = false),
  a2: +Processing(BCS = o.caller.Subscription.TelephoneService.BCS, Call = c)
```

Figure 23: CFB feature module

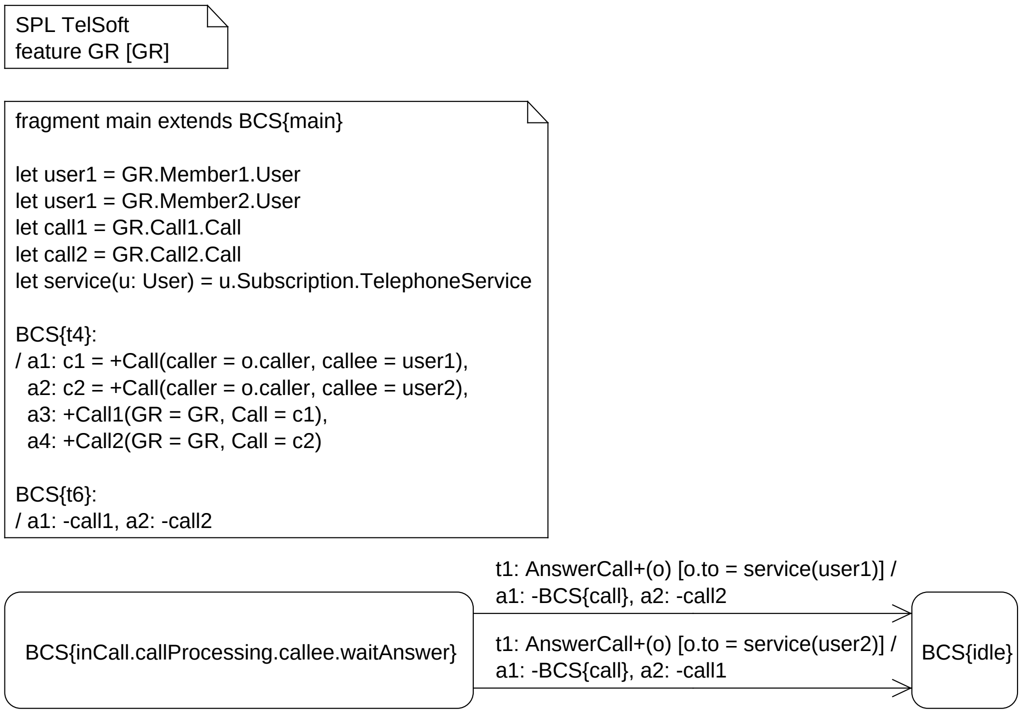


```
SPL TelSoft
feature CT [CT]
```

```
fragment main extends BCS{main}

let transfer = CT.Transfer.Call
let remote = if BCS{call}.caller = BCS{user} then BCS{call}.callee else BCS{call}.caller
let service(u: User) = u.Subscription.TelephoneService

