TECHNICAL REPORT
IRIT/RT–2013-14–FR

Integrating verifiable Assume/Guarantee contracts in UML/SysML

Iulia Dragomir — Iulian Ober — Christian Percebois

Université de Toulouse - IRIT
118 Route de Narbonne, 31062 Toulouse, France
{iulia.dragomir,iulian.ober,christian.percebois}@irit.fr

July 15, 2013

Abstract

A compositional approach based on components and driven by requirements is a common method used in the development of critical real-time embedded systems. Since the satisfaction of a requirement is subject to the composition of several components, defining abstract and partial behaviors for components with respect to the point of view of the requirement allows for a manageable design of systems. In this paper we consider such specifications in the form of contracts. A contract for a component is a pair (assumption, guarantee) where the assumption is an abstraction of the component's environment behavior and the guarantee is an abstraction of the component's behavior given that the environment behaves like the assumption. In previous work we have defined a formal contract-based theory for Timed Input/Output Automata with the aim of using it to express the semantics UML/SysML models. In this paper we propose an extension of the UML/SysML language with a syntax and semantics for contracts and for the relations they must satisfy. Besides the important role that contracts have in design, they can also be used for the verification of requirement satisfaction and for their traceability.

Keywords

component, contract, compositional reasoning, UML/SysML, meta-model, design, well-formedness, V&V

Contents

1 Introduction

Nowadays critical real-time embedded systems grow larger is size and more complex. Their development is a challenging task and is often error-prone. A way for system designers to tackle this issue is to use a compositional approach driven by requirements. For example, process-oriented standards such as DO-178C [12] highlight the need to model requirements at different levels of abstractions during development and to ensure their traceability at each design iteration step.

However, requirements are often difficult to be mapped to components: several components combine together to satisfy a requirement and a component may be involved in the satisfaction of several requirements. In order to achieve provably correct compositional design, one needs a way to abstractly specify how a particular component K participates in fulfilling a requirement φ. Such a specification can take the form of a *contract*: a pair (*assumption*, *guarantee*) where the *assumption* is an abstraction of K's environment behavior and the *guarantee* is an abstraction of K's behavior given that the environment behaves according to the assumption. Such a contract can then be used to model the point of view of the component with respect to the requirement φ. Contracts for reactive and real-time components have received a lot of attention from the research community recently, as discussed in §5.

Besides the important role contracts play in system design, they can also be used as basic blocks for compositional verification of requirement satisfaction. In [9] we have introduced a contract-based theory for compositional verification of systems of communicating Timed Input/Output Automata (TIOA) with the intention to use it as underlying semantics for contract-based UML [17]/SysML [16] modeling and verification. This paper complements the formal theory by extending UML/SysML with the language elements needed for modeling contracts and their relations.

Paper structure. In §2 we summarize the contract-based reasoning theory we have defined and we present the OMEGA UML/SysML Profile [7] on which we want to apply the formal theory. In §3 we propose a meta-model for the contract theory and a set of constraints and well-formedness rules needed to make the system model verifiable with contracts. Then, an instantiation of the meta-model for the OMEGA UML/SysML Profile is discussed. §4 presents the application of this technique to an industry-grade system model, the ATV SGS case study previously described without contracts in [8], before concluding.

2 Background

2.1 Timed input/output automata

Many mathematical formalisms have been proposed in the literature for modeling communicating timed reactive components. Our work is based on a variant of Timed Input/Output Automata of [11] since it is one of the most general formalisms, thoroughly defined and for which several interesting compositionality results are already available. A TIOA specifies a state space and a set of admitted timed behaviors for a component. The parallel composition of TIOAs (denoted $\|$ in the following) is based on synchronization of corresponding inputs/outputs and the interleaving of other actions. For full definitions of all these notions, the reader is referred to [11].

2.2 A UML/SysML profile based on timed input/output automata

In previous work [4] we have considered the high-level modeling of embedded real-time systems in UML/SysML with a semantics provided in terms of the TIOA. The result of this work is a semantic profile for UML and SysML called OMEGA and a set of tools for simulating and model-checking OMEGA models.

