Université Paul Sabatier Toulouse III

Master Informatique et Télécommunication
Parcours Système Informatique et Génie Logiciel

Année : 2010

Institut de Recherche en Informatique de Toulouse
Directeur : Luis Fariñas del Cerro

# *Formalisation sémantique et vérification de structures composites*

Auteur : Iulia - Elena Dragomir

Directeur de Recherche : Iulian Ober
Responsable du stage : Iulian Ober

# Résumé

Ce rapport introduit les structures composites dans le contexte du profile OMEGA UML pour représenter la structure hiérarchique des systèmes complexes, comme les systèmes temps-réel embarqués. Comme la norme UML est sous-spécifiée pour préserver la généralité du langage, on se propose de définir un ensemble expressif de notions et de règles pour la bonne modélisation et cohérence de structures composites. Une formalisation en OCL est mise en œuvre afin de pouvoir identifier les fautes de modélisation le plus souvent rencontrées dans des systèmes hiérarchiques. La formalisation en Isabelle/HOL est prévue pour prouver le typage sûr des structures composites qui respectent l'ensemble de principes proposés.

OMEGA UML est un profil dédié à la spécification et validation de systèmes temps-réel et embarqués, basé sur un sous-ensemble des éléments d'UML et intégré dans une plate-forme, IFx Toolset, qui propose des techniques comme la simulation, le model-checking et l'analyse statique pour la validation de systèmes. Du point de vue structurel le profil utilise les diagrammes de classes qui, à leur tour, contiennent des classes actives ou passives avec des attributs, relations (associations, compositions, héritage), opérations et machines à états. Du point de vue comportemental, le profil permet d'utiliser les opérations (primitives ou déclencheurs), les signaux, les machines à états et les actions (qui décrivent le corps d'une opération ou l'effet d'une transition). Des extensions pour modéliser le comportement temporel sont disponibles dans ce profil, par exemple les notions temporisateur et horloge. Les observateurs sont des objets spéciaux (classes stéréotypées « observer ») qui décrivent des propriétés dynamiques de sûreté du système et dans leur machine à états, un état peut être qualifié comme un état d'erreur pour exprimer la non-satisfaction de la propriété.

La mise au niveau de la norme UML à la version 2.x (de la version 1.4) introduits les structures composites. La nécessité des structures composites est donnée par le fait qu'elles sont un mécanisme puissant pour augmenter l'expressivité et la lisibilité des diagrammes de classes UML. Elles sont utilisées pour spécifier l'initialisation des structures complexes d'objets. En particulier, les systèmes temps-réel présentent des topologies hiérarchiques complexes et, surtout statiques, qui se prêtent bien à un modélisation avec des structures composites.

Une *structure composite* est une structure "d'éléments interconnectés qui sont créés en même temps qu'une instance du classifier contenant". Plus précisément, la structure composite est formée de *sous-composants* ou *parts* (instances de classes) et de *connecteurs* (liens de communication entre deux sous-composants, un sous-composant et un port ou deux ports). La norme UML fait la distinction entre: "assembly connector" qui fait la liaison entre deux sous-composants et "delegation connector" qui fait la liaison entre l'environnement de la structure composite et un sous-composant. Un connecteur peut être typé avec une association. Un *port* est un point d'interaction entre son propriétaire et l'environnement. Un port est défini par son contrat (une interface) qui lui permet de transférer les requêtes (opérations ou signaux) vers l'environnement, si son contrat requis (*required*), ou vers son propriétaire, si son contrat est fourni (*provided*).

Il est évident que tous les connecteurs syntaxiquement corrects ne sont pas valides.

Nous avons trouvé de problèmes de direction des ports et des connecteurs, de problèmes de typage des connecteurs et de problèmes sur le comportement d'un port. Un point essentiel est la définition d'un système de types permettant de caractériser chaque connecteur et chaque port.

Mais comme la norme laisse libres différents points de variation sémantique, on a identifié quelques inconsistances au niveau de la modélisation des structures composites et qui sont présentées ci-dessous avec la solution que nous avons adopté dans le nouveau profil OMEGA2.

1. **Directionalité des ports.** En UML, les ports sont bidirectionnels, c'est-à-dire qu'ils peuvent spécifier un ensemble de requêtes entrantes autorisées (l'interface *fournie*) et un ensemble de requêtes sortantes autorisées (l'interface *requise*). Pour modéliser un port bidirectionnel une relation de dépendance stéréotypé «Usage» doit être défini entre l'interface fournie et l'interface requise. Le type d'un tel port est sa interface fournie. Toutefois, le fait qu'un port puisse être bidirectionnel pose des problèmes de typage, par exemple lorsqu'un port est utilisé par le composant propriétaire pour envoyer des requêtes conformes à l'interface requise, le port doit être traité par le système de types comme une entité du type *requis*, même s'il est déclaré comme une entité du type *fourni*. D'un autre côté, quand on veut spécifier le comportement du port la machine à états doit gérer de requêtes provenant des deux sens, donc conformes aux deux interfaces: celle requise est celle fournie. Notre solution est d'interdire les ports bidirectionnels et les remplacer par deux port unidirectionnel (un spécifiant l'interface requise et l'autre l'interface fournie).

2. **Directionalité des connecteurs.** Dans le cas où on a un connecteur entre deux ports, sa direction est déterminée par la direction de ses ports. Mais la direction peut être inconsistante, par exemple si on lie un port requis d'un sous-composant avec un port fourni de la structure composite. Dans le cas d'un "delegation connector" qui lie deux ports, on demande que les deux aient la même direction (requis ou fourni). Dans le cas d'un "assembly connector" qui lie deux ports, on demande que l'un soit requis et l'autre fourni.

3. **Typage statique des connecteurs.** Dans le cas qu'un connecteur parts d'un composant c'est nécessaire de typer le connecteur avec une association, car sinon il n'y a aucun moyen pour le composant d'envoyer une requête à travers ce connecteur.

4. **Typage dynamique d'un connecteur.** Par défaut le comportement d'un connecteur est de transporter des requêtes (signaux ou appels d'opérations) d'une extrémité à l'autre. Pour déterminer les requêtes valides nous introduisons une règle basé sut le fait qu'un port peut requérir ou fournir un *ensemble* d'interfaces. Cette règle calcule l'ensemble des interfaces possibles à être transportées par un connecteur: l'ensemble des interfaces transportées en OMEGA2 est l'intersection des ensembles trouvés aux deux extrémités d'un connecteur.

5. **Plusieurs connecteurs partant du même port.** Il y a les cas quand plusieurs connecteurs on le même port comme point de départ et ils transportent la même interface. Pour lever l'ambigüité, OMEGA2 demande qu'au moins un connecteur soit typé par une association et le comportement du port soit explicitement spécifié par une machine à état.

6. **Comportement du port.** Le comportement d'un port est de transférer les requêtes vers son propriétaire ou l'environnement solen sont type et il peut être spécifié par une machine à états. Cela implique que le comportement soit en mesure de faire référence aux connecteurs qui ont une extrémité dans ce port. La sémantique définit des associations implicites qui permettent d'accéder au connecteurs qui ne sont pas typés avec une association. En plus, on demande que les ensembles des interfaces transportées sur n'importe quels deux connecteurs différents partant du même port et non-typés avec une association soient disjoints. Pour la complétude, on demande que la réunion de tous les ensembles des interfaces transportées soit égale à l'ensemble des interfaces du port.

Au final, l'ensemble de règles définissent un langage bien typé pour lequel nous sommes intéressé de prouver le typage sûr. Ça veut dire que chaque requête atteint sa destination finale et que chaque composant reçoit que de requêtes compatible avec ses interfaces.

Toutes les règles présentées ci-dessus sont formalisées en OCL pour capturer les fautes de modélisation les plus sensibles par rapport à nos contraintes dans les modèles OMEGA2. Cet ensemble de règles est utilisé pour la validation statique en phase compilation du modèle, avant la transformation vers un modèle IF pour être validé du point de vue comportemental. En appliquant l'ensemble des principes dans des modèles réels, nous avons trouvé des fautes concernant le typage dynamique des connecteurs (connecteurs qui ne transportent pas des requêtes) et le comportement de ports (la même interface appartient à deux ensembles des interfaces transportées qui doivent être disjoints; une interface requise n'a pas un correspondant à l'autre extrémité du connecteur et une interface fournie n'est pas réalisée par le propriétaire du port).

La formalisation Isabelle/HOL a comme but de prouver que les structures composites qui respectent l'ensemble de règles ont un typage sûr. Nous avons défini une syntaxe abstraite pour représenter le profil dans cet assistant de preuves et nous avons ajouté de règles pour que la syntaxe abstraite corresponde au profile. Les règles présentés sont formalisés à l'aide de fonctions récursives. Le typage sûr n'a pas encore été prouvé, à cause de la preuve de terminaison des ces fonctions récursives qui ne sont pas totales.

Pareil à la version précédente du profil OMEGA, celle présentée ici est soutenue d'une boîte à outils (IFx2) qui offre de possibilités pour la simulation et vérification d'un modèle.

# Contents

# List of Figures

# Introduction

This report introduces a new version of the OMEGA UML Profile, an extension based on composite structures. Composite structures were introduced starting with the version 2.0 in the UML standard and represent a big evolution in the representation of complex hierarchical systems. Because the UML standard is under-specified in order to preserve the generality of the language, various ambiguities are introduced in the model when using composite structures. Our purpose is to define an expressive set of notions and principles to clarify the composite structures at the modelling level and also at the execution model level. Our rule set can be applied to other component based systems, like SysML or MARTE. All the principles were formalized in OCL in order to catch the most frequent modelling issues regarding composite structures. Isabelle/HOL formalization was developed for proving the type-safety of composite structures observing this rule set.

As already mentioned, our interest in composite structures is given by the powerful expressiveness of these constructs when modelling the architecture of hierarchical systems. Common applications are real-time embedded systems. This kind of systems are scattered through a large number of domains like avionics, aeronautics, consumer electronics and many others. An important research topic is to prove their safety.

The context of our work is the previous OMEGA UML Profile, dedicated to the specification and validation of real-time embedded systems. This profile is based on a subset of UML 1.4 elements for modelling the structure and the behaviour of a system and has as extensions time modelling and observers (elements that express safety properties of a model). The profile is integrated in a platform, the IFx Toolset, which proposes validation techniques like model-checking, simulation and static analysis via a translation to the intermediate IF representation.

# Chapter 1

# An overview of OMEGA Profile and tools

The OMEGA Profile ([20]) identifies a subset of the UML language which is sufficiently expressive for modeling the structure and behavior of real-time systems, and for which an operational semantics is defined by closing the relevant semantic variation points left open in the UML standard ([8]). This profile it is integrated within a framework (the IFx toolset [4]) that supports techniques like static analysis, model checking and simulation for validating real-time embedded system.

## 1.1 The OMEGA UML Profile

The starting point of this work is the OMEGA UML Profile. The previous version is based on a subset of UML 1.4. From structural point of view the profile consists in :

- **Classes.** They can be *active* or *passive*, partitioning the object space in *activity groups*. Each instance of an *active* class[1] defines an *activity group*. Each instance of a *passive* class belongs to one single *activity group*, the one that has created it. They can own attributes, relationships, operations and state machines. Activity groups are considered concurrent and they react to external stimuli (like signals and operation calls) in a run-to-completion manner. When a request is received from the outside environment, it is stored in group's queue and is treated later when the group is stable. By *stable* we mean that every object owned by the group has no spontaneous transitions (transitions that are guarded by a boolean condition and have no trigger) or pending operations from inside the group (i.e., the *object* is *stable*).

- **Structural features.** Classes have attributes which can have *predefined types*: Integer, Real, Boolean, or *reference types*. Since the OMEGA UML Profile was developed for modelling real-time embedded systems, two extensions representing time (*timer*

[1] Active classes are represented with a thick border to distinguish them for passive classes.

and *clocks*) have been included in the profile. *Timer* objects measure durations. They may be set to a relative deadline and can be reset. Upon deadline different operations may be executed by objects: sending a signal, calling an operation, etc. *Clock* objects measure also durations, but their values can be consulted by other objects.

- **Relationships.** The relationships that can be defined between classes are *associations* and *generalisations*. The associations supported are *simple* or *compositional*.

An example of an OMEGA UML model (structural model) is shown in Figure 1. The modelled problem states that a system is composed from a *Client* and a *Server* between which a communication path exists. The Client is interested in how many packets the Server accepts so that no information is lost on the way. For that it sends an initial *request* with the number of packets equal to 8. If the server has the capacity to accept 8 packets at once it answers with *ack*; otherwise the Server ignores the message received (this behaviour simulates that some packets are lost on the communication path). If in 10 units of time, the Client does not receive an answer from Server then this sends another *request* with a halved number of packets. A test is considered finished if the Server answers with *ack* or if the Client sends a request with the number of packets equal to 1 and no answer is provided. In the last case, the Client restarts testing the connection. This model is the running example of this chapter.

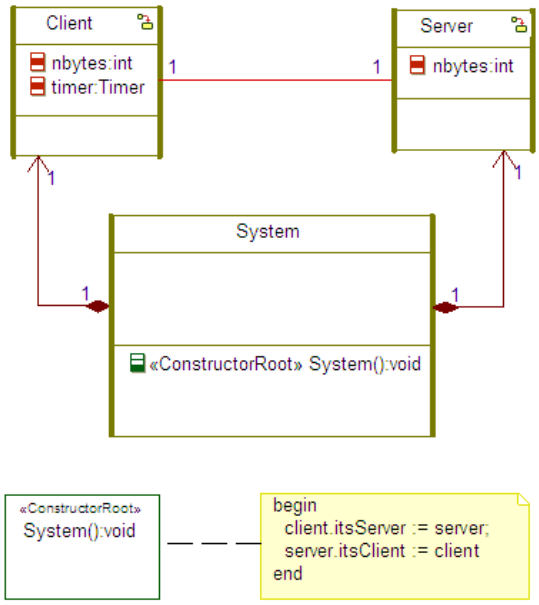


Figure 1: Structural OMEGA UML model for Client-Server application

From behavioural point of view, the OMEGA UML contains:

- **Operations.** We distinguish two types of operations: *primitive* and *triggered*. *Triggered operations* are a special kind of transition trigger: the call of such an operation

enables the transition which this guards. *Primitive operations* are similar to the methods in object oriented programming languages: they are subject to polymorphism and dynamic binding because of the inheritance relationship that may be defined between classes. They can own a body which is described by an action. When an operation is called by an object from the same activity group, the call is handled immediately by the object called using a call stack. If the call is made from another activity group, then it is queued by the receiving group and handled in a later run-to-completion step.

- **Signals.** They are the second method for asynchronous communication between objects and are usually used for triggering actions in the state machine of the target object. They can have parameters and they differ from triggered operation in the sense that they cannot have a return value. Signals always pass through object's activity group and are handled in a later run-to-completion step, no matter if the target is in the same activity group as the sender or not.

- **State machines.** They describe the behaviour of a class in term of states, transitions, triggers, actions, etc.

- **Actions.** They describe the effect of a transition in a state machine or the body of an operation. The OMEGA UML Profile introduces a textual action language, OMAL, compatible with the action metamodel of UML and which covers notions like: object creation/destruction, operation calls, expression evaluation, variable assignments, signal output, return action and control flow structuring statements (if-then-else and do-while).

The behaviour of classes for the problem stated above is shown in Figure 2[2].

The extension for time modelling covers prescriptive modelling and descriptive modelling. *Prescriptive modelling* of time defines the two notions presented above: *timers* and *clocks*. *Descriptive modelling* expresses hypothesis or requirements of the system. It allows us to identify events in a system execution, to express duration constraints between events which appear in the system and to express notions as resources, execution time and priorities.

Besides the timing extension, OMEGA UML Profile introduces the notion of *UML Observers*. They are special objects that monitor the system respectively its states and its events. Observers are modelled by classes stereotyped with `<<observer>>`. They have local memory and a state machine describes their behaviour. The states qualified as `<<error>>` states can be used in the model in order to express the satisfaction or the non satisfaction of a safety property. Observers may access any part of UML model's state (object attributes and states, signal queues) and they may use clocks to express timing properties. So, special events have been defined for observers in order to meet their purpose, events related to:

[2]The semantics used for actions is conform to OMEGA Action Language. `itsClient!ack()` corresponds to sending the `ack` signal to the target `itsClient`.