BCS{t3}:
[strengthen with $s1: no BCS{call}]
```

BCS{inCall.callProcessing}
BCS{talking}
t1: CTToggleHold() / a1: BCS{call}.voice = false
t2: CTToggleHold() / a1: BCS{call}.voice = true
waitStartCall
t3: StartCall(target) /
a1: c = +Call(caller = BCS{user}, callee = target, voice = false),
a2: +Transfer(CT = CT, Call = c)
InTwoCalls
waitAnswer
t4: [transfer.voice]
talking
t5: override(BCS{t2}) /
a1: -transfer, a2: BCS{call}.voice = true
t6: transfer@pre- /
a1: BCS{call}.voice = true
t8: override(BCS{t3}) /
a1: -CT.Transfer,
a2: +Processing(BCS = BCS, Call = transfer)
t7: override(BCS{t3}) /
a1: -CT.Transfer,
a2: +Processing(BCS = BCS, Call = transfer)
BCS{callerWaitAnswer}
t9: override(BCS{t2}) /
a1: -BCS{call}, a2: -transfer,
a3: c = +Call(caller = remote, callee = transfer.callee, voice = true),
a4: +Processing(BCS = service(remote).BCS, Call = c)
a5: +Processing(BCS = service(transfer.callee).BCS, Call = c),
BCS{idle}

Figure 24: CT feature module

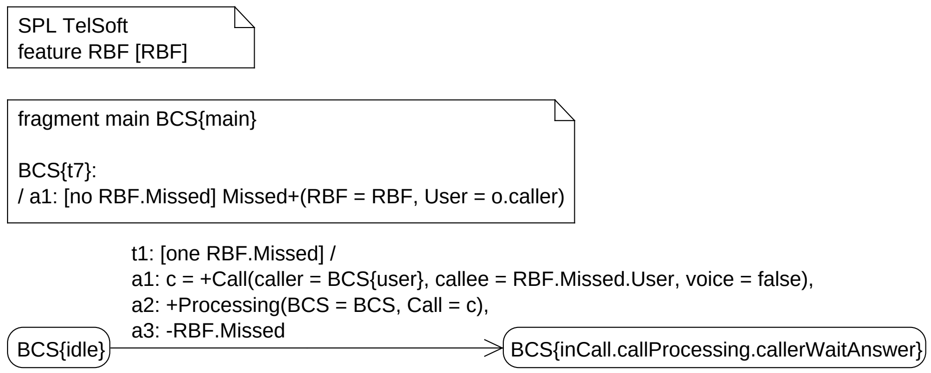


```
SPL TelSoft
feature GR [GR]
```

```
fragment main extends BCS{main}

let user1 = GR.Member1.User
let user1 = GR.Member2.User
let call1 = GR.Call1.Call
let call2 = GR.Call2.Call
let service(u: User) = u.Subscription.TelephoneService

BCS{t4}:
/ a1: c1 = +Call(caller = o.caller, callee = user1),
  a2: c2 = +Call(caller = o.caller, callee = user2),
  a3: +Call1(GR = GR, Call = c1),
  a4: +Call2(GR = GR, Call = c2)

BCS{t6}:
/ a1: -call1, a2: -call2
```

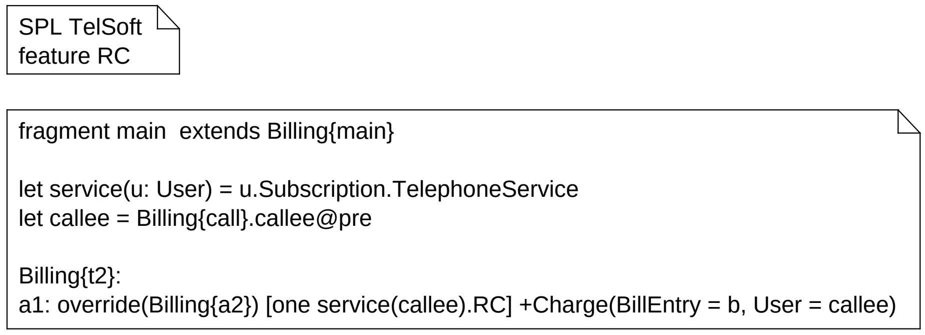
BCS{inCall.callProcessing.callee.waitAnswer}
t1: AnswerCall+(o) [o.to = service(user1)] /
a1: -BCS{call}, a2: -call2
t1: AnswerCall+(o) [o.to = service(user2)] /
a1: -BCS{call}, a2: -call1
BCS{idle}

Figure 25: GR feature module



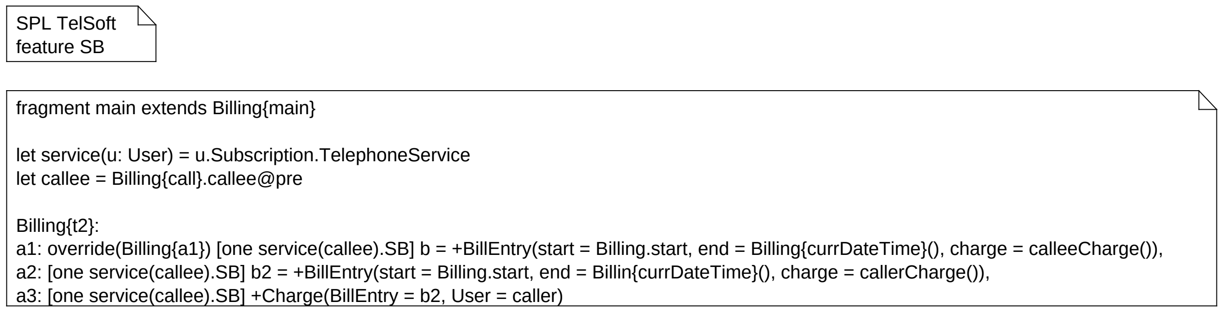
```
SPL TelSoft
feature RBF [RBF]