In OMEGA, the architecture of a system is expressed in the usual way in UML (class diagrams) and in SysML (block definition diagrams and internal block diagrams). Classes/blocks may use most of their features: properties (attributes and parts), signals receptions, interconnection elements (port, connector, interface) and relations (association, composition and generalization). The hierarchical architecture of components (and systems) is specified through composite structures.

The behavior of atomic components is modeled by state machines with usual UML actions on transitions. The operational semantics of each component instance is a timed input/output automaton. The TIOAs corresponding to components are composed in parallel and communicate by asynchronous signal exchanges. This imposes that all communications between objects/block instances are defined as signal outputs and receptions that are transferred via ports and connectors. Ports need to be typed with interfaces that contain the list of signals transferred to or from the component's environment.

The temporal elements of TIOA, such as clocks, clock actions and timed guards actions have corresponding language constructs in the OMEGA profile. The elapsing of time is constrained by *transition urgency* stereotypes inspired from timed automata with urgency [3]: time delay is *blocked* if one of the active transitions in the current state is stereotyped ≪*eager*≫, it is *upper-bounded* if an active transition is stereotyped ≪*delayable*≫ and is unbounded otherwise (i.e., if all active transitions are stereotyped ≪*lazy*≫, which is the default).

The profile also proposes mechanisms for formalizing requirements, in particular in the form of real-time safety properties described by *observer* classes (identified by a stereotype ≪*observer*≫). The state machine of these classes uses special primitives for monitoring the system state and events and give verdicts about the (non-)satisfaction of a property by using labels (e.g., stereotype ≪*error*≫) on states.

For a more complete description of the UML/SysML component model used in OMEGA and of the mapping between the modeling concepts and the underlying TIOA semantics, the reader is referred to [15].

2.3 A theory of contracts for timed input/output automata

In [9] we defined a theory for modeling and reasoning with contracts for TIOA. The theory is an extension of a meta-theory defined in [18]. A contract for a TIOA component K is a pair of TIOAs that model the assumption (A) and the guarantee (G). The satisfaction of the contract is defined formally by a relation based on trace inclusion between $K \parallel A$ and $G \parallel A$ modulo the set of actions that are of interest for the contract. The theory also provides the necessary mechanisms for compositional reasoning with contracts, explained in the following.

Consider that the objective is to prove that a system S composed of several components $K_1, K_2, \cdots, K_n$ satisfies a property φ (see Fig. 1) under a certain hypothesis A on the behavior of its environment. The method consists in defining a more abstract specification G of the system such that $A \parallel G$ satisfies φ (*conformance* step in Fig. 1). However, it is often impossible to verify directly (i.e. by checking trace set inclusion) that the composite system S satisfies the contract because of the combinatorial explosion of the state space. To avoid this problem, the method defined in [9, 18] uses a decomposition of the proof in independent steps based on the definition of a set of individual contracts C_1, C_2... for the components K_1, K_2..., which, when put together, ensure the global contract $C = (A, G)$. We say that $\{C_1, C_2, \cdots\}$ *dominates* C (*dominance* step in Fig. 1). The theory in [9, 18] provides a set of *sufficient conditions* for dominance which can be checked independently with lesser combinatorial complexity. In addition to the conditions for dominance, one also has to check that each component K_i satisfies the contract C_i (*conformance* step in Fig. 1).