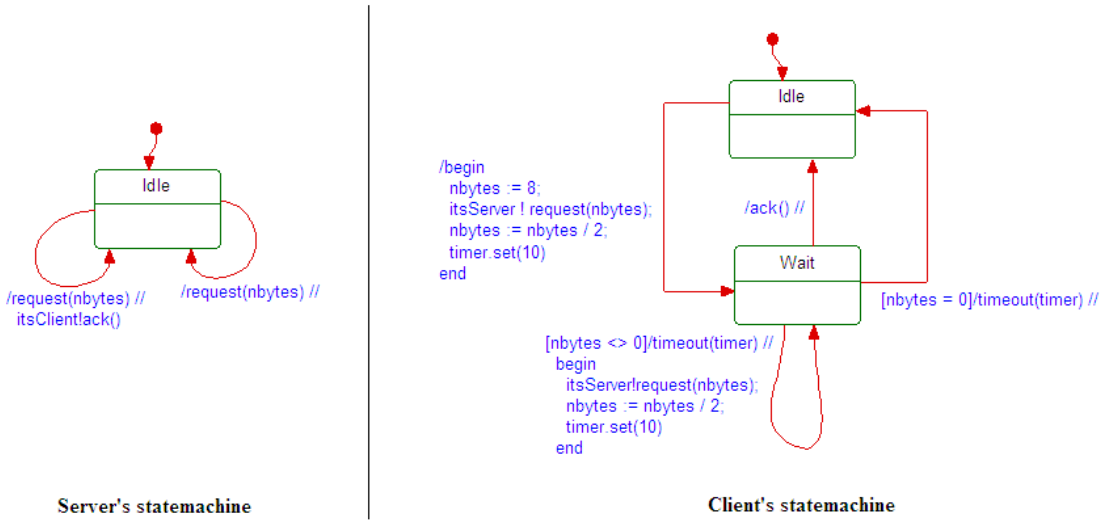


Figure 2: Behavioural OMEGA UML model for Client-Server application

- signal exchange: **send**, **acceptsignal**, **receivesignal**;
- operation calls: **invoke**, **receive** (reception of call), **accept** (start of the actual processing of call), **invokereturn** (sending of a return value), **receivereturn** (reception of the return value), **acceptreturn** (consumption of the return value);
- execution of actions or transitions: **start**, **end**, **startend**;
- timers: **occur**, **timeout**, **set**, **reset**.

The trigger of an observer transition is a **match** clause specifying the type of the event (previously presented), some related information (for example the operation name) and observer variables that may receive related information (variables receiving the values of the signal/operation call parameters). Figure 3 presents an UML Observer for the running example: *the connection between the Server and the Client is well established*. This means that the two entities can communicate and the Server has to provide a valid answer to the Client (with the number of accepted packets) from the first executed test. Otherwise our property is false and the *Error* state is reached.

## 1.2 The IF Language

The IF language is based on communicating extended timed automata. It involves processes that may be created and destroyed dynamically, running in parallel and communicating through messages. The IF language offers a description for the structure and the behaviour of the system.

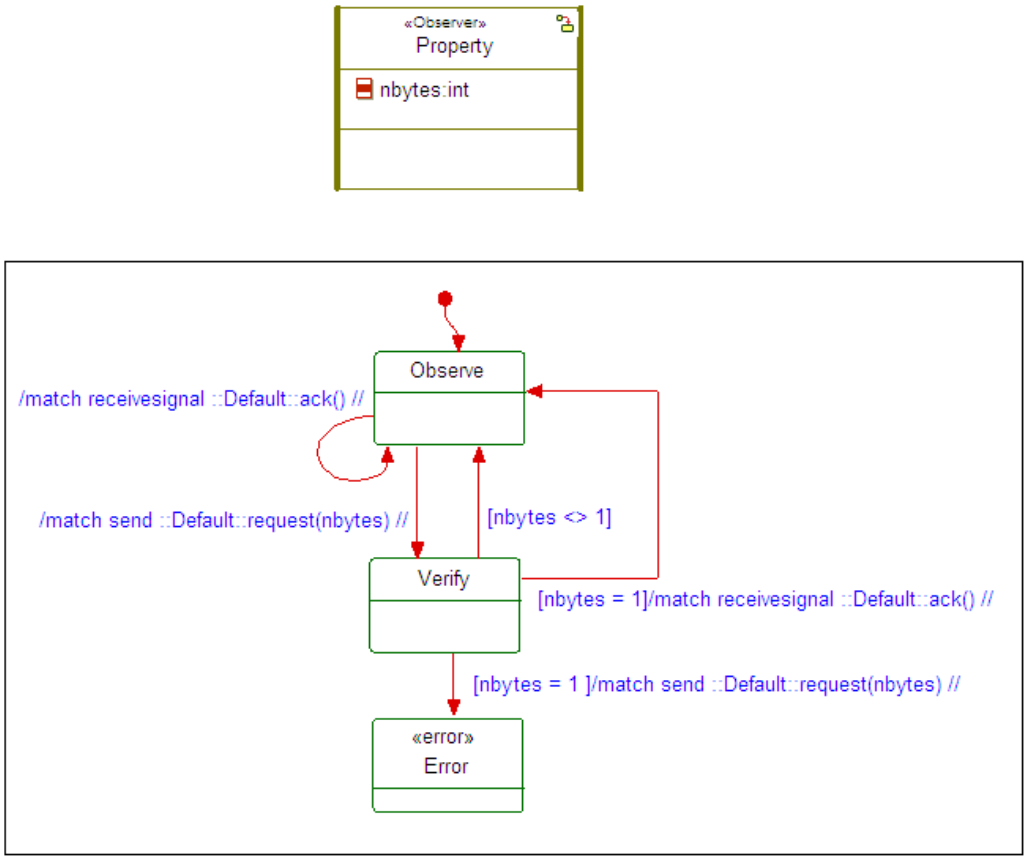


Figure 3: UML Observer for Client-Server application

The *process* is the most important component of a system. It has a unique process identifier value (*pid*) and local memory containing variables (including clocks), control states (defined by the state machine of the process) and a queue of pending messages (received and not consumed).

Processes move from one control state to another one by executing transitions. A transition can be triggered by a message in the input queue or can be spontaneous. Transitions are guarded by conditions on variables. During a transition actions expressed sequentially can be performed by the process like message sending, variable assignments, clock settings, etc. The body of a transition can be structured using control-flow statements (*if-then-else*, *while-do*, etc.) and use external functions/procedures written in another programming language (C/C++). An example of a process can be found in Figure 4.

The processes communicate between them via signals or shared variables. *Signals* can have parameters and can be addressed directly to a process (using its identifier - *pid*) or to

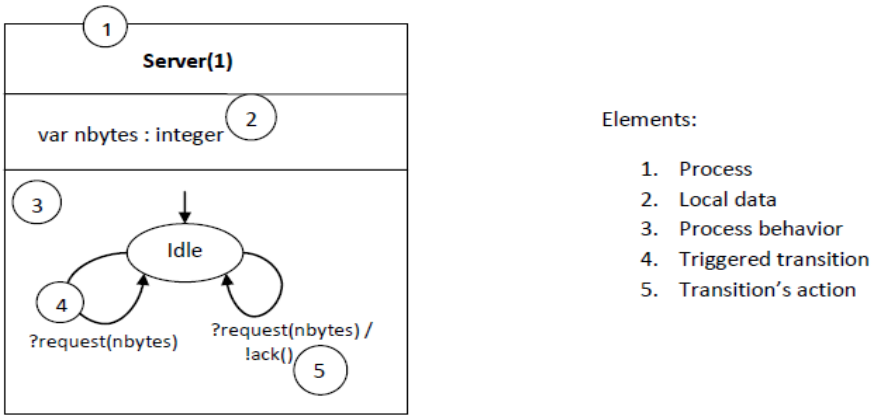


Figure 4: An IF process

a *signalroute* which delivers the message to one or more processes. The messages received by a process are stored in its queue and are consumed in a FIFO manner.

A process can define local variables which have one of the *predefined types*: **bool**, **integer**, **real**, **pid** and **clock**. Structured data can be defined by using *type constructors*: **enumeration**, **range** (domains), **record**, **array** and **abstract**. The *abstract* type makes possible the definition of more complex data, the implementation being provided into an external language. The *clock* type defines time progress. This kind of variables is usually used to express timing constraints or timed guards for transitions. The basic operations on clocks are: creation, value setting or resetting, and can be used in any transition. Variables can also be defined globally for the system.

*Observers* are mechanisms used to describe safety properties of a system. They are special processes which execute synchronously with the system and react to events occurring in the system. Special primitives for retrieving the contents of queues, values of variables, state of processes, etc. are defined in IF language. In order to express properties, the observers have states marked with *ordinary*, *error* or *success*. The last two states express the satisfaction or non satisfaction of the property described and they are both terminating states.

The IF Language implements the priority system mechanism based on dynamic priority relations. A *priority system* is a transition system with a dynamic priority relation on its actions ([10]). A *priority relation* $\prec$ is a set of predicates of the form $[condition] \Rightarrow x \prec y$ meaning that under the specified condition the action $x$ has a higher priority than the action $y$. At a given state of the transition system, only enabled actions with maximal priority can be executed so that, a priority relation restricts the behaviour of the system eliminating possible deadlocks that might appear by selecting a transition among a set of possible transitions.

For further details on the semantics of the IF Language, the reader is referred to [5].

## 1.3 Translating OMEGA UML to IF

Our interest is to have a mapping between the OMEGA Profile and the IF language such that model-checking techniques, simulation and static analysis tools developed in the IFx Toolset can be applied on real-time embedded UML models. For that, we have defined some principles which are explained into this Section.

Every UML class $X$ is mapped to a process $P_X$ with a local variable for each attribute or association of $X$. Inheritance between classes is translated by the duplication of each inherited attribute in the processes corresponding to subclasses. A special process, of type *group manager*, is defined in order to handle the activity groups. This process is responsible to treat requests coming from the environment: when the group is stable it forwards the signals or operation calls stored in its queue to the target objects inside the group. The link between processes and the activity group which owns them is made with a variable called *leader* which, for each process, points to the group manager handling its activity group.

The operations are defined by three basic signals: a call signal, a return signal and a complete signal. The *call signal* has as predefined parameters the *pid* of the process waiting for the completion of the call (the object caller or the target's activity group manager), the *pid* of the caller and the *pid* of the called object. It transports all user defined parameters of the operation. The *return signal* indicates the return from an operation execution and it is sent to the caller with some return values (if they are demanded). The *complete signal* specifies the end of the computation for an operation and it is sent to the waiting process. It differs from the return signal; the last case the operation can continue its computation after returning a value. Moreover, for a primitive operation a new process is defined having as behaviour the body of the operation. For a triggered operation, the behaviour is modelled in the state machine of $P_X$ and there has to be an explicit return action.

When an operation is invoked, a call signal is sent directly to the target, if they both are in the same activity group, or to target's activity group manager, if they are in different activity groups. The call is stored in group's queue and the group's manager forwards the call to the target object when the group is stable so that it is handled in a subsequent run-to-completion step. While the call has not been treated, the caller object waits for a return or complete signal before resuming its activities. For primitive operation $m$, since a specific process is defined $P_{X::m}$, upon reception of a call signal, the process that owns the operation, $P_X$, creates a new instance of the handler process $P_{X::m}$ and waits for it to terminate.

State machines and signals are mapped almost syntactically to IF. The actions enumerated in Section 1.1 are translated into IF as it follows: object creation is equivalent to the instantiation of the object constructor's handler process; method call is modelled by sending a call signal and waiting for a return/complete signal; return action is modelled by sending a return signal; variable assignment, signal output and control structures are directly supported in IF.

The time extensions already presented are mapped straightforward in IF. *Clocks* exist as a predefined type in IF. *Timers* are translated using a clock and a timer process sending

*timeout* signals. *Events* correspond to transitions in the IF model: the call of an operation $X :: m$ corresponds to the transition in which the signal $call_{X::m}$ is sent.

Observers are classes stereotyped with `<<Observer>>` , their translation being similar to how classes are mapped.

## 1.4 IFx Toolset

The IFx Toolset is the validation environment built upon the language and the profile presented before. It is structured in three layers:

- *front-ends*, which provide an interface with high-level languages (UML);
- *static analysis tools* for state space reduction and optimisation;
- *behavioural tools* for simulation, verification of properties, test case generator, etc.

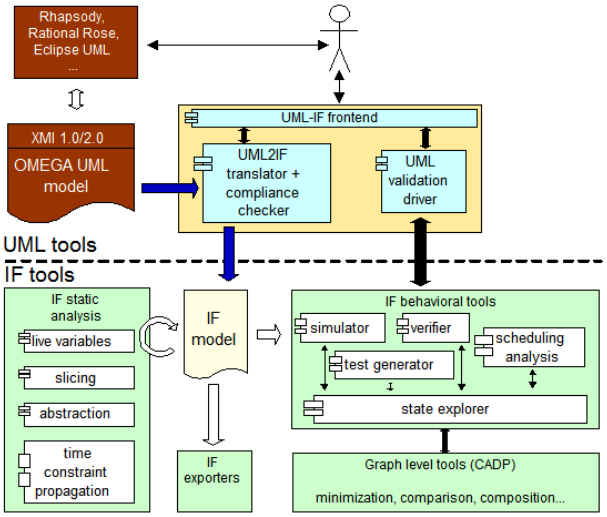


Figure 5: IFx toolbox

The workflow of an OMEGA UML model through the environment is shown in Figure 6.

The *dfa* tool (*automatic abstractions*) is based on static analysis techniques. These techniques simplify an IF description with the help of: *live variables analysis* (it removes the globally dead variables and signal parameters and resets locally dead variables), *dead-code elimination* (unreachable states and transitions are removed from the IF description under some user given assumptions) and *variable abstraction* (by eliminating variables and their dependencies which are not relevant for the user).

The *automatic verification* identifies errors in the model's behaviour (if there are errors) by exploring the state graph with the following options: *exploration order* (breadth-first or

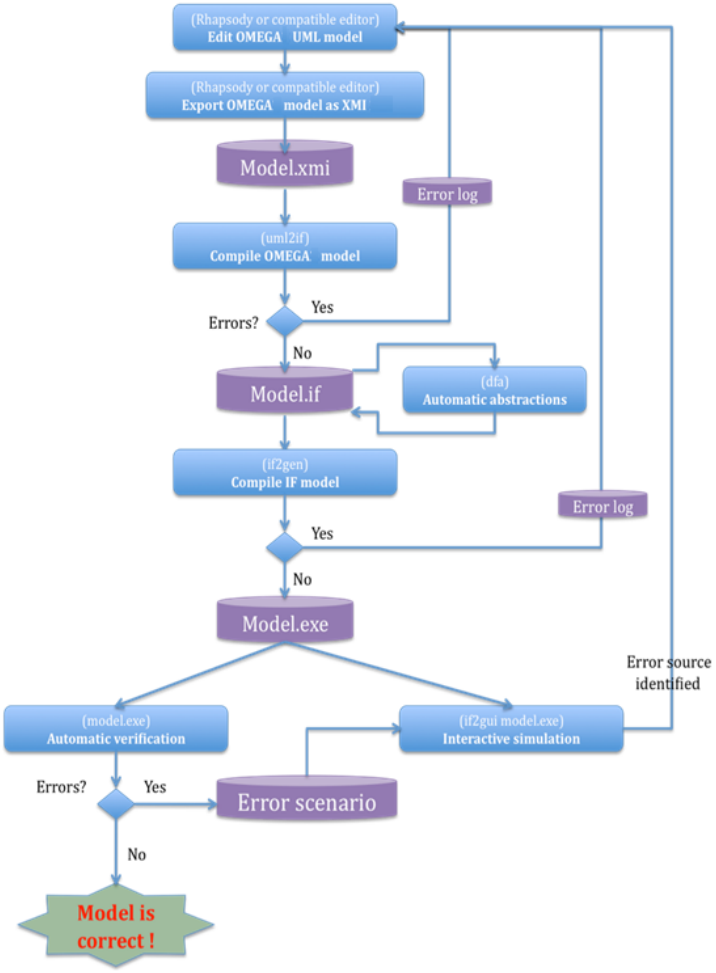


Figure 6: IFx workflow

depth-first), *error handling* (generate error scenarios, stop the state space exploration if any error has been found) and *partial order reduction* (abstraction method that during the state space search yields smaller state spaces). The interactive simulator permits to visualise model's behaviour with the possible use of error scenarios. Other behavioural techniques implemented in the toolbox are: on-the-fly model-checking based on $\mu - calculus$, model-checking with observers, model minimisation by computing an abstract model and test case generation.