fragment main BCS{main}

BCS{t7}:
/ a1: [no RBF.Missed] Missed+(RBF = RBF, User = o.caller)

t1: [one RBF.Missed] /
a1: c = +Call(caller = BCS{user}, callee = RBF.Missed.User, voice = false),
a2: +Processing(BCS = BCS, Call = c),
a3: -RBF.Missed
```

Figure 26: RBF feature module



```
SPL TelSoft
feature RC

fragment main  extends Billing{main}

let service(u: User) = u.Subscription.TelephoneService
let callee = Billing{call}.callee@pre

Billing{t2}:
a1: override(Billing{a2}) [one service(callee).RC] +Charge(BillEntry = b, User = callee)
```

Figure 27: RC feature module



```
SPL TelSoft
feature SB

fragment main extends Billing{main}

let service(u: User) = u.Subscription.TelephoneService
let callee = Billing{call}.callee@pre

Billing{t2}:
a1: override(Billing{a1}) [one service(callee).SB] b = +BillEntry(start = Billing.start, end = Billing{currDateTime}(), charge = calleeCharge()),
a2: [one service(callee).SB] b2 = +BillEntry(start = Billing.start, end = Billin{currDateTime}(), charge = callerCharge()),
a3: [one service(callee).SB] +Charge(BillEntry = b2, User = caller)
```

Figure 28: SB feature module

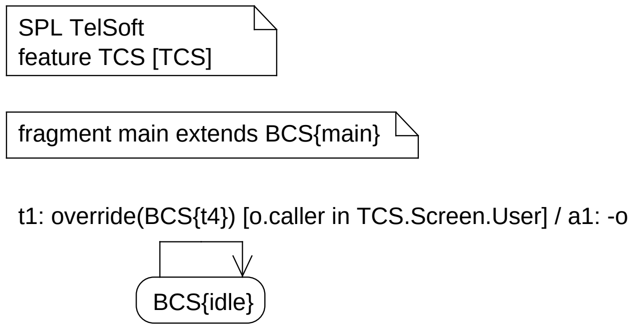

SPL TelSoft
feature TCS [TCS]
fragment main extends BCS{main}
t1: override(BCS{t4}) [o.caller in TCS.Screen.User] / a1: -o