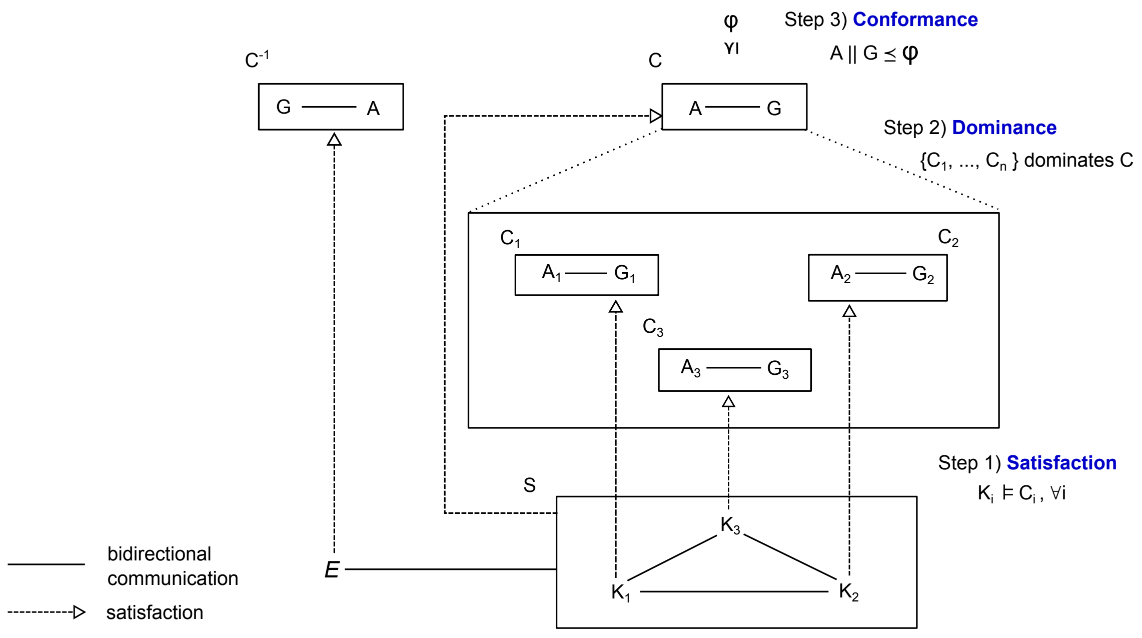

Figure 1: Contract-based reasoning for a subsystem with three components [18].

The method described before does not explicitly prescribe how to derive the contracts for the whole system and for the components. In the case study described in §4, this task was relatively straightforward: since we make no additional assumption (A) about the environment, G is roughly the same as φ, and the component guarantees are a projection of the desired global guarantee. There may however be cases where the definition of contracts is less obvious and further work will be needed to define methodological guidelines for constructing the intermediate contracts, but this is outside the scope of this paper.

3 Extending UML/SysML for modeling contracts

In this section we present the UML/SysML extensions that we propose in order to support modeling and reasoning with contracts. We first describe a domain meta-model of the contract-related concepts in §3.1. We then discuss in §3.2 the constraints and well-formedness rules imposed on the key notions in order to make models with contracts compliant with the theory from [9] and thus verifiable. Finally, in §3.3 we discuss the mapping of the meta-model concepts as UML/SysML profile, using the standard extension mechanisms (stereotypes).

3.1 A meta-model for contracts in UML/SysML

Within the contract theory we have presented there are two categories of concepts: (1) those related to contract modeling represented in the upper part of the meta-model given in Figure 2 and (2) those related to modeling relations between contracts, used for example in verification, represented in the lower part of Figure 2.

The requirement φ that the component model has to satisfy is represented by the meta-class *SafetyProperty*. This meta-class is left unspecified at this point since different formalisms could be used to model a property, such as temporal logics, automata-based languages (observers), etc. In §3.3 we instantiate the meta-model in the OMEGA profile, which uses an observer for modeling a *SafetyProperty*.