The workflow of the UML model provided in Section 1.1 stops at the automatic verification step, error scenarios being generated. The observer specifies that the connection between the Client and the Server is well established but the behaviour of the Server allows ignoring all messages received from the Client for one test of the connection, forcing the last one to restart its behaviour. This makes our property to be false and the *Error* state to be reached.

# Chapter 2

# Composite structures

Composite structures have been introduced in the UML standard starting with version 2.0. They refer to "a composition of interconnected elements, representing run-time instances collaborating via communication links to some common objectives" ([14] pp. 161). Composite structures are a big evolution in modelling a system and are often used for the hierarchical representation of real-time embedded systems.

## 2.1 Background

A *composite structure* is formed by inner components, that are called *parts*, and communication paths, that are called *connectors* or *links*. *Parts* are instances of classes with predefined role and they usually are in a fix number within the composite structure. In Figure 7 parts are represented by the instances of Keypad, Display, CashUnit, CardUnit, Controller and BankTransactionBroker. *Links* connect inner components (e.g. **e** in Figure 7), an inner component with a port (e.g. **c, d**) or two ports (e.g. **f**), so that these elements can communicate between them via signals or operation calls. Such links can transport signals between elements that know how to answer to them. A link can be the realization of an *association* (especially in the case of a connector between two parts), but this is not mandatory. The UML standard classifies links in two categories: *delegation links* which connect the composite structure with one of its components (part or port of a part) and *assembly links* which connect two components between them. *Delegation links* can be separated in *outbound delegation links* and *inbound delegation links*, depending of connector's direction (if it is oriented to the outside environment or, correspondingly, to the inner structure).

A *port* (e.g. the elements **k, d, ca, cu** from ATM and Controller, **bank** from BankTransactionBroker and ATM_Bank from ATM in Figure 7) is an interaction point between its owner and the outside environment. Every port has a contract, given by classifiers (interfaces or classes), which allows it to answer to known requests by forwarding them in the needed direction. These requests can be incoming requests from the environment (the port *provides* the interface; e.g. **g** from Figure 7) or outgoing requests to the environment

(the port *requires* the interface; e.g. **h** from the same Figure).[1]

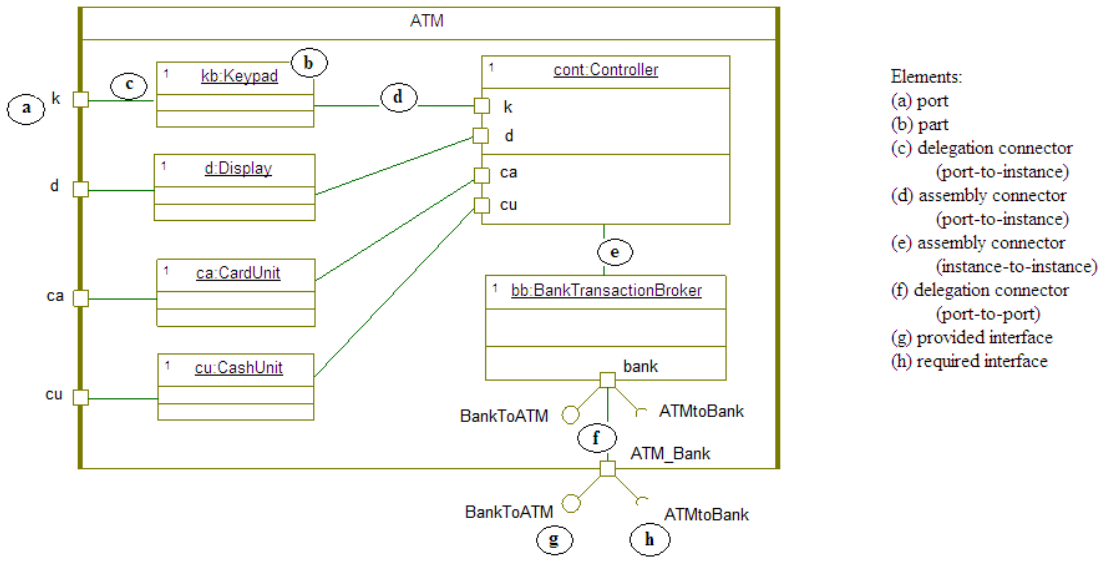


Figure 7: Composite structure example

Anyhow composite structures, as presented in UML standard, are ambiguous. For example, every connector links two entities which can be either ports or components (parts). In both cases, the two entities are typed[2]. In addition, the modeller can specify that the connector *realizes* an *association*. It is clear that, in general, connecting entities of arbitrary types does not make sense, and there should be clear compatibility rules (based on types, link direction, etc.) specifying what are the well formed structures. However, these type compatibility rules for connectors are not detailed in UML. The standard merely states that "the connectable elements attached to the ends of a connector must be compatible." and that "what makes connectable elements compatible is a semantic variation point" ([14] pp. 175-176). Various causes of ambiguity, such as the existence of several connectors starting from a same end-point, are not even mentioned.

Our purpose is to define a rule set in order to disambiguate the composite structures so that we shall have a clear a coherent executable semantics. Therefore, we had extended the OMEGA UML Profile to cover unambiguous composite structures by setting well formedness constraints and by clarifying the run-time behaviour of these structures. A formalization of the rules provided is needed for proving the type-safety of our system.

[1] *Provided* and *required* are defined from the component owning the port point of view.

[2] For a part the type is given by the class whose instance it is and for a port the type is given by its contract.

## 2.2 Extended OMEGA Profile

The extended OMEGA UML Profile (called OMEGA2) introduces an expressive and unambiguous set of constructs for modelling hierarchical structures, with an operational semantics that integrates in the existing execution model of OMEGA. The typing system and the consistency rules we have formulated can be applied to other component-based models like SysML ([12]) or MARTE ([13]).

### 2.2.1 Bidirectional ports

A first typing problem comes from the fact that in UML the ports are bidirectional, i.e. they can specify a set of allowed incoming requests (the *provided* interfaces) and a set of allowed outgoing requests (the *required* interfaces). This is represented in the model as follows: all the interfaces that are directly or indirectly *realized* by the type of the port (its contract) are considered to be *provided* interfaces. The *required* interfaces are those interfaces for which there exists a *Dependency* stereotyped with `<<Usage>>` between the port type (or one of its supertypes) and the respective interface(s). Figure 8-a shows a simple example of bidirectional port.

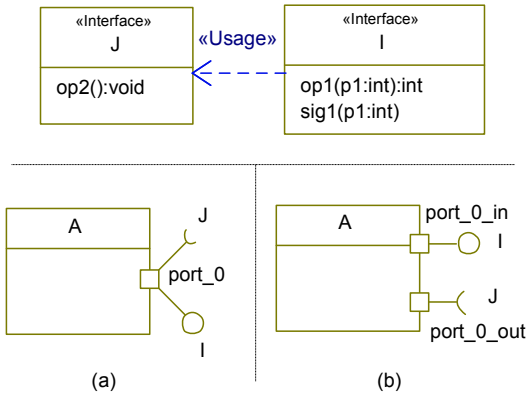


Figure 8: (a) - Example of a biriderctional port, (b) - Equivalent in OMEGA2

The type of the port `port_0` in Figure 8-a is $I$. However, the fact that the port is bidirectional raises typing problems, which are apparent in the following situations:

- When `port_0` is used by $A$ to send out requests conforming to interface $J$, by an action such as "`port_0.op2()`". In this case, `port_0` has to be treated by the type system as an entity of type $J$, although it is declared of type $I$.
- When one wants to specify behaviour of `port_0` by a state machine[3]. Then the state machine has to handle requests coming from both directions, i.e. requests conforming both to $I$ and to $J$.

[3]This is deemed possible by the UML 2.x standard [14], but without further detail.

These typing inconsistencies are not addressed by the UML standard. When we translate UML models into their IF description (or other implementation languages) they raise homologous problems for the typing of the actual object that will represent the port. A general solution, based on qualifying the types ($I$, $J$) with the corresponding directions (*in*, *out*) and on allowing the port entity to comply to multiple types, is possible but it greatly complicates the type checking of UML models.

For these reasons, the solution we adopt in OMEGA2 is to forbid bidirectional ports. This is possible because any bidirectional port can be split in two unidirectional ports, like in the example from Figure 8-b, although it can be argued that it leads to less convenient models.

Syntactically, an unidirectional outgoing port specifying a *required* interface $J$ (such as `port_0_out` from Figure 8-b) will be represented as a port typed with $J$ and stereotyped with `<<reversed>>` (to distinguished it from a port *providing* $J$).[4]

### 2.2.2 Directionality rules

A second typing problem is raised by connectors. No compatibility rules for links are given by the standard. Before presenting type compatibility issues for links, some simple directionality rules must be observed by well-formed structures:

**Rule 1** *If a* delegation *link exists between two ports, the direction* (provided *or* required) *of the ports must be the same.*

**Rule 2** *If an* assembly *link exists between two ports, one of the ports (the source) must be a «`reversed`» port (required) and the other (the destination) must be a normal port (provided).*

**Rule 3** *If a link is typed with an association, the direction of the association must be conform to the direction of the link (derived from the direction of the ports at the ends).*

Rule 1 restricts the links that can be used and we summarize here which connectors are accepted in OMEGA2:

1. Part - Part link ⇒ assembly link, needs to be typed with an association[5]
2. Port - Port link

[4]Note that a mechanism identical to the «`reversed`» stereotype is supported by the IBM Rhapsody tool [15], including support for graphical representation using the standard *required interface* symbol of UML like in Figure 8-b. For editing convenience, the Rhapsody representation is also supported by the IFx2 tools.

[5]The need of typing a link with an association is given by the fact that a component has to know to address the connector (see Rule 5 later on).

2.1 One port owned by the composite structure, the other one owned by a part
port required - port required ⇒ outbound delegation link
port provided - port required ⇒ forbidden
port required - port provided ⇒ forbidden
port provided - port provided ⇒ inbound delegation link

2.2 Both ports are owned by parts
port required - port required ⇒ forbidden
port provided - port required ⇒ assembly link
port required - port provided ⇒ assembly link
port provided - port provided ⇒ forbidden

3. Part - Port link

3.1 Port owned by a part
part - port provided ⇒ assembly link, needs to be typed with an association
part - port required ⇒ assembly link

3.2 Port owned by the composite structure
part - port provided ⇒ inbound delegation link
part - port required ⇒ outbound delegation link, needs to be typed with an association

The third rule introduces more constraints in the profile by establishing a correspondence between the direction of a connector typed with an association and the related association. These types of connectors need to be treated carefully so that by typing a link with an association, the direction in which it transports the messages does not become inconsistent and therefore the composite structure is not well-formed. This rule can be expanded in three cases:

- The association is navigable at both ends. This type of association is accepted only for a link that connects two parts and the types of each end of the link and association must be compatible.
- The association is navigable only at one end. Then the types at each end of the link and the association should be compatible. This restricts us the associations that we may have:
  - For a link between two parts, the accepted associations are associations between two classes, a class and an interface or two interfaces.
  - For a link from a part to a port, the accepted associations are between a class and an interface pointing to the interface or between two interfaces.
  - For a link from a port to a part in this direction, only the association between two interfaces is accepted.
  - For a link between two ports, only the association between two interfaces is accepted.

- The association is not navigable at both ends then the connector is not well-formed.

The end of the link is compatible with the corresponding end of the association means:

- If the end of the link is a part and the association's end is a class then the association's end has to be equal or a supertype for the link's part type.
- If the end of the link is a part and the association's end is an interface then link's end type has to realize directly or indirectly the association's end type.
- If the end of the link is a port and the association's end is an interface then the port has to provide/require the association's end.

We have mentioned as notion the *direction of a link*. We can establish the direction based on a link's type and we can define the following. A connector starts from a port providing interfaces if:

- It is an inbound delegation link between provided ports and the port is owned by the composite structure;
- It is an inbound delegation link between a provided port and a part.

A connector starts from a port requiring interfaces if:

- It is an outbound delegation link between required ports and the port is owned by the inner component;
- It is an assembly link between provided-required ports;
- It is an assembly link between a part and a required port.

A connector starts from an inner component if:

- It is an assembly link between a part and a provided port;
- It is an outbound delegation link between a part and a required port;
- It is an assembly link between parts.

Taking as a running example the Figure 9 we can express the reason behind these rules. The example is a composite $A$ with two sub-components of types $D$ and $E$, one using ports for communication ($E$) and one not ($D$). For both sub-components there are incoming links (links from port $pIJL$ of $A$) and outgoing links (links to ports $rK$ and $bak\_rA\_K$ of $A$).

So, Rule 1 forbids putting a connector, for example between $pIJL$ and $rK$, since the direction of the connector would be ambiguous. Rule 3 forces the direction of a connector to be coherent with the direction of the realized association, like in the case of the link between $d$ and $rA\_K$ (realizing association $itsK$).

These three rules allow us to have an overview on the composite structure and its behaviour. We can follow the flow of requests based on link's direction and establish which component reacts such that the main goal is achieved.

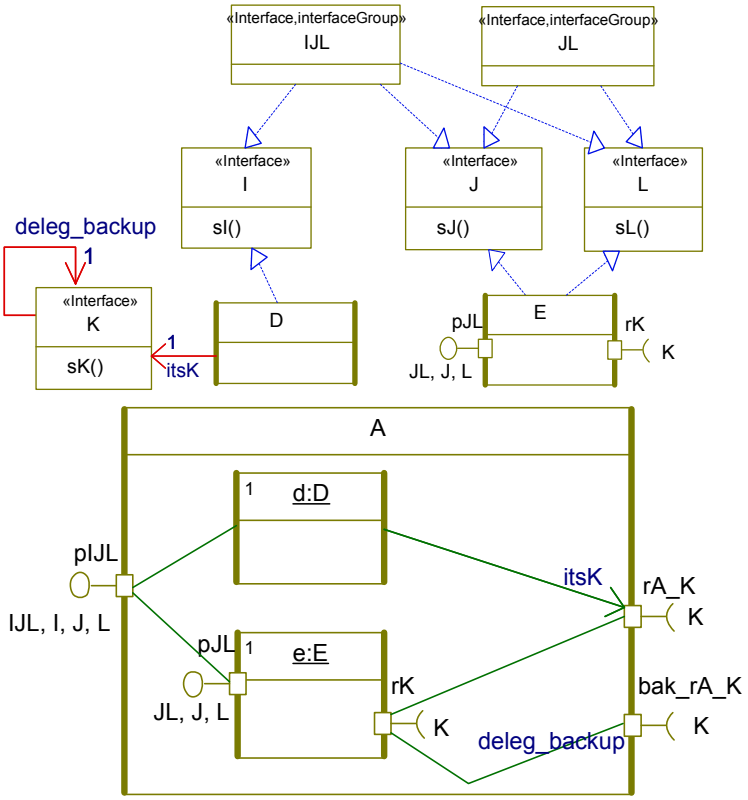


Figure 9: Connection rules in composite structures

### 2.2.3 Type coherence rules

Before presenting the type system for connectors and type-based rules, we need to introduce some notions: *interface groups*, *default delegation associations* and *set of transported interfaces*.