BCS{idle}

Figure 29: TCS feature module

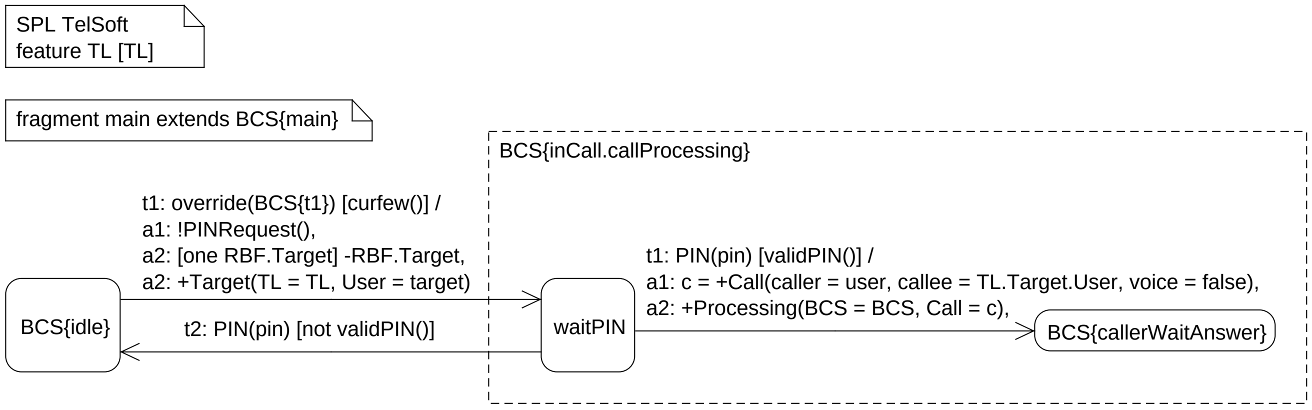
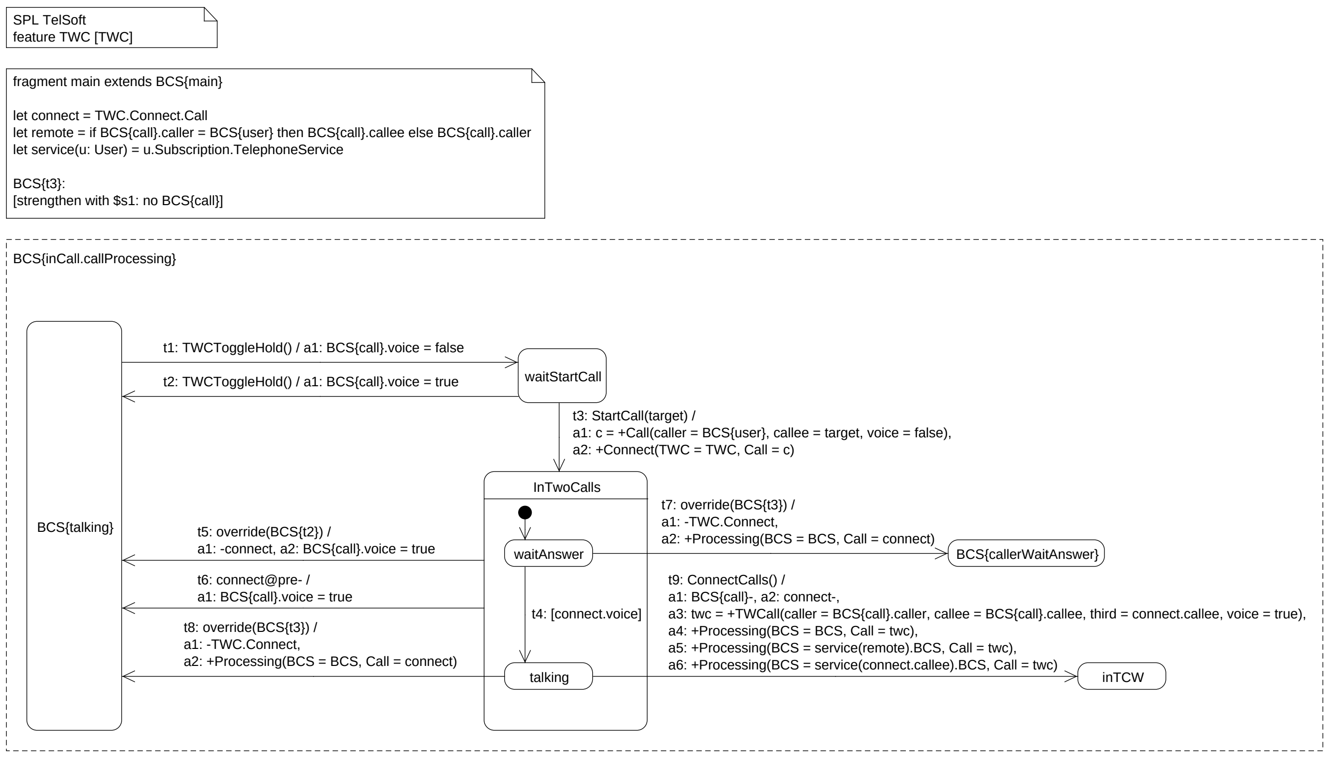

SPL TelSoft
feature TL [TL]
fragment main extends BCS{main}
BCS{inCall.callProcessing}
t1: override(BCS{t1}) [curfew()] /
a1: !PINRequest(),
a2: [one RBF.Target] -RBF.Target,
a2: +Target(TL = TL, User = target)
BCS{idle}
t2: PIN(pin) [not validPIN()]
waitPIN
t1: PIN(pin) [validPIN()] /
a1: c = +Call(caller = user, callee = TL.Target.User, voice = false),
a2: +Processing(BCS = BCS, Call = c),
BCS{callerWaitAnswer}

Figure 30: TL feature module

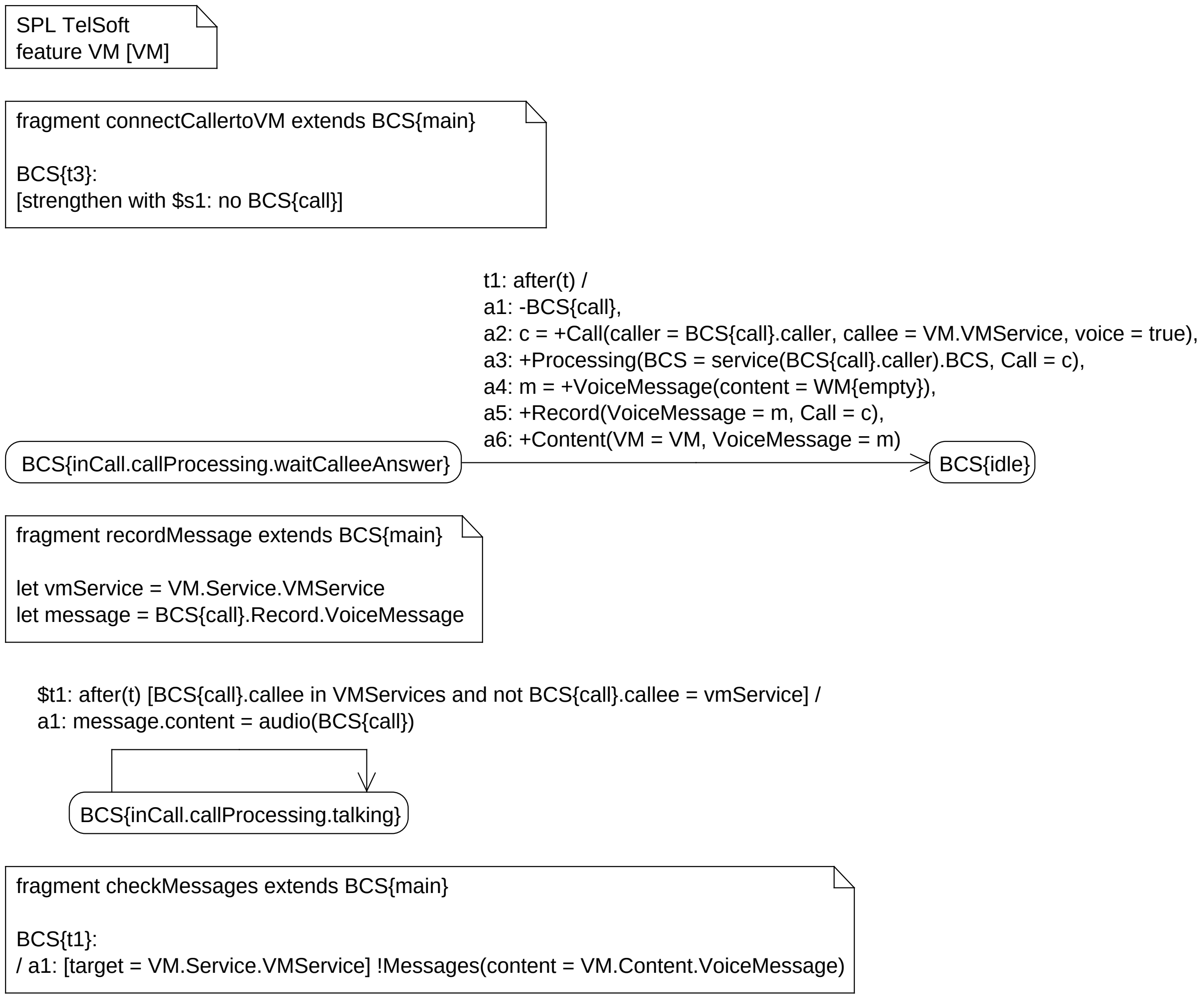