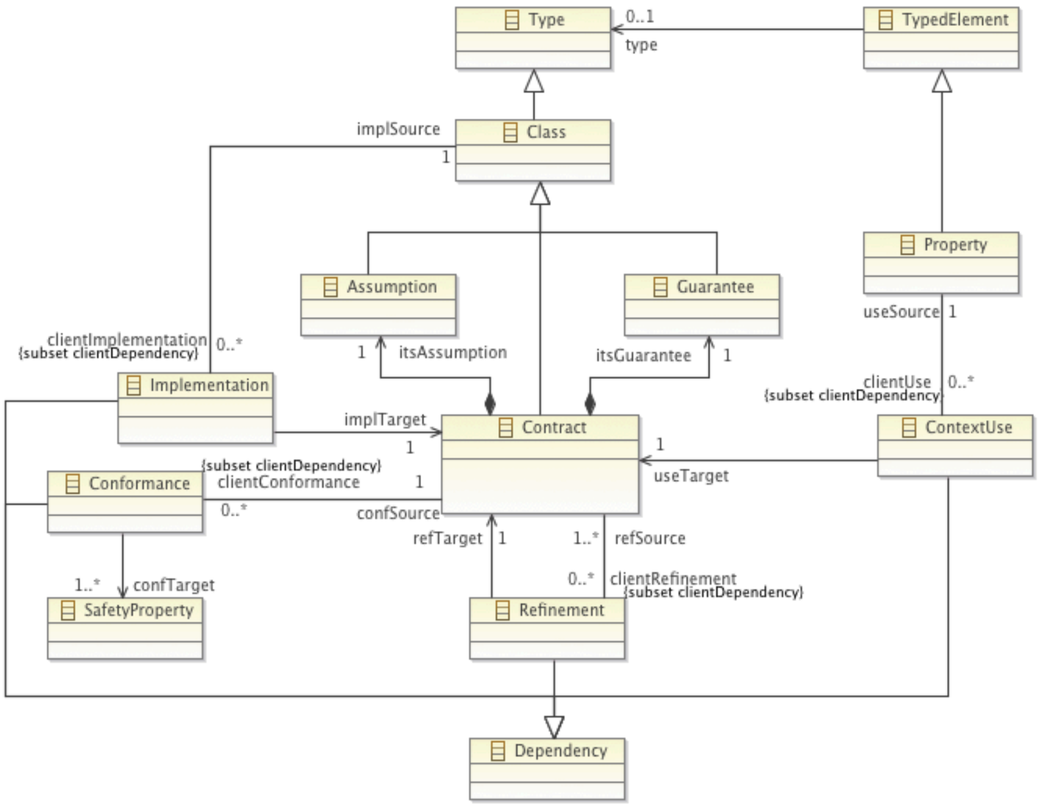

Figure 2: UML/SysML metamodel extended with Contract

The assumption/guarantee of a contract is modeled by the corresponding meta-class *Assumption/Guarantee* type of *Class*. So, both elements are modeled by a class that has a behavior expressed by one state machine and communicates through ports. The latter constraint imposes the following: an *Assumption/Guarantee* is not involved in associations, generalizations, realizations (except the interface realizations demanded by ports) and dependencies. However, the Assumption/Guarantee may define a composite sub-structure. Such an example is provided in the case study of §4.

Constraint 1 *An Assumption has no relations: associations, generalizations, realization and dependency are forbidden.*

```
context Assumption

-- Rule: An assumption has only properties with predefined types (i.e. an
    assumption is not involved in associations, aggregations and
    compositions)
def: assumptionHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
      name<>'Timer' and not a.isComposite)->size() = 0

def: assumptionPropertiesWellFormed : Boolean =
  self.assumptionHasNoPropertiesClassType

inv assumptionPropertiesWellFormed : self.assumptionPropertiesWellFormed

-- Rule: An assumption is not involved in any generalization relations (has
    no parents)
```

```
def: assumptionHasNoGenerals : Boolean =
  self.general->size() = 0

inv assumptionGeneralizationsWellFormed : self.assumptionHasNoGenerals

-- Rule: An assumption does not depend on any model element
def: assumptionHasNoDependencies : Boolean =
  self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization))->
      size() = 0

inv assumptionDependenciesWellFormed : self.assumptionHasNoDependencies
```

Constraint 2 *A Guarantee has no relations: associations, generalizations, realization and dependency are forbidden.*

```
context Guarantee

-- Rule: A guarantee has only properties with predefined types (i.e. an
    assumption is not involved in associations, aggregations and
    compositions)
def: guaranteeHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
      name<>'Timer' and not a.isComposite)->size() = 0

def: guaranteePropertiesWellFormed : Boolean =
  self.guaranteeHasNoPropertiesClassType

inv guaranteePropertiesWellFormed : self.guaranteePropertiesWellFormed

-- Rule: A guarantee is not involved in any generalization relations (has no
     parents)
def: guaranteeHasNoGenerals : Boolean =
  self.general->size() = 0

inv guaranteeGeneralizationsWellFormed : self.guaranteeHasNoGenerals

-- Rule: A guarantee does not depend on any model element
def: guaranteeHasNoDependencies : Boolean =
  self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization))->
      size() = 0

inv guaranteeDependenciesWellFormed : self.guaranteeHasNoDependencies
```

A contract is represented by the meta-class *Contract* as a composite structure, containing exactly one *assumption* and one *guarantee* (i.e. any other properties are forbidden). The only relations a contract may be involved in are those that represent the verification relations used in our theory, as described below.