**Interface groups.** Let us note that it is sometimes necessary to declare several provided or required interfaces for one port (for example, $pIJL$ of $A$ which provides interfaces $I$, $J$ and $L$, see Figure 9). In UML, this is done by declaring a new interface that inherits from these interfaces and by using this new interface as the port type ($IJL$ in Figure 9). However, such interfaces are artificial syntactic additions to the model, and they should not be taken into consideration by the link compatibility rules stated in the following. In our example, $d$ and $e$ only realize interfaces $I$ and respectively $J$ and $L$, so interface $IJL$ is irrelevant for the semantics of the model. In OMEGA2, such interfaces must be stereotyped with «`interfaceGroup`» to distinguish them from meaningful ones, as shown in the upper part of Figure 9.

**Default delegation associations.** The default behaviour of a port is to forward requests from one side to the other according to its direction: to the environment, if it is a

*required* port, and to its owner, if it is a *provided* port. The minimum information needed by the port is, for each provided/required interface, which the destination should be. For example in Figure 9, port $pIJL$ needs to know (and be able to refer to) the destination of requests belonging to interface $I$ (here, $d$) and the destination of requests belonging to $J$ or $L$ (here, $pJL$ of $e$). Similarly, $rK$ needs to know that the destination of outgoing requests is by default $rA\_K$.

It follows that, for each provided/required interface, the port has to possess an association designating to which the port should forward requests belonging to that interface. In OMEGA2, every interface type $I$ has by default an association called $deleg\_I$ pointing to itself, used for this purpose (for modelling convenience, the semantics considers they exist by default if they are omitted in the model). These associations are used to define the forwarding semantics of ports, described later on.

**The dynamic type of a connector.** The type of a connector determines what type of invocations (signals or operation calls) can travel through the connector and how port behaviour descriptions refer to the connector. In general, in the case of a connector *originating*[6] from a port (i.e., not directly from a part), its type can be derived from the type of the entities situated at its two ends and does not necessarily need to be statically specified using an *association*. The following notion defines the dynamic type of the connector:

**Definition 1 [Set of transported interfaces]**. For a connector starting from a port, the *set of transported interfaces* is defined as the *intersection* between the two sets of interfaces provided/required at the two ends of the link.

As the ends of a link can be either ports or components, the meaning of provided/required interfaces is defined for each case:

- For a *Port*, the set of required/provided interfaces is the set containing the *Port*'s type and all its supertypes, without all the interfaces stereotyped as «`interfaceGroup`».
- For a component, the set of provided interfaces is the set of all interfaces directly or indirectly realized by the component's class.

According to this definition, the set of transported interfaces for the links in Figure 9 are as follows[7]:

- For link $pIJL$ to $d$ the set is $\{I\}$.
- For link $pIJL$ to $pJL$ the set is $\{J, L\}$.
- For link $rK$ to $rA\_K$ the set is $\{K\}$.

Let us note that the link from $pIJL$ to $d$ given as example above could have been statically typed with association $deleg\_I$, because the set of transported interfaces $\{I\}$ only

[6] According to link directionality, as explained in Section 2.2.2.

[7] Link $d$ to $rA\_K$ starts from the part $d$ and therefore the set of transported interfaces cannot be computed; moreover the link has to be statically typed (see Rule 5 later on).

contains one element. However, in the general case when the derived set contains several interfaces (like for example the link between $pIJL$ and $pJL$ which transports $\{J, L\}$), statically typing a link with an association is not necessary and may be restrictive.

If a static type is specified, it must be compatible with the dynamic type, as stated in the following rule:

**Rule 4** *If a link outgoing from a port is statically typed with an association, then the association is necessarily directed (cf. Rule 3) and the type pointed at by the association must belong to the* set of transported interfaces *for that link.*

On the example in Figure 9, Rule 4 implies that, for example, if the link $pIJL$ to $pJL$ is statically typed with an association then the association must point at either $J$ or $L$. But this restricts the set of requests forwarded through the link to only those requests which belong to the pointed interface ($J$ or $L$), therefore the behaviour is restricted compared to a dynamically typed link.

While the type for a connector starting *from a port* does not need to be statically specified as it can be derived as shown before, if the connector starts directly *from a component* (and not from a port) then the static type *must* be specified:

**Rule 5** *If a link originates in a component, then the link* must *be statically typed with an association, and the type of the entity at the other end of the link must be compatible with (i.e. be equal or a subtype of) the type at the other end of the association.*

In Figure 9, only the link from $d$ to $rA\_K$ is in this case; the link has indeed to be typed (here, with $itsK$) or otherwise the component would have no means to refer to it for communication.

Finally, a link is meaningful only if it can transport some requests:

**Rule 6** *The set of transported interfaces for each link should not be void.*

The above rules allow us to specify exactly what requests (signals and operation calls) can travel through connectors by defining compatible interfaces for each component.

### 2.2.4 Port behaviour rules

In OMEGA2, the default behaviour of a port is to forward requests from one side to the other, depending on the port's direction. Each request (signal or operation call) will be forwarded to a destination which depends on the interface to which the signal or operation belongs, using the default *deleg* associations above described. For example, the default forwarding behaviour of port $pIJL$ from Figure 9 can be described by the state machine in Figure 10-a[8].

[8] `deleg_I!sI` is the OMEGA2 syntax for the action of sending signal `sI` to the destination `deleg_I` (if the signal has formal parameters and no actual parameters are specified in the sending action, the actual values that will be sent are those ones received at the last *reception* – here the one that triggered the transition).

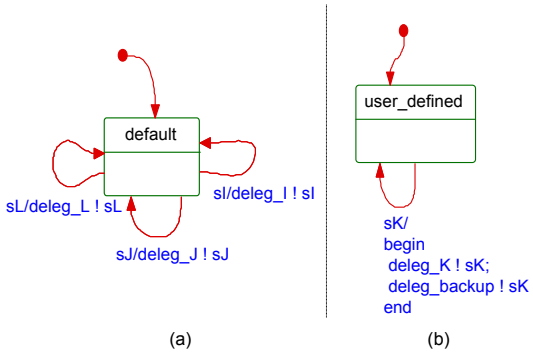


Figure 10: (a) - Default state machine for port $pIJL$, (b) - User-defined machine for port $rK$

The default behaviour is unambiguous only if for any interface, the entity to which the corresponding *deleg* association points at is clear. Therefore, the following rules are necessary:

**Rule 7** *If several* non-typed *connectors start from one port, then the sets of interfaces transported by each of the connectors have to be pairwise disjoint.*

The last rule does not forbid the case where a port is connected to $n$ entities that provide or require the same interface $I$ ($n > 1$): it states that in this case at least $n-1$ connectors have to be explicitly typed with associations. The one connector which is not explicitly typed, if it exists, is implicitly typed with *deleg_I*. In the example from Figure 9, port $rK$ of $e$ is in this situation: it has two links to two ports ($rA\_K$ and $bak\_rA\_K$), both typed with the same interface ($K$). According to Rule 7, one of the links has to be explicitly typed; here, the second one is statically typed with the association *deleg_backup*.

The default port behaviour may be redefined by attaching a state machine to the port's type. In OMEGA2, this state machine may use the implicitly typed connectors ( accessed via the default *deleg* associations), as well as the explicitly typed connectors (via their defining association). In Figure 10-b we show an example of port behaviour for port $rK$ (from Figure 9), which duplicates every $sK$ signal on both the default connector (*deleg_K*, communicating with $rA\_K$) and the secondary connector (*deleg_backup*, communicating with $bak\_rA\_K$).

In addition, for completeness of the port behaviour, we require the following:

**Rule 8** *The union of the sets of interfaces transported by each of the connectors* originating *from a port $P$ must be equal to the set of interfaces provided/required by $P$.*

Applied for example to port $pIJL$ from Figure 9, this rule says that the two links originating from the port must transport, together, the entire set of interfaces provided by the port, i.e. $\{I, J, L\}$ (remember that $IJL$ is an «`interfaceGroup`» and does not count in type checks).

### 2.2.5 Concurrency model and observers

The concurrency model is left open in UML. The previous version of OMEGA defined a particular concurrency model, based on the standard UML notion of *active* and *passive* classes. Due to the choice of partitioning the object space in activity groups attached to active objects, certain forms of simple resource sharing and synchronisation generated quite complex models, as sharing could only be achieved via an explicitly modelled resource manager – an active object. In order to overcome this problem, a new kind of passive class can be defined in OMEGA2 (using the stereotype «`protected`»).

*Protected objects* are passive objects that do not belong to one activity group but rather are shared between the groups. They work in the same way as Ada protected objects [3]. Like in Ada, protected objects are a synchronization mechanism. They provide *functions* (which may only read but not modify object attributes) that can be executed concurrently, and *entries* that are executed in mutual exclusion from each other and from functions (this corresponds to the classical readers-writers pattern). In addition, an entry has a guard; a call to an entry from a thread (activity group) will wait before beginning the execution until the guard is true. Our model of protected objects is slightly simplified (more non-deterministic) compared to the *eggshell* model of Ada [3] and therefore suppresses the need for *procedures* existing in Ada: a procedure can be seen as an entry with guard `true`.

The OMEGA2 concurrency model therefore distinguishes three kinds of classes: active, passive and protected. Since every passive object is considered to *belong* to an active object, in the sense that its behaviour is executed on the execution thread of its owner, some rules are necessary to avoid confusing configurations in composite structures.

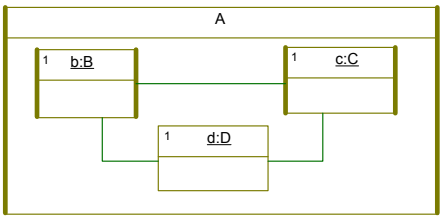


Figure 11: Forbidden composite structure.

For example, the composite structure in Figure 11 shows, on the same level, two active objects ($b$ and $c$), which have their own *activity group*, and a passive object ($d$) which belongs to the group of its creator (instance of $A$). This kind of structure is forbidden. If the desired semantics is to have a shared passive object $d$, then $d$ may be declared as «`protected`» and the structure becomes valid.

**Rule 9** *A passive class may define a composite structure formed only of passive classes.*

**Rule 10** *An active class may define a composite structure formed of either only passive classes, or of a combination of active and protected classes.*

As an extension to the original profile, an observer can also define a simple composite structure. Composite observers have proved to be a way for making more compact the specification of some complex verification properties. The well-formedness rule is:

**Rule 11** *If an Observer defines a composite structure, the components must also be instances of Observers.*

## 2.3 Translating composite structures to IF

As with the previous version of the OMEGA profile, the one presented above is supported by a toolset (IFx2) offering simulation and verification functionality. The tool handles UML models in XMI 2.0, edited with any compatible UML editor such as IBM Rhapsody, and translates them into IF description corresponding to the principles presented below.

The challenge in the definition of the translation is to fit the relatively complex, hierarchical UML modelling constructs into a simple and flat object space, IF automata, while preserving traceability of the original constructs and without worsening the state space explosion problem.

Our translation is based on the principle that the modelling elements involved in composite structures, namely ports and connectors, should be handled as first class language citizens. This means that we refrain from flattening the model during compilation and hard-wiring all the communication paths (something that is done, for example, in certain SDL compilers). Concretely, each port instance is implemented as an IF process instance (whose behaviour corresponds to the routing behaviour described in Section 2.2.4) and each connector is represented by attributes in the end-points (in ports or in components), corresponding to the association defining the connector (the default *deleg* association or the explicitly specified one).

In this setting, a UML composite structure diagram is used simply as an initialization scheme for instantiating components and ports and for creating links. A composite structure is therefore translated to a constructor.

As a consequence of the translation sketched above, a signal or operation call sent through a connector chain will pass through several objects (the intermediate ports) before reaching the destination. In order to avoid the state space explosion problem due to the interleaving of such "forwarding" actions, the translator defines a *total priority order* between these actions. Thus, even if several signals are in transit on connector chains, only one forwarding action (belonging to the enabled port with the highest priority) will be enabled at any given time. This yields an increase of state space due to connector chains which is linear in the length of the chain, instead of combinatorial explosion. Note that starvation of lower priority actions is not possible since, in any state, eventually all signals that are in transit through connectors will arrive at destination and the rest of the system will be able to make progress. Moreover, this abstraction is made without any loss of generality, since all the possible interleavings at the level of component transitions (which is the observable level) remain feasible. The implementation of the abstraction is made very easy by the dynamic priority mechanism of IF .

Another element that is added in the second version of the OMEGA profile is *protected* classes. Compared to normal passive classes, protected classes add the classical *readers-writers* synchronization protocol for functions and entries. The readers-writers protocol implemented in our translation is a variant of the classical solution that may be found in many textbooks (e.g., [6]). The implementation is however facilitated by the fact that the IF language offers mutually exclusive and atomic *transitions* by default, and transitions can specify conditional waiting simply using guard conditions.

For structured observers, the same mapping as for composite structures is applied.

# Chapter 3

# OCL Formalization

The rules disambiguating composite structures also implemented in the OMEGA2 compiler are formalized in OCL to verify that UML models comply with our profile. The OCL code was developed in Topcased OCL Environment [2].

For our formalisation we have defined helper functions for accessing:

- type of elements connected with a link: *has2Ports*, *has2Parts*, *has1PartAnd1Port*, *has1PartWithPort*, *has2PartsWithPort*;
- the connected elements: *part1*, *part2*, *port1*, *port2*;
- association's properties: *isTyped*, *isBidirectional*, *isNotNavigable*, *isEnd1Navigable*, *associationEnds*, *isClassClassAssociation*, *isInterfaceInterfaceAssociation*, *associationStartPoint*, *associationStartPointType*, *associationEndPoint*, *associationEndPointType*, *isClassInterfaceAssociation*;
- port's type: *isReversed*;
- types of classifiers: *isInterfaceGroup*, *isInterface*, *isProtected*, *isObserver*.

The definition of these functions, together with the invariants presented in this Section, can be found in the Appendix.

For the formalization of Rule 1 and Rule 2 we define a function that will compute the exact link type based on the classification presented in Section 2.2.2. The OCL invariant corresponding to these two rules becomes the verification for each connector in the model if its type is not *forbidden*:

```
context Connector

-- Definition of link's type
def: linkType : String =
  if has2Parts
    then 'assembly link between parts'
  else
```

```
    if has2Ports
      then if has1PartWithPort
             then if not port1.isReversed and not port2.isReversed
                    then 'inbound delegation link between provided ports'
                  else if port1.isReversed and port2.isReversed
                         then 'outbound delegation between required ports
                             '
                       else 'forbidden'
                       endif
                  endif
           else if (port1.isReversed and port2.isReversed) or
                   (not port1.isReversed and not port2.isReversed)
                  then 'forbidden'
                else  'assembly between provided-required ports'
                endif
           endif
    else
      if has1PartWithPort
        then if not port1.isReversed
               then 'assembly link between part and provided port'
             else 'assembly link between part and required port'
             endif
      else if not port1.isReversed
             then 'inbound delegation link between part and provided port
                 '
           else 'outbound delegation link between part and required port'
           endif
      endif
    endif
  endif

-- Rule 1 and Rule 2
inv LinkType: self.linkType <> 'forbidden'
```

For the formalization of the third rule we need to verify the compatibility between the association end (of the association typing the link) and the corresponding link end (i.e. the compatibility has to be verified between the start point for both link and association and for the end point). As explained in Section 2.2.2, this resumes to verify the inclusion of the association's end type in link's end type (realized interfaces or superclasses). We define functions that verify if a link starts from a port or a part as already presented (*isStartingFromProvidedPort*, *isStartingFromRequiredPort*, *isStartingFromPort*, *isStartingFromPart*), and for each link which is its starting point and its ending point (*linkStartPort*, *linkEndPort*, *linkStartPart*, *linkEndPart*). Since the formalization of the following rules consists in computing the provided/required interfaces for a port and a component and since the same calculus can be used for Rule 3, we shall continue by computing the needed sets.