```
SPL TelSoft
feature TWC [TWC]
```

```
fragment main extends BCS{main}

let connect = TWC.Connect.Call
let remote = if BCS{call}.caller = BCS{user} then BCS{call}.callee else BCS{call}.caller
let service(u: User) = u.Subscription.TelephoneService

BCS{t3}:
[strengthen with $s1: no BCS{call}]
```

BCS{inCall.callProcessing}
BCS{talking}
t1: TWCToggleHold() / a1: BCS{call}.voice = false
t2: TWCToggleHold() / a1: BCS{call}.voice = true
waitStartCall
t3: StartCall(target) /
a1: c = +Call(caller = BCS{user}, callee = target, voice = false),
a2: +Connect(TWC = TWC, Call = c)
InTwoCalls
waitAnswer
t4: [connect.voice]
talking
t5: override(BCS{t2}) /
a1: -connect, a2: BCS{call}.voice = true
t6: connect@pre- /
a1: BCS{call}.voice = true
t8: override(BCS{t3}) /
a1: -TWC.Connect,
a2: +Processing(BCS = BCS, Call = connect)
t7: override(BCS{t3}) /
a1: -TWC.Connect,
a2: +Processing(BCS = BCS, Call = connect)
BCS{callerWaitAnswer}
t9: ConnectCalls() /
a1: BCS{call}-, a2: connect-,
a3: twc = +TWCall(caller = BCS{call}.caller, callee = BCS{call}.callee, third = connect.callee, voice = true),
a4: +Processing(BCS = BCS, Call = twc),
a5: +Processing(BCS = service(remote).BCS, Call = twc),
a6: +Processing(BCS = service(connect.callee).BCS, Call = twc)
inTCW

Figure 31: TWC feature module



```
SPL TelSoft
feature VM [VM]

fragment connectCallertoVM extends BCS{main}

BCS{t3}:
[strengthen with $s1: no BCS{call}]

t1: after(t) /
a1: -BCS{call},
a2: c = +Call(caller = BCS{call}.caller, callee = VM.VMService, voice = true),
a3: +Processing(BCS = service(BCS{call}.caller).BCS, Call = c),
a4: m = +VoiceMessage(content = WM{empty}),
a5: +Record(VoiceMessage = m, Call = c),
a6: +Content(VM = VM, VoiceMessage = m)

fragment recordMessage extends BCS{main}

let vmService = VM.Service.VMService
let message = BCS{call}.Record.VoiceMessage

$t1: after(t) [BCS{call}.callee in VMServices and not BCS{call}.callee = vmService] /
a1: message.content = audio(BCS{call})

fragment checkMessages extends BCS{main}

BCS{t1}:
/ a1: [target = VM.Service.VMService] !Messages(content = VM.Content.VoiceMessage)
```

BCS{inCall.callProcessing.waitCalleeAnswer}
BCS{idle}
BCS{inCall.callProcessing.talking}

Figure 32: VM feature module