Constraint 3 *A Contract does not own any properties (except the composite assumption and guarantee), any operations or signal receptions and any state machines. A Contract is not involved in other relations besides Implementation, Refinement and Conformance (i.e. associations, generalizations and aggregations/compositions are forbidden).*

```
context Contract
```

```
-- Rule: A contract has no properties besides one part typed assumption and
    one part typed guarantee (i.e. no properties with predefined type and no
     properties from associations, aggregations or compositions)
def: contractHasNoPropertiesPredefinedType : Boolean =
  self.ownedAttribute->select(a | not a.type.oclIsTypeOf(uml::Class))->size
      () = 0
def: contractHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and
  ((not a.type.oclAsType(uml::Class).isAssumption) or (a.type.oclAsType(uml
      ::Class).isAssumption and not a.isComposite)) and
  ((not a.type.oclAsType(uml::Class).isGuarantee) or (a.type.oclAsType(uml::
      Class).isGuarantee and not a.isComposite)))->size() = 0
def: contractPropertiesWellFormed : Boolean =
  self.contractHasNoPropertiesPredefinedType and self.
      contractHasNoPropertiesClassType

inv contractPropertiesWellFormed: self.contractPropertiesWellFormed

-- Rule: A contract has no operations
def: contractHasNoOperations : Boolean =
  self.ownedOperation->size() = 0

inv contractOperationsWellFormed : self.contractHasNoOperations

-- Rule: A contract has no statemachine
def : contractHasNoStateMachine : Boolean =
  self.ownedBehavior->size() = 0

inv contractStateMachineWellFormed : self.contractHasNoStateMachine

-- Rule: A contract is not involved in any generalization relations (has no
    parents)
def : contractHasNoGenerals : Boolean =
  self.general->size() = 0

inv contractGeneralizationsWellFormed : self.contractHasNoGenerals
```

If a contract serves in the conformance step in the methodology depicted in Fig. 1, this is modeled using a *Conformance* relation (a kind of *Dependency*) between the *Contract* and the corresponding *SafetyProperty*. One can use the same contract for several safety properties.

The dominance relation is represented by the meta-class *Refinement* type of *Dependency*. One contract is *refined* by a set of contracts. Note that this is possible since UML/SysML defines *Dependency* from n *clients* to n *suppliers*. To ensure that no cycles may be modeled, the following constraint is imposed: the target of a *Refinement* is not a member of the source set.

Constraint 4 *The target of a Refinement relation is not a member of the source set.*

```
context Refinement

-- Rule: The target of a Refinement relation is not a member of the source
    set
def: refinementTargetIsNotSource : Boolean =
  not self.refSource->includes(self.refTarget)

inv refinementTargetWellFormed : self.refinementTargetIsNotSource
```

Finally, the relation between a component and a contract that it must satisfy is represented by two relations: one at the level of the type of the component and one at the level of the instance (the part which participates in a composite structure where the contract is relevant and which is modeled by the meta-class *Property*). On the level of the type, an *Implementation* relation (a kind of *Dependency*) between a class and a contract models the fact that the class satisfies the contract. One class can satisfy several distinct contracts. On the level of instances, a *ContextUse* relation (also a kind of *Dependency*) between a *Property*, which is part of a composite structure, and a *Contract* models the fact that the contract is used for verification within the context of that composite structure. A *Property* may *use* a contract if and only if its class *implements* that contract.