We continue with the calculus of the provided/required interfaces for a port and the interfaces provided by a component. In the case of a port, the set of realized interfaces is given

by the set of provided interfaces without those stereotyped with <<interfaceGroup>>.[1] Please note that interfaces stereotyped <<interfaceGroup>> are artificially added to the model and they should not be taken into consideration in our formalization.

```
context Port

-- Definition of interfaces realized by a port
def: interfaces : Set(Classifier) = self.provided->reject(
    isInterfaceGroup)
```

Before computing the set of realized interfaces by a class (the type of a part), we have to compute recursively the list of parents. We suppose that our model is well-formed and has no cycles.

```
context Classifier

-- Definition of classifier's parents recursive computation
def: getParentsRec : Set(Classifier) =  self.general->union(self.general
    ->iterate(p:Classifier; res:Set(Classifier)=Set{}| res->union(p.
    getParentsRec)))
```

For a class, the set of provided interfaces is the set of realized interfaces summed with the set of parents for each realized interface and summed with the set of provided interfaces for each parent of our class, without those stereotyped with <<interfaceGroup>>.

```
context Class

-- Definition of all interfaces directly realized by a class
def: iRealizations : Set(Classifier) = self.interfaceRealization.contract
    ->asSet()

-- Definition of interfaces provided by a class directly or indirectly
    realized (used in the case of a link not typed by an association)
def: interfaces : Set(Classifier) =
  iRealizations->union(iRealizations->iterate(i:Interface; res:Set(
      Classifier)=Set{}| res->union(i.getParentsRec)))
  ->union(self.getParentsRec->iterate(c:Class; res:Set(Classifier)=Set{}|
       res->union(c.interfaces)))
  ->reject(isInterfaceGroup)
```

In the case of a link typed with an association which has as an end an interface, the set of provided interfaces is the set of the interface to which it points summed with its parents and without those interfaces stereotyped with <<interfaceGroup>>.

[1] Required interfaces are modelled with reversed ports and are therefore also accessed using provided.

```
context Interface

-- Definition of interfaces provided by an interface (used in the case of
     a link typed by an association pointing to an interface)
def: interfaces : Set(Classifier) = self.oclAsType(uml::Classifier)->
    asSet()->union(self.getParentsRec)->select(not isInterfaceGroup and
    isInterface)
```

Because of the mishandling of polymorphic functions in OCL, we need to define explicitly the polymorphism of the function *interfaces* on the subtypes of *Type* (*Class* and *Interface*).

```
context Type

-- Determines the set of provided interfaces by a class or an interface
def: interfaces : Set(Classifier) =
  if self.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces
  else
    self.oclAsType(uml::Class).interfaces
  endif
```

We compute the set of transported interfaces as the intersection of provided interfaces of both ends and we formalize Rule 6: the cardinal of the set of transported interfaces should be at least equal to one. We need to remark that this set is computed for links starting from a port (typed or not typed with an association) and it is not computed in the case of a part-part connector.

```
context Connector

-- Definition of the set of transported interfaces
def: setTransportedInterfaces : Set(Classifier) =
  if has2Parts
    then Set{OclInvalid}
  else if has2Ports
         then if isTyped
                then if isStartingFromPort(port1)
                       then (port1.interfaces) -> intersection(
                          associationEndPointType.interfaces)
                     else (port2.interfaces) -> intersection(
                        associationEndPointType.interfaces)
                     endif
              else (port1.interfaces) -> intersection(port2.interfaces)
              endif
       else
         if isTyped
```

```
            then (port1.interfaces) -> intersection(
               associationEndPointType.interfaces)
          else (port1.interfaces) -> intersection(part1.type.interfaces)
          endif
        endif
  endif

-- Rule 6
inv SetOfTransportedInterfacesNonEmpty: self.setTransportedInterfaces->
    size()<>0
```

In order to formalize Rule 4 and Rule 5, we define the compatibility between two classes and between a class and an interface as the inclusion of association's end type in the set of provided interfaces or in the set of parents. It is followed by the compatibility between a port and an interface as the inclusion of all realized interfaces by the association's end type in the set of provided/required interfaces of the port.

```
context Classifier

-- Verifies if the current classifier is compatible with the one given as
    parameter
def: isCompatible(c:Classifier) : Boolean =
  if self.oclIsKindOf(uml::Interface) and c.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces->includes(c)
  else
    if self.oclIsKindOf(uml::Class) and c.oclIsKindOf(uml::Interface)
      then self.oclAsType(uml::Class).interfaces->includes(c)
    else
     (c->asSet()->union(c.getParentsRec))->includes(self)
    endif
  endif

context Type

-- Verifies if the link end's type (the type of a part) is compatible
    with the association end's type given as parameter
def: isCompatible(t:Type) : Boolean = self.oclAsType(uml::Classifier).
    isCompatible(t.oclAsType(uml::Classifier))

context Port

-- Verifies if the current port is compatible with the association end's
    type given as parameter
def: isCompatible(t:Type) : Boolean = self.interfaces->includesAll(t.
    oclAsType(uml::Interface).interfaces)
```

Rule 4 states that for a link starting from a port and typed with an association, the association must be directed (unidirectional) and the interface pointed by the association

has to be included in the set of transported interfaces. We include here Rule 3, which adds that the direction of the link has to be conforming to the direction of the association, as defined in Section 2.2.2. We define a function (*linkStartingFromPortVerification*) that verifies if for a link typed with an interface-interface association (the only association accepted for a connector starting from a port) the corresponding start points and end points are compatible and that the interface to which it points is included in the set of transported interfaces.

```
context Connector

-- Verifies if a link starting from a port and typed with an association
   has the same direction with the association and the interface pointed
   is included in the set of transported interfaces
def: linkStartingFromPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isInterfaceInterfaceAssociation
        then if has1PartAnd1Port
              then linkStartPort.isCompatible(associationStartPointType)
                  and linkEndPart.type.isCompatible(
                  associationEndPointType) and
                    setTransportedInterfaces->includes(
                        associationEndPointType.oclAsType(uml::Classifier))
             else linkStartPort.isCompatible(associationStartPointType)
                 and linkEndPort.isCompatible(associationEndPointType) and
                  setTransportedInterfaces->includes(
                     associationEndPointType.oclAsType(uml::Classifier))
             endif
    else false
    endif
  endif
```

Then the OCL invariant corresponding to these two rules verifies that for each link in the model starting from a port and typed with an association the compatibility stated above is verified.

```
context Connector

-- Verifies if a link starting from a port is well-formed
def: linkStartingFromPort : Boolean =
  if (not isStartingFromPart) and isTyped
    then linkStartingFromPortVerification
  else true
  endif

-- Rule 3 and Rule 4
inv LinkStartingFromPort: self.linkStartingFromPort
```

Rule 5 completes Rule 3, by adding that all connectors starting from a part have to be typed with an association and if the association is bidirectional (the only bidirectional association accepted is the association between two classes that may type only the link that connects two parts) it has to be compatible with the link in a direction. For a unidirectional association that types the link, we need to have the compatibility between the corresponding ends (link's start part with association's start point and link's end part/port with association's end point). This is expressed by the below functions (*linkPartPartVerification*, *linkPartPortVerification*), which make the difference between a link that connects two parts (which accepts all kinds of associations) and the link that connects a part with a port (which accepts only the association between two interfaces or between a class and an interface).

```
context Connector

-- For a link between two parts verifies if the ends are compatible with
   the corresponding ends of the accepted association
def: linkPartPartVerification : Boolean =
  if isNotNavigable
    then false
  else
    if isBidirectional
      then (linkStartPart.type.isCompatible(associationStartPointType)
          and linkEndPart.type.isCompatible(associationEndPointType)) or
               (linkStartPart.type.isCompatible(associationEndPointType)
                  and linkEndPart.type.isCompatible(
                  associationStartPointType))
    else (linkStartPart.type.isCompatible(associationStartPointType) and
       linkEndPart.type.isCompatible(associationEndPointType))
    endif
  endif

-- For a link between a part and a port verifies if the ends are
   compatible with the corresponding ends of accepted association
def: linkPartPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isClassClassAssociation
      then false
    else
      if isInterfaceInterfaceAssociation
        then (linkStartPart.type.isCompatible(associationStartPointType)
           and linkEndPort.isCompatible(associationEndPointType))
      else
        if isClassInterfaceAssociation
          then linkStartPart.type.isCompatible(associationStartPointType)
              and linkEndPort.isCompatible(associationEndPointType)
        else false
        endif
```

```
      endif
    endif
  endif
```

The invariant for Rule 3 and Rule 5 verifies that each connector in the model starting from a part is typed with an association and the direction of the association is compatible with the direction of the link:

```
context Connector

-- Verifies if a link starting from part is typed with an association and
    if it is well-formed
def: linkStartingFromPart : Boolean =
  if isStartingFromPart
    then if has2Parts
           then isTyped and linkPartPartVerification
         else isTyped and linkPartPortVerification
         endif
  else true
  endif

-- Rule 3 and Rule 5
inv LinkStartingFromPart: self.linkStartingFromPart
```

We formalize now the rules for port behaviour. The default behaviour is that the port forwards the requests received according to its direction: to the environment if it is a *required port* and to the component that owns it if it is a *provided port*. This means that the port knows how to respond to any received request and also which is the destination of the request.

The context in our formalization becomes the *Port* and we define functions that give all the connectors (typed or not with an association) starting from the port (*connectors*, *connectorsNotTyped*, *connectorsStartingFromPort*) and that verify if the port has connectors (typed or not with an association) starting from it (*hasConnectors*, *hasTypedConnectors*, *isStartingPort*).

The relation behind the first rule concerning port's behaviour states that the sets $A_1, A_2, ..., A_n$ are *pairwise disjoint* if and only if $\text{card}(A_1 \cup A_2 \cup ... \cup A_n) = \text{card}(A_1) + \text{card}(A_2) + ... + \text{card}(A_n)$. The function *unionSetForTransportedInterfacesOnLinks* computes the left hand side of the equality, and the function *noOfTransportedInterfacesOnLinks* computes the right hand side of the expression.

```
context Port

-- Determines the union of the sets of transported interfaces on each
    link starting from the port
def: unionSetForTransportedInterfacesOnLinks(withType:Boolean) : Set(
    Classifier) =
```

```
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; s:Set(
     Classifier)=Set{} | s->union(c.setTransportedInterfaces))

-- Determines the sum of the number of transported interfaces on each
   link starting from the port
def: noOfTransportedInterfacesOnLinks(withType:Boolean) : Integer =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; i:
     Integer=0 | i + (c.setTransportedInterfaces->size()))
```

Then Rule 7 becomes the verification of the equality stated above:

```
context Port

-- Definition of pairwise disjoint sets of transported interfaces
def: isPairwiseDisjoint : Boolean =
  if isStartingPort and hasConnectorsNotTyped
    then  if unionSetForTransportedInterfacesOnLinks(false)->size() <>
       noOfTransportedInterfacesOnLinks(false)
            then false
         else true
         endif
  else true
  endif

-- Rule 7
inv PairwiseDisjoint: self.isPairwiseDisjoint
```

For Rule 8 we will use the union of the sets of transported interfaces computed above and we will test its equality with the set of provided/required interfaces by the port.

```
context Port

-- Verifies if the union of sets of transported interfaces is equal to
   the interfaces provided/required
def: isComplete : Boolean =
  if isStartingPort
    then unionSetForTransportedInterfacesOnLinks(true) = self.interfaces
  else true
  endif

-- Rule 8
inv Completeness: self.isComplete
```

For the two rules concerning the execution model for composite structures we will reason on the number of active, passive and protected components given by the functions *noOfComponents*, *noOfActiveComponents*, *noOfPassiveComponents*, *noOfProtectedComponents* and *isComposite*.

Rule 9 says that if a composite structure is passive then it is well-formed if and only if the number of passive parts this owns is equal with the total number of parts. Rule 10 says that if a composite structure is active then it is well-formed if and only if or the number of passive parts is equal to the total number of parts or the sum between the number of active parts and the number of protected parts is equal to the total number of parts.

```
context Class

-- Definition of a well-formed class
def: isWellFormed : Boolean =
  if self.isActive and isComposite
    then  if noOfActiveComponents + noOfProtectedComponents =
       noOfComponents or
            noOfPassiveComponents = noOfComponents
           then true
         else false
         endif
  else
    if (not self.isActive) and (not isProtected) and isComposite
      then if noOfPassiveComponents = noOfComponents
             then true
           else false
           endif
    else OclInvalid
    endif
  endif

-- Rule 9 and Rule 10
inv CompositeStructure: self.isWellFormed <> false
```

The last rule regarding the simple composite observers is also formalized with the means of the number of observer parts. This rule is equivalent to: the number of parts of a composite observer is equal to the number of observer parts (*noOfObservers*) of the same composite structure.

```
context Class

-- Definition of a well formed observer
def: isObserverWellFormed : Boolean =
  if self.isObserver and isComposite
    then noOfComponents = noOfObservers
  else true
  endif

-- Rule 11
inv CompositeObserver: self.isObserverWellFormed <> false
```

# Chapter 4

# Isabelle/HOL Formalization

Isabelle/HOL is a theorem proving system which can be used for the specification and verification of systems. It is based on interactive generic proof assistant Isabelle, extended with High-Order Logic theory. Describing a formalism in Isabelle/HOL means creating a theory: types, functions and theorems (with their proof).

[22] presents the formalization and verification of a Java subset, called $\text{Java}_{light}$, formed of classes (with attributes and methods), interfaces and relations between them. The same principles as explained in this work were applied in the formalization of the OMEGA2 Profile which uses the same notions for modelling composite structures.

We begin the formalization by specifying an abstract syntax of our profile as Isabelle datatypes. Since we are interested in the formalization of composite structures, we will provide an abstract syntax for the structural part of our profile. The approach is to describe the profile in a bottom-up manner (starting with stereotypes and ending with the definition of the model).

The stereotypes defined in OMEGA2 which concern our formalisation are those for the execution model of a class (*active*, *passive* and *protected*) and for the difference between native interfaces and syntactically added ones (*interfaceGroup*).

$$\begin{array}{ll} \textit{clssConcurrency} & ::= \textbf{active} \mid \textbf{passive} \mid \textbf{protected} \\ \textit{intfType} & ::= \textbf{interfaceGroup} \mid \textbf{none} \end{array}$$

Variables can have a *predefined type* or a *reference type*. The *predefined types* from OMEGA2 are entirely covered here. The *reference type* can be a *Class* or an *Interface*, since the only associations accepted are between classes and interfaces. The opaque type *tname* refers to the name of the newly defined types in the UML model.

$$\begin{array}{ll} \textit{dt} & ::= \textbf{PrimDT}\ \textit{primDT} \mid \textbf{RefDT}\ \textit{refDT} \\ \textit{primDT} & ::= \textbf{boolean} \mid \textbf{integer} \mid \textbf{real} \mid \textbf{Timer} \\ \textit{refDT} & ::= \textbf{NullT} \mid \textbf{IfaceT}\ \textit{tname} \mid \textbf{ClassT}\ \textit{tname} \end{array}$$

A *field declaration* is formed by its name (the opaque type *ename*) and its datatype (predefined or reference).

$$fdecl \quad ::= ename \times dt$$

An *interface* (*intfDecl*) contains its name, its stereotype (if it is an `<<interfaceGroup>>` or not), the list of superinterfaces names and the list of realized interfaces names.

$$\begin{array}{ll} intfDecl & ::= tname \times intfTb \\ intfTb & ::= intfType \times (tname)list \times (tname)list \end{array}$$

An *association* (*assocDecl*) contains its name, the type name of one end and its navigability and the type name of the other end and its navigability.

$$\begin{array}{ll} assocDecl & ::= ename \times assocTb \\ assocTb & ::= tname \times \textbf{boolean} \times tname \times \textbf{boolean} \end{array}$$

Since our language for describing the UML model is textual, we need to define a datatype (*portDirection*) to express if the port is *providing* or *requiring* interfaces. A *port* (*portDecl*) contains its name, direction, contract and owner.

$$\begin{array}{ll} portDecl & ::= tname \times portTb \\ portTb & ::= portDirection \times tname \times tname \\ portDirection & ::= \textbf{provided} \mid \textbf{required} \end{array}$$

A *connector* (*connDecl*) contains its name, its two ends, the owner, a boolean determining if the link is typed with an association or not and if it is typed the association which types the link.

$$\begin{array}{ll} connDecl & ::= ename \times connTb \\ connTb & ::= tname \times tname \times tname \times bool \times (assocDecl\ option) \end{array}$$

The *class* (*clssDecl*) is defined by its name, its concurrency, the list of superclasses names, the list of realized interfaces names, the list of local variables, the list of ports and the list of connectors that it owns.

$$\begin{array}{ll} clssDecl & ::= tname \times clssTb \\ clssTb & ::= csig \times cbodyP \\ cbodyP & ::= (fdecl)list \times (portDecl)list \times (connDecl)list \\ csig & ::= clssConcurrency \times (tname)list \times (tname)list \end{array}$$

Finally, the *model* (*model*) is defined by classes, interfaces and ports.

$$model \quad ::= (clssDecl)list \times (intfDecl)list \times (portDecl)list$$

The opaque datatype *tname* introduced for the representation of class and interface names is too wide. In order that our abstract syntax comply to OMEGA2, we have to add rules for well-formedness of elements:

- The ends of a connector can be classes and / or ports; the owner has to be a class and all these elements have to be defined in the model:

```
wf_Link :: "model ⇒ connDecl ⇒ bool"
"wf_Link M Conn ≡ ( let x=( fst( snd Conn)); y=( fst( snd( snd Conn)));
owner=( fst( snd( snd( snd Conn)))) in ((is_class M x ∧ (is_class M y ∨
is_port M y)) ∨ (is_port M x ∧ (is_class M y ∨ is_port M y))) ∧
(is_class M owner))"
```

- The ends of an association can be classes and / or interfaces that are already defined in the model:

```
wf_Association :: "model ⇒ assocDecl ⇒ bool"
"wf_Association M Assc ≡ ( let x=( fst( snd Assc)); y=( snd( snd Assc)) in
(is_class M x ∧ (is_class M y ∨ is_iface M y)) ∨
(is_iface M x ∧ (is_class M y ∨ is_iface M y)))"
```

- The type of a field must exist in the model; if it is a reference type it can be only a class:

```
wf_Field :: "model ⇒ fdecl ⇒ bool"
"wf_Field M Fld ≡ is_primitiveDT ( snd Fld) ∨ (is_defined M ( snd Fld) ∧
is_class M (getReferenceDT_Name ( snd Fld)))"
```

- The contract of a port can be an interface or a class defined in the model:

```
wf_Port :: "model ⇒ portDecl ⇒ bool"
"wf_Port M Prt ≡ ( let Contr= fst( snd( snd Prt)); owner= snd( snd( snd Prt)) in
(is_iface M Contr ∨ is_class M Contr) ∧ (is_class M owner))"
```

The functions *is_class*, *is_iface* and *is_port* are searching in the model for the definition of a class, interface or port with the given name as parameter. The method used is the lookup table, as described in [17]. As in the OCL formalisation, helper functions are defined for qualifying certain types of elements: *is_provided*, *is_required*, *is_active*, *is_passive*, *is_protected* and *is_ifaceGroup*. The entire definition of these functions can be found in the Appendix.

For the formalization of first two rules regarding connector's type, we need to define a custom datatype with all the accepted types of connectors in OMEGA2.

```
datatype linkTP = assembly_parts
                | assembly_provided_required_ports
                | assembly_part_provided_port
                | assembly_part_required_port
                | inbound_delegation_provided_ports
                | inbound_delegation_part_provided_port
                | outbound_delegation_required_ports
                | outbound_delegation_part_required_port
                | forbidden
```

A function that computes the type for a link is defined (*linkType* - similar to the OCL definition) and the formalization of first two rules resumes to the following relation[1]:

$$wf\_R1R2(M) \ = \ \forall x \in \{cls \mid is\_class(M, cls)\}, \ \forall y \in \{conn \mid conn \ is \ connector \ of \ x \ \}, \ linkType(M, y) \neq forbidden$$

Rule 10 states that an active class is formed only of active and protected parts or only of passive parts. This is formalized by the following logical relation:

$$wf\_activeCls(M) \ = \ \forall x \in \{cls \mid is\_class(M, cls) \wedge \ is\_active(cls)\}, \ (\forall y \in \{fld \mid fld \in get\_parts(x) \ \}, \ is\_active(y) \vee is\_protected(y)) \vee (\forall y \in \{fld \mid fld \in get\_parts(x) \ \}, \ is\_passive(y))$$

The relation below is the formalization for Rule 9 (a passive class is formed only of passive parts):

$$wf\_passiveCls(M) \ = \ \forall x \in \{cls \mid is\_class(M, cls) \wedge \ is\_passive(cls) \ \}, \ \forall y \in \{fld \mid fld \in get\_parts(x) \ \}, \ is\_passive(y)$$

We can summarize the rules concerning the execution model in:

$$wf\_R9R10(M) \ = \ wf\_activeCls(M) \vee wf\_passiveCls(M)$$

In order to prove that a model is well-formed we have to add relations about the well-formedness of:

- fields of any class

$$wf\_fields(M) \ = \ \forall x \in \{cls \mid is\_class(M, cls)\}, \ \forall y \in \{fld \mid fld \ is \ an \ attribute \ of \ x \ \}, \ wf\_Field(M, y)$$

- ports in the model

$$wf\_ports(M) \ = \ \forall x \in \{port \mid is\_port(M, port) \ \}, \ wf\_Port(M, x)$$

- associations in the model

$$wf\_assocs(M) \ = \ \forall x \in \{assoc \mid assoc \ is \ an \ association \ in \ M \ \}, \ wf\_Association(M, x)$$

- connectors in the model

[1]By $M$ we define an entity of type *model*. All the functions are defined on a model and take as value a Boolean.

$$wf\_links(M) \;=\; \forall x \in \{cls \mid is\_class(M, cls)\},\ \forall y \in \{conn \mid conn\ is\ a\ connector\ in\ x\},\ wf\_Link(M, y)$$

Also, a UML model should not contain cycles. We define two sets formed of tuples with the structure $(heir, parent)$. We give the relation that formalises this rule.

$$subint1(M) \;= \{(Intf, Intf') \mid is\_iface(M, Intf) \;\wedge\; is\_iface(M, Intf') \;\wedge\; Intf'\ is\ a\ parent\ of\ Intf\}$$
$$ws\_intf(M, Intf) = \forall Intf' \in \{si \mid si\ is\ a\ parent\ of\ Intf\},\ is\_iface(M, Intf') \;\wedge\; (Intf, Intf') \notin (subint1(M))^+$$
$$subcls1(M) \;= \{(Cls, Cls') \mid is\_class(M, Cls) \;\wedge\; is\_class(M, Cls') \;\wedge\; Cls'\ is\ a\ parent\ of\ Cls\}$$
$$ws\_clss(M, Cls) = \forall Cls' \in \{sc \mid sc\ is\ a\ parent\ of\ Cls\},\ is\_class(M, Cls') \;\wedge\; (Cls, Cls') \notin (subcls1(M))^+$$
$$ws\_model(M) \;=\; (\forall x \in \{Intf \mid is\_iface(M, Intf)\},\ ws\_iface(M, x)) \;\wedge\; (\forall y \in \{Cls \mid is\_class(M, Cls)\},\ ws\_clss(M, x))$$

Finally we can say that:

$$wf\_model(M) \;=\; wf\_R1R2(M) \wedge wf\_R9R10(M) \wedge wf\_fields(M) \wedge wf\_ports(M) \wedge wf\_assocs(M) \wedge wf\_links(M) \wedge ws\_model(M)$$

The purpose of the Isabelle/HOL formalisation is to prove the type-safety of the rule set observing composite structure. In order to achieve our goal we has to define recursive functions for which a termination proof was needed. Because such a proof could not be established with the tools provided by Isabelle, a partial termination proof was intended using relations. The definition of these relations is not so easy to determine. We sketch below such a function and a relation that might prove the termination but which is not sufficient (using this relation Isabelle/HOL does not know to induce the termination).

```
constdefs
   ws_wfrel :: "(model ⇒ (tname × tname)set) ⇒ ((model × tname) ×
(model × tname))set"
  "ws_wfrel R ≡ ((M, Intf), (M', Intf')). M'=M ∧ ws_model M ∧
(Intf',Intf)∈ R M"

function recIntf :: "model ⇒ tname ⇒ (tname) list"
where
"recIntf M Intf = ( case (iface_ M Intf) of  None ⇒ [] |
                                             Some (st,si,ri) ⇒
                        ( if ws_model M  then
                             if si=[]  then []
                             else  si @ (concat (map (λ x. recIntf M x) si))
                          else [])
                  )
"
by pat_completeness auto
termination apply ( relation "ws_wfrel subint1")
```

# Conclusions

Composite structures play an important role in modelling real-time embedded systems. They offer a clear structure of these systems and an initialization scheme for the objects contained. They are a big evolution of the UML standard version 2.x, since in the version 1.4 the initialization order of complex systems was user-defined. Since the standard is ambiguous and semantic variation point left open, we proposed to define a rule set observing composite structure and to prove its type safety.

We presented a definition and formalization of an operational model of UML composite structures, our approach being based on :

- dynamic typing of connectors based on a derived notion of *transported interfaces*;
- a set of static well-formedness rules, including type checking rules;
- a full definition of the default behaviour of *Ports*, and the means for defining port behaviour differing from the default (by using implicit port associations, etc.)
- rules for relating composite structures with the concurrency model.

The rule set defined in Chapter 3 is used by the type checker of the OMEGA UML compiler. In addition, the compiler goes all the way down to an operational implementation of composite structures, by translating OMEGA UML models (edited with any XMI 2.0 compatible UML editor) into IF models, for which a simulation and model-checking platform exists allowing us to prove the correctness of UML embedded models.

Experiments have been conducted to prove that models observing this rule set are correct. While the OMEGA UML compiler is able to catch all modelling errors when translating the model into its IF description, the OCL formalisation can also reveal these issues in a step preceding the translation. Applying this formalisation on the model, it yields the elements that do not comply with our profile catching many corner cases.

Even though the type-safety has not been proved yet (using the Isabelle/HOL formalisation), we were able to show on realistic models using the simulation and exhaustive state-space search from IFx2 Toolset that no routing problems (deadlocks in ports due to missing links, unexpected requests not conforming to object interfaces, etc.) exist in the model. The next step in our formalisation is to prove that the recursive functions we used terminate. In this case the type-safety means that: any request that travels through connectors reaches its terminus and every destination object receives only request compatible with its interfaces.

# Bibliography


[1] IFx Toolset. Available at http://www-omega.imag.fr/tools/IFx/IFx.php.

[2] TOPCASED, The Open-Source Toolkit for Critical Systems. Available at http://www.topcased.org/.

[3] ISO/IEC 8652/1995. *Ada 2005 Reference Manual. Language and Standard Libraries*, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[4] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM*, pages 237–267, 2004.

[5] Marius Bozga and Yassine Lakhnech. IF-2.0: Common Language Operational Semantics. Technical report, Verimag, 2002.

[6] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley, 2001.

[7] Arnaud Cuccuru, Sébastien Gérard, and Ansgar Radermacher. Meaningful Composite Structures. In *MoDELS*, pages 828–842, 2008.

[8] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Sci. Comput. Program.*, 55(1-3):81–115, 2005.

[9] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, Second International Symposium, FMCO 2003, Leiden, The Netherlands, November 4-7, 2003, Revised Lectures*, volume 3188 of *Lecture Notes in Computer Science*. Springer, 2004.

[10] Gregor Gößler and Joseph Sifakis. Priority systems. In de Boer et al. [9], pages 314–329.

[11] Object Management Group. Object Constraint Language, v2.2. Available at http://www.omg.org/spec/OCL/2.2/.

[12] Object Management Group. Systems Modeling Language, v1.1. Available at http://www.omg.org/spec/SysML/1.1/.

[13] Object Management Group. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. Available at http://www.omgmarte.org.

[14] Object Management Group. Unified Modeling Language, v2.2. Available at http://www.omg.org/spec/UML/2.2.

[15] IBM. Rational Rhapsody v7.5. reference manuals. Available at http://www.ibm.com/developerworks/rational/.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2009.

[17] Tobias Nipkow and David von Oheimb. $\text{Java}_{light}$ is type-safe - definitely. In *POPL*, pages 161–170, 1998.

[18] Iulian Ober and Iulia Dragomir. OMEGA2: A new version of the profile and the tools (regular paper). In *UML & AADL'2009 - 14th IEEE International Conference on Engineering of Complex Computer Systems, Oxford, Royaume Uni, 24/03/2010-25/03/2010*, pages 373–378, http://www.ieee.org/, 2010. IEEE.

[19] Iulian Ober and Iulia Dragomir. Unambigous composite structures: the OMEGA2 experience. Submitted to ACM/IEEE MODELS, 2010.

[20] Iulian Ober, Susanne Graf, and Ileana Ober. A real-time profile for UML. *International Journal on Software Tools for Technology*, 8(2):113–127, 2006.

[21] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *International Journal on Software Tools for Technology*, 8(2):128–145, 2006.

[22] David von Oheimb. *Analyzing Java in Isabelle/HOL - Formalization, Type Safety and Hoare Logic*. PhD thesis, Institut für Informatik, Lehrstuhl IV, 2000.