Constraint 5 *A Contract can be used by a Property if and only if the property's type implements the contract.*

```
context ContextUse

-- Rule: A contract can be used in a proof tree if and only if the type of
    the property using it implements the contract
def: getImplementationsForTarget : Set(Class) = self.useTarget.oclAsType(uml
    ::Classifier).getModel().getDependencies->select(d | d.isImplementation
    and d.implTarget = useTarget).implSource.oclAsType(uml::Class)->asSet()

def: canContractBeUsed : Boolean = self.getImplementationsForTarget->
    includes(self.useSource.type.oclAsType(uml::Class))

def: contractUseWellFormed : Boolean = self.canContractBeUsed

inv contractUseWellFormed : self.contractUseWellFormed
```

3.2 Well-formedness rules for verifiable contracts

In order to be able to apply the contract-based verification theory from [9] we need to make sure that the hypotheses and constraints imposed by the formal framework are satisfied by the system model. In the following we formalize these constraints at the meta-model level by a set of well-formedness rules.

Within the formal framework, a contract is modeled by a pair (A, G) of TIOA such that the set of inputs/outputs of G is a subset of the set of inputs/outputs of the component implementing the contract and the composition of A and G is a closed system. To ensure this, the set of ports of a *Guarantee* must correspond to a subset of the set of ports of the component for which the guarantee is defined. The correspondence is based on the port name, and the port type and direction must coincide. We consider that when a port is present in the guarantee, all the corresponding signal receptions defined by the port type are handled in the guarantee.

Rule 1 *Given an Implementation, the set of ports of the contracts' Guarantee is included in the set of ports of the component source.*

```
context Port

def: isIdenticalTo(p:Port) : Boolean = self.name = p.name and self.provided
    = p.provided and self.required = p.required

context Implementation

-- Rule: The set of ports of the Guarantee is a subset or equal to the set
    of ports of the Part implementing it
-- Two ports are identical if they have the same name, direction and type
```

```
def: guaranteePortsSubsetPartPorts : Boolean =
  self.implTarget.itsGuarantee.ownedPort->forAll(p1 | self.implSource.
      ownedPort->select(p2 | p2.isIdenticalTo(p1))->size() = 1)

def: guaranteePortsWellFormed : Boolean =
  self.guaranteePortsSubsetPartPorts

inv implementationGuaranteePortsWellFormed : self.guaranteePortsWellFormed
```

For the composition between an *Assumption* and a *Guarantee* to be closed, every port of the *Guarantee* must have a corresponding conjugated port on the side of the *Assumption*, with the same type and reversed direction.

Rule 2 *Given a Contract, the Assumption and the Guarantee define a closed system: all ports of each entity have a correspondent within the ports of the other entity.*

```
context Port

def: isConjugated(p:Port) : Boolean = self.provided = p.required and self.
    required = p.provided

context Contract

-- Rule: The assumption and guarantee of a contract define a closed system
    with respect to ports
def: assumptionPortsSubsetGuaranteePorts : Boolean =
  self.itsAssumption.ownedPort->forAll(p1 | self.itsGuarantee.ownedPort->
      select(p2| p1.isConjugated(p2))->size() = 1)
def: guaranteePortsSubsetAssumptionPorts : Boolean =
  self.itsGuarantee.ownedPort->forAll(p1 | self.itsAssumption.ownedPort->
      select(p2 | p1.isConjugated(p2))->size() = 1)

def: contractAGPortsWellFormed : Boolean =
  self.assumptionPortsSubsetGuaranteePorts and self.
      guaranteePortsSubsetAssumptionPorts

inv contractClosedSystem : self.contractAGPortsWellFormed
```

The dominance relation is also subject to refinement of provided/required requests. This rule is also expressed with respect to ports: a port of the guarantee which is the target of the refinement must be a matched (by name and type) by a port of one of the refining guarantees, and must not be matched by a corresponding conjugated port (i.e. with reversed directionnality) of another of the refining guarantees.