# Appendices

# OCL Formalization

```
-- HELPER FUNCTIONS

context Classifier

-- Verifies if the classifier is an interface
def: isInterface : Boolean = self.oclIsTypeOf(uml::Interface)

-- Verifies if the classifier is stereotyped with <<interfaceGroup>>
def: isInterfaceGroup : Boolean = self.getAppliedStereotypes()->select(
   name='portType')->size()<>0

-- Verifies if the classifier is stereotyped with <<protected>>
def: isProtected : Boolean = self.getAppliedStereotypes()->select(name='
   protected')->size()<>0


context Port

-- Verifies if the port is reversed (the port requires interfaces)
-- def: isReversed : Boolean = self.getAppliedStereotypes()->select(name
   ='reversed')->size()<>0

-- Since the tool used for the development of our models is IBM Rhapsody
   tool we need to make some adjustments since Rhapsody supports the
   reversed mechanism and saves it as an attribute of RhpPort stereotype
def: isReversed : Boolean = self.getValue(self.getAppliedStereotypes()->
   select(name='RhpPort')->asOrderedSet()->at(1),'isReversed').oclAsType(
   Boolean)


context Connector

-- Verifies if the connector is of type port-port
def: has2Ports : Boolean = self.end.role->select(oclIsTypeOf(uml::Port))
   ->size() = 2

-- Verifies if the connector is of type part-part
def: has2Parts : Boolean = self.end.role->select(oclIsTypeOf(uml::
   Property) and not oclIsTypeOf(uml::Port))->size() = 2

-- Verifies if the connector is of type port-part
def: has1PartAnd1Port : Boolean = self.end.role->select(oclIsTypeOf(uml::
   Property) and not oclIsTypeOf(uml::Port))->size() = 1

-- For a port-port connector verifies if one of the ports is owned by an
   inner component and the other one by the composite structure
-- For a port-part connector verifies if the port is owned by an inner
   component
def: has1PartWithPort : Boolean = self.end.partWithPort -> reject(
```

```
    oclIsTypeOf(OclVoid))->size() = 1

-- For a port-port connector verifies if both ports are owned by inner
    components
def: has2PartsWithPort: Boolean = self.end.partWithPort -> reject(
    oclIsTypeOf(OclVoid))->size() = 2

-- For a port-port connector returns the first port from the set of ends
-- For a port-part connector returns the port
def: port1 : Port = self.end.role->select(oclIsTypeOf(uml::Port))->
    asOrderedSet()->at(1).oclAsType(uml::Port)

-- For a port-port connector returns the second port from the set of ends
def: port2 : Port = self.end.role->select(oclIsTypeOf(uml::Port))->
    asOrderedSet()->at(2).oclAsType(uml::Port)

-- For a part-part connector returns the first part from the set of ends
-- For a port-part connector returns the part
def: part1 : Property = self.end.role->select(oclIsTypeOf(uml::Property)
    and not oclIsTypeOf(uml::Port))->asOrderedSet()->at(1).oclAsType(uml::
    Property)

-- For a part-part connector returns the second part from the set of ends
def: part2 : Property = self.end.role->select(oclIsTypeOf(uml::Property)
    and not oclIsTypeOf(uml::Port))->asOrderedSet()->at(2).oclAsType(uml::
    Property)

-- Verifies if the connector is typed with an association
def: isTyped : Boolean = (not self.type.oclIsTypeOf(OclVoid))

-- For a part-part connector, in the case of a unidirectional association
     that may type the link we need to know which end is navigable such
    that we can determine the direction of the link
def: isEnd1Navigable : Boolean = self.end.definingEnd->asOrderedSet()->at
    (1).isNavigable()

-- Determines the ends of the association with which the link is typed
def: associationEnds : OrderedSet(Property) = self.type.memberEnd->
    asOrderedSet()

-- Verifies if both ends of the association with which a link is typed
    are navigable
def: isBidirectional : Boolean = associationEnds->select(isNavigable())->
    size() = 2

-- Verifies if an association with which a link is typed is not navigable
def: isNotNavigable : Boolean = associationEnds->select(isNavigable())->
    size() = 0

-- Verifies if the association is between two classes
def: isClassClassAssociation : Boolean = associationEnds->select(type.
```

```
   oclIsKindOf(uml::Class))->size()=2

-- Verifies if the association is between two interfaces
def: isInterfaceInterfaceAssociation : Boolean = associationEnds->select(
   type.oclIsKindOf(uml::Interface))->size()=2

-- For an association that types a link determines the origine
def: associationStartPoint : Property =
  if isBidirectional
    then self.associationEnds->at(1)
  else
    self.associationEnds->select(not isNavigable())->at(1)
  endif

-- For an unidirectional association that types a link determines origine
   's type
def: associationStartPointType : Type = associationStartPoint.type

-- For an unidirectional association that types a link determines the
   target
def: associationEndPoint : Property =
  if isBidirectional
    then self.associationEnds->at(2)
  else
    self.associationEnds->select(isNavigable())->at(1)
  endif

-- For an unidirectional association that types a link determines target'
   s type
def: associationEndPointType : Type = associationEndPoint.type

-- Verifies if the association is between a class and an interface and it
    has this direction
def: isClassInterfaceAssociation : Boolean = associationStartPointType.
   oclIsKindOf(uml::Class) and associationEndPointType.oclIsKindOf(uml::
   Interface)


-- RULE 1 AND RULE 2

context Connector

-- Definition of link's type
def: linkType : String =
  if has2Parts
    then 'assembly link between parts'
  else
    if has2Ports
      then if has1PartWithPort
             then if not port1.isReversed and not port2.isReversed
                    then 'inbound delegation link between provided ports'
```

```
                else if port1.isReversed and port2.isReversed
                       then 'outbound delegation between required ports
                           '
                      else 'forbidden'
                      endif
                endif
           else if (port1.isReversed and port2.isReversed) or
                    (not port1.isReversed and not port2.isReversed)
                  then 'forbidden'
                else  'assembly between provided-required ports'
                endif
           endif
    else
      if has1PartWithPort
        then if not port1.isReversed
               then 'assembly link between part and provided port'
             else 'assembly link between part and required port'
             endif
      else if not port1.isReversed
             then 'inbound delegation link between part and provided port
                 '
           else 'outbound delegation link between part and required port'
           endif
      endif
    endif
  endif

-- Rule 1 and Rule 2
inv LinkType: self.linkType <> 'forbidden'


-- HELPER FUNCTIONS

context Connector

-- Verifies if a connector starts from the provided port given as
   parameter
def: isStartingFromProvidedPort (p:Port) : Boolean =
  (self.linkType = 'inbound delegation link between provided ports' and p
     .owner=self.owner) or
   self.linkType = 'inbound delegation link between part and provided
      port'

-- Verifies if a connector starts from the required port given as
   parameter
def: isStartingFromReversedPort (p:Port) : Boolean =
  (self.linkType = 'outbound delegation between required ports' and p.
     owner<>self.owner) or
   self.linkType = 'assembly between provided-required ports ' or
   self.linkType = 'assembly link between part and required port'
```

```
-- Verifies if a connector starts from the port given as parameter
def: isStartingFromPort (p:Port) : Boolean =
  isStartingFromProvidedPort(p) or isStartingFromReversedPort(p)

-- Verifies if a connector starts from a part
def: isStartingFromPart : Boolean =
  self.linkType = 'assembly link between part and provided port' or
  self.linkType = 'outbound delegation link between part and required
     port' or
  self.linkType = 'assembly link between parts'

-- For a port-port connector determines the port from which the link
   starts
-- For a port-part connector the starting port is the only port of the
   link
def: linkStartPort : Port =
  if has2Ports
    then
      if isStartingFromPort(port1)
        then port1
      else port2
      endif
  else port1
  endif

-- For a port-port connector determines the port in which the link ends
-- For a port-part connector the ending port is the only port of the link
def: linkEndPort : Port =
  if has2Ports
    then if isStartingFromPort(port1)
           then port2
         else port1
         endif
  else port1
  endif

-- For a port-part connector the starting part is the only part of the
   link
-- For a part-part connector determines the part from which the link
   starts (this part it is not navigable)
def: linkStartPart : Property =
  if has1PartAnd1Port
    then part1
  else
    if isEnd1Navigable
      then part2
    else part1
    endif
  endif

-- For a port-part connector the ending part is the only part of the link
```

```
-- For a part-part connector determines the part in which the link ends (
   this part it is navigable)
def: linkEndPart : Property =
  if has1PartAnd1Port
    then part1
  else
    if isEnd1Navigable
      then part1
    else part2
    endif
  endif


-- PROVIDED / REALIZED INTERFACES

context Port

-- Definition of interfaces realized by a port
def: interfaces : Set(Classifier) = self.provided->reject(
   isInterfaceGroup)

context Classifier

-- Definition of classifier's parents recursive computation
def: getParentsRec : Set(Classifier) =  self.general->union(self.general
   ->iterate(p:Classifier; res:Set(Classifier)=Set{}| res->union(p.
   getParentsRec)))

context Class

-- Definition of all interfaces directly realized by a class
def: iRealizations : Set(Classifier) = self.interfaceRealization.contract
   ->asSet()

-- Definition of interfaces provided by a class directly or indirectly
   realized (used in the case of a link not typed by an association)
def: interfaces : Set(Classifier) =
  iRealizations->union(iRealizations->iterate(i:Interface; res:Set(
     Classifier)=Set{}| res->union(i.getParentsRec)))
  ->union(self.getParentsRec->iterate(c:Class; res:Set(Classifier)=Set{}|
      res->union(c.interfaces)))
  ->reject(isInterfaceGroup)

context Interface

-- Definition of interfaces provided by an interface (used in the case of
    a link typed by an association pointing to an interface)
def: interfaces : Set(Classifier) = self.oclAsType(uml::Classifier)->
   asSet()->union(self.getParentsRec)->select(not isInterfaceGroup and
   isInterface)
```

```
context Type

-- Determines the set of provided interfaces by a class or an interface
def: interfaces : Set(Classifier) =
  if self.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces
  else
    self.oclAsType(uml::Class).interfaces
  endif


-- RULE 6

context Connector

-- Definition of the set of transported interfaces
def: setTransportedInterfaces : Set(Classifier) =
  if has2Parts
    then Set{OclInvalid}
  else if has2Ports
         then if isTyped
                then if isStartingFromPort(port1)
                       then (port1.interfaces) -> intersection(
                          associationEndPointType.interfaces)
                     else (port2.interfaces) -> intersection(
                        associationEndPointType.interfaces)
                     endif
              else (port1.interfaces) -> intersection(port2.interfaces)
              endif
       else
         if isTyped
           then (port1.interfaces) -> intersection(
              associationEndPointType.interfaces)
         else (port1.interfaces) -> intersection(part1.type.interfaces)
         endif
       endif
  endif

-- Rule 6
inv SetOfTransportedInterfacesNonEmpty: self.setTransportedInterfaces->
    size()<>0


-- HELPER FUNCTIONS

context Classifier

-- Verifies if the current classifier is compatible with the one given as
def: isCompatible(c:Classifier) : Boolean =
  if self.oclIsKindOf(uml::Interface) and c.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces->includes(c)
```

```
  else
    if self.oclIsKindOf(uml::Class) and c.oclIsKindOf(uml::Interface)
      then self.oclAsType(uml::Class).interfaces->includes(c)
    else
     (c->asSet()->union(c.getParentsRec()))->includes(self)
    endif
  endif

context Type

-- Verifies if the link end's type (the type of a part) is compatible
   with the association end's type given as parameter
def: isCompatible(t:Type) : Boolean = self.oclAsType(uml::Classifier).
   isCompatible(t.oclAsType(uml::Classifier))

context Port

-- Verifies if the current port is compatible with the association end's
   type given as parameter
def: isCompatible(t:Type) : Boolean = self.interfaces->includesAll(t.
   oclAsType(uml::Interface).interfaces)


-- RULE 3 AND RULE 4

context Connector

-- Verifies if a link starting from a port and typed with an association
   has the same direction with the association and the interface pointed
   is included in the set of transported interfaces
def: linkStartingFromPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isInterfaceInterfaceAssociation
        then if has1PartAnd1Port
              then linkStartPort.isCompatible(associationStartPointType)
                  and linkEndPart.type.isCompatible(
                  associationEndPointType) and
                    setTransportedInterfaces->includes(
                        associationEndPointType.oclAsType(uml::Classifier))
             else linkStartPort.isCompatible(associationStartPointType)
                 and linkEndPort.isCompatible(associationEndPointType) and
                   setTransportedInterfaces->includes(
                       associationEndPointType.oclAsType(uml::Classifier))
             endif
    else false
    endif
  endif

-- Verifies if a link starting from a port is well-formed
```

```
def: linkStartingFromPort : Boolean =
  if (not isStartingFromPart) and isTyped
    then linkStartingFromPortVerification
  else true
  endif

-- Rule 3 and Rule 4
inv LinkStartingFromPort: self.linkStartingFromPort


-- RULE 3 AND RULE 5

context Connector

-- For a link between two parts verifies if the ends are compatible with
   the corresponding ends of the accepted association
def: linkPartPartVerification : Boolean =
  if isNotNavigable
    then false
  else
    if isBidirectional
      then (linkStartPart.type.isCompatible(associationStartPointType)
          and linkEndPart.type.isCompatible(associationEndPointType)) or
            (linkStartPart.type.isCompatible(associationEndPointType) and
               linkEndPart.type.isCompatible(associationStartPointType))
    else (linkStartPart.type.isCompatible(associationStartPointType) and
       linkEndPart.type.isCompatible(associationEndPointType))
    endif
  endif

-- For a link between a part and a port verifies if the ends are
   compatible with the corresponding ends of accepted association
def: linkPartPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isClassClassAssociation
      then false
    else
      if isInterfaceInterfaceAssociation
        then (linkStartPart.type.isCompatible(associationStartPointType)
           and linkEndPort.isCompatible(associationEndPointType))
      else
        if isClassInterfaceAssociation
          then linkStartPart.type.isCompatible(associationStartPointType)
              and linkEndPort.isCompatible(associationEndPointType)
        else false
        endif
      endif
    endif
  endif
```

```
-- Verifies if a link starting from part is typed with an association and
    if it is well-formed
def: linkStartingFromPart : Boolean =
  if isStartingFromPart
    then if has2Parts
           then isTyped and linkPartPartVerification
         else isTyped and linkPartPortVerification
         endif
  else true
  endif

-- Rule 3 and Rule 5
inv LinkStartingFromPart: self.linkStartingFromPart


-- HELPER FUNCTIONS

context Port

-- Determines all the connectors starting or ending in a port
def: connectors : Set(Connector) = self.end.owner.oclAsType(uml::
   Connector)->asSet()

-- Verifies if a port has connectors starting or ending in it
def: hasConnectors : Boolean = self.connectors->size()<>0

-- Determines all the connectors starting or ending in a port that are
   not typed with an association
def: connectorsNotTyped : Set(Connector) = self.connectors->select(type.
   oclIsTypeOf(OclVoid))

-- Verifies if a port has connectors not typed with an association that
   are starting or ending in it
def: hasConnectorsNotTyped : Boolean = self.connectorsNotTyped->size()<>0

-- Determines all the connectors starting from a port (if the boolean
   parameter is true it collects all the connectors; if it is false it
   collects only the connectors not typed with association)
def: connectorsStartingFromPort(withType:Boolean) : Set(Connector) =
  if withType then
    if self.isReversed
      then self.connectors->select(c:Connector|c.
         isStartingFromReversedPort(self))
    else
      self.connectors->select(c:Connector|c.isStartingFromProvidedPort(
         self))
    endif
  else
    if self.isReversed
      then self.connectorsNotTyped->select(c:Connector|c.
```

```
          isStartingFromReversedPort(self))
    else
      self.connectorsNotTyped->select(c:Connector|c.
          isStartingFromProvidedPort(self))
    endif
  endif

-- Verifies if a port is the origine for at least one connector
def: isStartingPort : Boolean = self.connectorsStartingFromPort(true)->
    size() <> 0

-- Determines the union of the sets of transported interfaces on each
    link starting from the port
def: unionSetForTransportedInterfacesOnLinks(withType:Boolean) : Set(
    Classifier) =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; s:Set(
      Classifier)=Set{} | s->union(c.setTransportedInterfaces))

-- Determines the sum of the number of transported interfaces on each
    link starting from the port
def: noOfTransportedInterfacesOnLinks(withType:Boolean) : Integer =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; i:
      Integer=0 | i + (c.setTransportedInterfaces->size()))


-- RULE 7

context Port

-- Definition of pairwise disjoint sets of transported interfaces
def: isPairwiseDisjoint : Boolean =
  if isStartingPort and hasConnectorsNotTyped
    then  if unionSetForTransportedInterfacesOnLinks(false)->size() <>
        noOfTransportedInterfacesOnLinks(false)
            then false
          else true
          endif
  else true
  endif

-- Rule 7
inv PairwiseDisjoint: self.isPairwiseDisjoint


-- RULE 8

context Port

-- Verifies if the union of sets of transported interfaces is equal to
    the interfaces provided/required
def: isComplete : Boolean =
```

```
  if isStartingPort
    then unionSetForTransportedInterfacesOnLinks(true) = self.interfaces
  else true
  endif

-- Rule 8
inv Completeness: self.isComplete


-- HELPER FUNCTIONS

context Class

-- Determines the number of parts of a composite structure
def: noOfComponents : Integer = self.part->size()

-- Verifies if a class is a composite structure
def: isComposite : Boolean = noOfComponents <> 0

-- Determines the number of active parts owned by a composite structure
def: noOfActiveComponents : Integer = self.part.type.oclAsType(uml::Class
   )->select(isActive=true)->size()

-- Determines the number of passive parts owned by a composite structure
def: noOfPassiveComponents : Integer = self.part.type.oclAsType(uml::
   Class)->select(isActive=false)->select(not isProtected)->size()

-- Determines the number of protected parts owned by a composite
   structure
def: noOfProtectedComponents : Integer = self.part.type.oclAsType(uml::
   Class)->select(isProtected)->size()


-- RULE 9 AND RULE 10

context Class

-- Definition of a well-formed class
def: isWellFormed : Boolean =
  if self.isActive and isComposite
    then  if noOfActiveComponents + noOfProtectedComponents =
       noOfComponents or
            noOfPassiveComponents = noOfComponents
          then true
        else false
        endif
  else
    if (not self.isActive) and (not isProtected) and isComposite
      then if noOfPassiveComponents = noOfComponents
            then true
          else false
```

```
          endif
    else OclInvalid
    endif
  endif

-- Rule 9 and Rule 10
inv CompositeStructure: self.isWellFormed <> false


-- RULE 11

context Class

-- Determines the number of observer parts owned by a composite structure
def: noOfObservers : Integer = self.part.type.oclAsType(uml::Class)->
   select(isObserver)->size()

-- Definition of a well formed observer
def: isObserverWellFormed : Boolean =
  if self.isObserver and isComposite
    then noOfComponents = noOfObservers
  else true
  endif

-- Rule 11
inv CompositeObserver: self.isObserverWellFormed <> false
```