Rule 3 *Given a Refinement, the set of ports of the target's guarantee is a subset or equal to the union of not matched ports of its set of sources.*

```
context Port

def: isSubtypeConjugated(p:Port) : Boolean = self.direction <> p.direction
    and p.interfaces->includesAll(self.interfaces)

context Class

def: getPart : Set(Property) =  self.ownedAttribute->select(a | a.type.
    oclIsTypeOf(uml::Class) and a.isComposite)
def: getUsedContractsOfParts(target:Class) : Set(Class) =
  self.getPart->iterate(p:Property; res:Set(Class)=Set{} | res->union(p.
      clientDependency->select(d1:Dependency | d1.isUsage and d1.useTarget.
      clientDependency->select(d2:Dependency | d2.isRefinement and d2.
      refTarget = target)->size() > 0).useTarget.oclAsType(uml::Class)))
def: getPortsFromUsedContractsOfParts(target:Class) : Set(Port) =
  self.getUsedContractsOfParts(target)->iterate(c:Class; res:Set(Port)=Set{}
       | res->union(c.itsGuarantee.ownedPort))

context Dependency

def: getPartsUsingRefinementTarget : Set(Property) =
  self.refTarget.oclAsType(uml::Classifier).getModel().getDependencies->
      select(d:Dependency | d.isUsage and d.useTarget = self.refTarget).
      useSource->asSet()
def: getRequiredPorts(sp:Set(Port)) : Set(Port) = sp->select(p:Port | p.
    direction = 'required')
def: getProvidedPorts(sp:Set(Port)) : Set(Port) = sp->select(p:Port | p.
    direction = 'provided')

context Refinement

-- Rule: The set of Ports of the Guarantee of the Source is a subset or
    equal to the (union of) sets of ports of the Gurantees of the Target
def: nonMatchedPorts(sp:Set(Port)) : Set(Port) =
  let spr: Set(Port) = self.getRequiredPorts(sp),
       spp: Set(Port) = self.getProvidedPorts(sp) in
  spr->iterate(p1:Port; res:Set(Port)=Set{} | if not spp->exists(p2 | p1.
      isSubtypeConjugated(p2)) then res->union(p1->asSet()) else res endif)
      ->union(
  spp->iterate(p1:Port; res:Set(Port)=Set{} | if not spr->exists(p2 | p1.
      isSubtypeConjugated(p2)) then res->union(p1->asSet()) else res endif))

def: refTargetPortsSubsetSourcesPorts(p:Property) : Boolean =
  let r:Class = self.refTarget,
       sp:Set(Port) = self.nonMatchedPorts(p.type.oclAsType(uml::Class).
           getPortsFromUsedContractsOfParts(r)) in
  r.itsGuarantee.ownedPort->forAll(p1 | sp->select(p2 | p1.isIdenticalTo(p2)
      )->size() = 1)

def: targetGuaranteePortsWellFormed : Boolean =
  self.getPartsUsingRefinementTarget->forAll(p:Property | self.
      refTargetPortsSubsetSourcesPorts(p))

inv refinementTargetGuaranteePortsWellFormed : self.
    targetGuaranteePortsWellFormed
```

One of the advantages of contracts is their reusability: different components can implement the same contract and one contract can be subject to several refinement relations. However, the latter case may introduce inconsistencies when verifying a property. Assume a part K that uses a contract C having two possible refinements $\{C_1, C_2\}$ and $\{C_3, C_4\}$. This introduces a non-determinism in the computed proof tree and complicates the verification steps. In order to avoid such situations, we demand for a contract to be refined only once within the context implementing it (i.e. each part whose type implements the contract can refine it only once with respect to the part's parts).

Rule 4 *Within a context, a contract can only be refined once.*

```
context Class
```

```
def: refinementUniqueWithinContext(target:Class) : Boolean =
  self.getPart->forAll(p:Property | p.clientDependency->select(d1:Dependency
       | d1.isUsage and d1.useTarget.clientDependency->select(d2:Dependency
      | d2.isRefinement and d2.refTarget = target)->size() > 0)->size() = 1)

context Refinement

-- Rule: Within a context (i.e. the part implementing the contract), the
    target contract has only one possible refinement
def: refinementUniqueWithinContext : Boolean =
  self.getPartsUsingRefinementTarget->forAll(p:Property | p.type.oclAsType(
      uml::Class).refinementUniqueWithinContext(refTarget))

inv refinementUniqueWithinContext : self.refinementUniqueWithinContext
```

The theory from [9] also induces some constraints on the state machines of assumptions and guarantees. In particular, the behavior of a guarantee or assumption should not impose constraints on time progress. This is realized on the UML/SysML level ensuring that all transitions in these state machines are stereotyped ≪*lazy*≫, and that there is at most one output action on any transition.