# Isabelle/HOL Formalization

**theory** *Table*
**imports** *Main*
**begin**

**types** *('a,'b)table = "'a ⇒ 'b option"*

**syntax**
*table_of :: "('a × 'b) list ⇒ ('a, 'b)table"*

**translations**
*"table_of" == "map_of"*

**end**

```
theory Omega
imports Main
begin
```

* Stereotype definitions

* Class concurrency

```
datatype clssConcurrency = active
                         | passive
                         | protected
```

* Interface type from port's contract point of view

```
datatype intfType = interfaceGroup
                  | none
```

* Port direction: the interface from the contract can be required or provided depending if the port is reversed or not

```
datatype portDirection = provided
                       | required
```

* Datatype definition

* Primitive datatype

```
datatype primDT = boolean
                | integer
                | real
                | Timer
```

* Reference datatype from the model

* Opaque datatype for name representation

```
typedecl tname arities tname::eq
```

* Reference datatype

```
datatype refDT = NullT
               | IfaceT tname
               | ClassT tname
```

* Datatype definition

```
datatype dt = PrimDT primDT
```

```
          | RefDT  refDT
```

* Elements simpler definition

**syntax**

```
  NT    ::          "dt"
  Iface :: "tname ⇒ dt"
  Class :: "tname ⇒ dt"
```

**translations**

```
  "NT"      == "RefDT NullT"
  "Iface I" == "RefDT (IfaceT I)"
  "Class C" == "RefDT (ClassT C)"
```

* Field declaration

* Opaque datatype for expression name

**typedecl** *ename* **arities** *ename::eq*

* field_name × field_type

**types** *fdecl = "ename × dt"*

* Interface declaration

* interface_type × superinterfaces_list × realized_interfaces_list

**types** *intfTb = "intfType × (tname)list × (tname)list"*

**types** *intfDecl = "tname × intfTb"*

* Port declaration

* port_direction × port_contract × port_owner

**types** *portTb = "portDirection × tname × tname"*

**types** *portDecl = "tname × portTb"*

* Association declaration

* association_end_1 × is_end_1_navigable × association_end_2 × is_end_2_navigable

**types** *assocTb = "tname × bool × tname × bool"*

**types** *assocDecl = "ename × assocTb"*

* Connector declaration

* connector_end_1 × connector_end_2 × owner × is_typed × association

**types** *connTb = "tname × tname × tname × bool × (assocDecl option)"*

**types** *connDecl = "ename × connTb"*

* Class declaration

* class_type × superclasses_list × realized_interfaces_list

**types** *csig = "clssConcurrency × (tname)list × (tname)list"*

* fields_list × ports_list × connectors_list

**types** *cbodyP = "(fdecl)list × (portDecl)list × (connDecl)list"*

**types** *clssTb = "csig × cbodyP"*

**types** *clssDecl = "tname × clssTb"*

* Model declaration

**types** *model = "(clssDecl)list × (intfDecl)list × (portDecl)list"*

**end**

**theory** *WFRules*
**imports** *Main Omega Table*
**begin**

**syntax**
```
  iface_   :: "model ⇒ (tname, intfTb) table"
  class_   :: "model ⇒ (tname, clssTb) table"
  port_    :: "model ⇒ (tname, portTb) table"
```

**translations**
```
  "iface_ M Intf" == "table_of (fst (snd M)) Intf"
  "class_ M Clss" == "table_of (fst M) Clss"
  "port_  M Prt"  == "table_of (snd (snd M)) Prt"
```

**syntax**
```
  is_iface :: "model ⇒ tname ⇒ bool"
  is_class :: "model ⇒ tname ⇒ bool"
  is_port  :: "model ⇒ tname ⇒ bool"
```

**translations**
```
  "is_iface M Intf" == "iface_ M Intf ≠ None"
  "is_class M Clss" == "class_ M Clss ≠ None"
  "is_port  M Prt " == "port_  M Prt  ≠ None"
```

**syntax**
```
  is_provided    :: "portTb ⇒ bool"
  is_required    :: "portTb ⇒ bool"
  is_active      :: "clssTb ⇒ bool"
  is_passive     :: "clssTb ⇒ bool"
  is_protected   :: "clssTb ⇒ bool"
  is_intfGroup   :: "intfTb ⇒ bool"
```

**translations**
```
  "is_provided    Prt"  == "(fst Prt) = provided"
  "is_required    Prt"  == "¬ (is_provided Prt)"
  "is_active      Clss" == "(fst(fst Clss)) = active"
  "is_passive     Clss" == "(fst(fst Clss)) = passive"
  "is_protected   Clss" == "(fst(fst Clss)) = protected"
  "is_intfGroup   Intf" == "(fst Intf) = interfaceGroup"
```

**constdefs**
```
   is_primitiveDT :: "dt ⇒ bool"
  "is_primitiveDT var ≡ case var of (PrimDT prim) ⇒ True  |
                                    (RefDT  ref ) ⇒ False"
   is_referenceDT :: "dt ⇒ bool"
```

```
  "is_referenceDT var ≡ case var of (PrimDT prim) ⇒ False  |
                                    (RefDT  ref ) ⇒ True"
```

**constdefs**

```
   is_defined :: "model ⇒ dt ⇒ bool"
  "is_defined M var ≡ ( if is_primitiveDT var
                          then True
                        else
                         (case var of Class Clss ⇒ is_class M Clss |
                                      Iface Intf ⇒ is_iface M Intf |
                                      Null       ⇒ False
                         )
                       )"
```

**constdefs**

```
   getReferenceDT_Name :: "dt ⇒ tname"
  "getReferenceDT_Name var ≡ case var of (Iface Intf) ⇒ Intf |
                                         (Class Clss) ⇒ Clss "
```

**primrec** `parts :: "(fdecl)list ⇒ (fdecl)list"`
**where**

```
 " parts [] = []" |
 " parts (elem # rList) = (if is_referenceDT (snd elem)
                            then elem # (parts rList)
                           else (parts rList)
                          )"
```

**constdefs**

```
  get_parts :: "clssDecl ⇒ (fdecl)list"
 "get_parts Clss ≡ parts (fst(snd(snd Clss)))"
```

* The ends of a connector can be classes and / or ports

**constdefs**

```
   wf_Link :: "model ⇒ connDecl ⇒ bool"
  "wf_Link M Conn ≡ (let x=(fst(snd Conn)); y=(fst (snd(snd Conn)));
     owner=(fst (snd(snd(snd Conn)))) in ((is_class M x ∧
     (is_class M y ∨ is_port M y)) ∨ (is_port M x ∧ (is_class M y ∨
     is_port M y))) ∧ (is_class M owner))"
```

**constdefs**

```
   wf_links :: "model ⇒ bool"
  "wf_links M ≡ ∀x∈(set(fst M)). ∀y∈(set(snd(snd(snd(snd x))))).
     (wf_Link M y)"
```

* The ends of an association can be classes and / or interfaces

**constdefs**

```
  wf_Association :: "model ⇒ assocDecl ⇒ bool"
"wf_Association M Assc ≡ (let x=(fst(snd Assc)); y=(fst(snd(snd
   (snd Assc)))) in (is_class M x ∧ (is_class M y ∨ is_iface M y))
   ∨ (is_iface M x ∧ (is_class M y ∨ is_iface M y)))"
```

**constdefs**

```
  wf_assocs :: "model ⇒ bool"
"wf_assocs M ≡ ∀x∈(set(fst M)). ∀y∈(set(snd(snd(snd(snd x))))).
   if (fst(snd(snd(snd(snd y)))))=True then wf_Association M
   (the(snd(snd(snd(snd(snd y)))))) else True"
```

* The type of a field must exist in the model; if is a reference it can be only a class instance

**constdefs**

```
  wf_Field :: "model ⇒ fdecl ⇒ bool"
"wf_Field M Fld ≡ is_primitiveDT (snd Fld) ∨ (is_defined M
    (snd Fld) ∧ is_class M (getReferenceDT_Name (snd Fld)))"
```

**constdefs**

```
  wf_fields :: "model ⇒ bool"
"wf_fields M ≡ ∀x∈(set(fst M)). ∀y∈(set(fst(snd(snd x)))).
    (wf_Field M y)"
```

* The contract of a port can be an interface or a class; the limitation that the port has only one item as contract is given by the Abstract Syntax Tree of the profile

**constdefs**

```
 wf_Port :: "model ⇒ portDecl ⇒ bool"
"wf_Port M Prt ≡ (let Contr = fst(snd(snd Prt));
   owner=snd(snd(snd Prt)) in (is_iface M Contr ∨ is_class M Contr)
   ∧ (is_class M owner) )"
```

**constdefs**

```
  wf_ports :: "model ⇒ bool"
"wf_ports M ≡ ∀x∈(set(fst M)). ∀y∈(set(fst(snd(snd(snd x))))).
   (wf_Port M y)"
```

* Rule 1 and Rule 2 datatype

```
datatype linkTP = assembly_parts
                | assembly_provided_required_ports
                | assembly_part_provided_port
```

```
                | assembly_part_required_port
                | inbound_delegation_provided_ports
                | inbound_delegation_part_provided_port
                | outbound_delegation_required_ports
                | outbound_delegation_part_required_port
                | forbidden
```

* Rule 1 and Rule 2 computing function

**constdefs**

```
   linkType :: "model ⇒ connDecl ⇒ linkTP"
  "linkType M Conn ≡ (let end1=fst(snd Conn); end2=fst(snd(snd(Conn)));

    owner=fst(snd(snd(snd Conn))) in
(if (is_class M end1) ∧ (is_class M end2) then
   assembly_parts
 else
  if (is_port M end1) ∧ (is_port M end2) then
    if owner=snd(snd(the(port_ M end1))) ∨
       owner=snd(snd(the(port_ M end2))) then
      if is_provided (the(port_ M end1)) ∧
         is_provided (the(port_ M end2)) then
        inbound_delegation_provided_ports
      else if is_required (the(port_ M end1)) ∧
              is_required (the(port_ M end2)) then
             outbound_delegation_required_ports
           else forbidden
     else if (is_provided (the(port_ M end1)) ∧
              is_provided (the(port_ M end2))) ∨
             (is_required (the(port_ M end1)) ∧
              is_required (the(port_ M end2))) then
            forbidden
          else assembly_provided_required_ports
  else if (is_port M end1) ∧ (is_class M end2) then
         if owner=snd(snd(the(port_ M end1))) then
           if is_provided (the(port_ M end1)) then
             inbound_delegation_part_provided_port
           else outbound_delegation_part_required_port
         else if is_provided (the(port_ M end1)) then
                assembly_part_provided_port
              else assembly_part_required_port
        else if owner=snd(snd(the(port_ M end2))) then
               if is_provided (the(port_ M end2)) then
                 inbound_delegation_part_provided_port
               else outbound_delegation_part_required_port
             else if is_provided (the(port_ M end2)) then
```

```
                    assembly_part_provided_port
                  else assembly_part_required_port
))"
```

* Rule 1 and Rule 2

* A class has only well-formed links

**constdefs**

```
  wf_LinkType :: "model ⇒ clssDecl ⇒ bool"
 "wf_LinkType M Clss ≡ ∀Conn∈set(snd (snd (snd (snd Clss)))).
    linkType M Conn ≠ forbidden"
```

* A model has only well-formed classes regarding Rule 1 and Rule 2

**constdefs**

```
  wf_R1R2 :: "model ⇒ bool"
 "wf_R1R2 M ≡ ∀x∈(set(fst M)). wf_LinkType M x"
```

* Rule 9 and Rule 10

* A passive class is formed only by passive components

**constdefs**

```
  wf_PassiveClssConcurrency :: "model ⇒ clssDecl ⇒ bool"
 "wf_PassiveClssConcurrency M Clss ≡ ∀Fld∈set(get_parts Clss).
    is_passive(the(class_ M (getReferenceDT_Name (snd Fld))))"
```

**constdefs**

```
  wf_passiveCls :: "model ⇒ bool"
 "wf_passiveCls M ≡ ∀x∈(set(fst M)). if (is_passive (snd(x)))
    then wf_PassiveClssConcurrency M x else True"
```

* An active class is formed only or by active and protected components or by passive components

**constdefs**

```
  wf_ActiveClssConcurrency :: "model ⇒ clssDecl ⇒ bool"
 "wf_ActiveClssConcurrency M Clss ≡ (∀Fld∈set(get_parts Clss).
    is_active(the(class_ M (getReferenceDT_Name (snd Fld)))) ∨
    is_protected(the(class_ M (getReferenceDT_Name (snd Fld))))) ∨
    (∀Fld∈set(get_parts Clss).
    is_passive(the(class_ M (getReferenceDT_Name (snd Fld)))))"
```

**constdefs**

```
  wf_activeCls :: "model ⇒ bool"
 "wf_activeCls M ≡ ∀x∈(set(fst M)). if (is_active (snd(x)))
```

```
    then wf_ActiveClssConcurrency M x else True"
```

* Rule 9 and Rule 10

**constdefs**

```
  wf_R9R10 :: "model ⇒ bool"
"wf_R9R10 M ≡ wf_activeCls M ∨ wf_passiveCls M"
```

* Set with elements of the form (heir,parent) for interfaces

**constdefs**

```
subint1 :: "model ⇒ (tname × tname)set"
subint1_def : "subint1 M ≡ {(Intf,Intf'). is_iface M Intf ∧
   is_iface M Intf' ∧ Intf'∈set(fst(snd(the(iface_ M Intf))))}"
```

* No cycles in the model for interfaces

**constdefs**

```
  ws_idecl :: "model ⇒ tname ⇒ (tname)list ⇒ bool"
"ws_idecl M Intf si ≡ ∀Intf'∈set(si). is_iface M Intf' ∧
   (Intf',Intf)∉(subint1 M)^+"
```

* Set with elements of the form (heir,parent) for classes

**constdefs**

```
subcls1 :: "model ⇒ (tname × tname)set"
subcls1_def : "subcls1 M ≡ {(Cls,Cls'). is_class M Cls ∧
   is_class M Cls' ∧ Cls'∈set(fst(snd(fst(the(class_ M Cls)))))}"
```

* No cycles in the model for classes

**constdefs**

```
  ws_cdecl :: "model ⇒ tname ⇒ (tname)list ⇒ bool"
"ws_cdecl M Cls sc ≡ ∀Cls'∈set(sc). is_class M Cls' ∧
   (Cls',Cls)∉(subcls1 M)^+"
```

* No cycles in the model

**constdefs**

```
  ws_model :: "model ⇒ bool"
"ws_model M ≡ (let clsSet=set(fst M); intfSet=set(fst(snd M)) in
   ((∀(Intf,(st,si,ri))∈intfSet. ws_idecl M Intf si) ∧
   (∀(Cls,(st,sc,ri),(fl,pr,cn))∈clsSet. ws_cdecl M Cls sc ))) "
```

* Well-formed model

**constdefs**

```
   wf_model :: "model ⇒ bool"
  "wf_model M ≡ (wf_R1R2 M) ∧ (wf_R9R10 M) ∧ (wf_fields M) ∧
     (wf_links M) ∧ (wf_ports M) ∧ (wf_assocs M) ∧ (ws_model M)"
```

* Recursive computation of the tree of superinterfaces starting from an interface

**constdefs**

```
   ws_wfrel :: "(model ⇒ (tname × tname)set) ⇒ ((model × tname) ×
     (model × tname))set"
  "ws_wfrel R ≡ {((M, Intf), (M', Intf')). M'=M ∧ ws_model M ∧
     (Intf',Intf)∈ R M}"
```

**function** *recIntf :: "model ⇒ tname ⇒ (tname)list"*
**where**

```
"recIntf M Intf = (case (iface_ M Intf) of
   None  ⇒ [] |
   Some (st,si,ri) ⇒ (if ws_model M then
                        if si=[] then []
                        else si @ (concat (map (λx. recIntf M x)
                          si))
                        else []
                      )
                  )
"
```

**end**