Furthermore, for a model with contracts to be used in compositional verification according to the methodology described in §2, the model must describe a unique and complete proof tree: all implemented contracts are used within a context and for all *SafetyProperty* there is a contract conforming to it.

Rule 5 *Within a model, all implemented Contracts must be used by a part and for all SafetyProperty there is a contract conforming to it.*

```
context Contract

-- Rule: All contracts must be implemented
def: isImplemented : Boolean =
        self.oclAsType(uml::Classifier).getModel().getDependencies->select(d
             | d.isImplementation and d.implTarget = self)->size() > 0
def: isUsed : Boolean =
        self.oclAsType(uml::Classifier).getModel().getDependencies->select(d
             | d.isUsage and d.useTarget = self)->size() > 0
def: contractIsImplemented : Boolean =
  if self.isContract
    then if self.isImplemented then self.isUsed else true endif
  else
    true
  endif

inv contractIsImplemented : self.contractIsImplemented

context SafetyProperty

-- Rule: All SafetyProperties have a contract conforming to it
def: isVerified : Boolean =
  self.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.
      isConformance and d.confTarget->includes(self))->size() > 0

def: spIsVerified : Boolean = self.isVerified

inv safetyPropertyIsVerified : self.spIsVerified
```

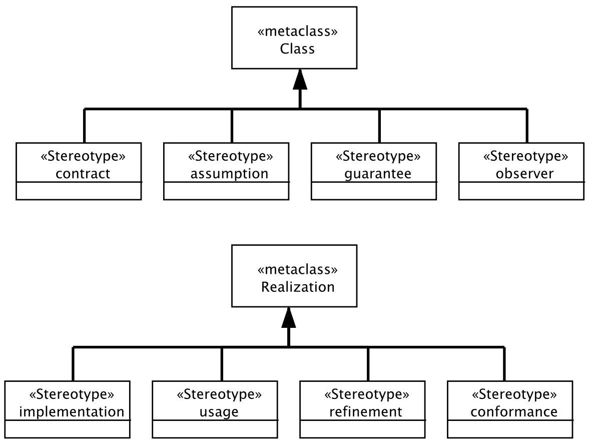

Figure 3: Contract stereotypes for the OMEGA2 Profile

3.3 Instantiating the meta-model in the OMEGA profile

In order to use contracts in a standard UML/SysML model, one needs to capture the information from the meta-model described in the previous section in the form of standard extensions, namely using stereotypes. Since all the new concepts introduced in the meta-model inherit from an existing meta-class (either *Class* or *Dependency*), we choose to represent them as stereotypes of these base meta-classes. Fig. 3 presents the *Class* stereotypes $\ll$*contract*$\gg$, $\ll$*assumption*$\gg$, $\ll$*guarantee*$\gg$, and $\ll$*observer*$\gg$ (which already exists in OMEGA and is reused for representing *SafetyProperty*). For contract relations, the stereotypes of *Dependency* that correspond to the meta-model elements are $\ll$*implementation*$\gg$, $\ll$*usage*$\gg$, $\ll$*refinement*$\gg$ and $\ll$*conformance*$\gg$.

From the semantic point of view, *Contracts* are not handled in the same way as usual classes/blocks: they are of course not considered executable elements of the system. Contracts are only used by the verification tools to check the validity of the conformance, dominance and satisfaction relations.

4 The ATV Solar Generation System case study

The concepts and the reasoning method described previously have been applied on a case study, an industrial-grade system model of a subsystem of the Automated Transfer Vehicle (ATV). The ATV, developed by Astrium Space Transportation for the European Space Agency, is a spacecraft put into orbit by the European heavy launcher Ariane-5 with the aim of supplying the International Space Station. This case study consists of the Solar Wing Generation System (SGS) [8] responsible for the deployment and management of the solar wings of the vehicle. The SysML model used in the following, provided by Astrium Space Transportation, was obtained by reverse engineering the actual SGS system for the purpose of this study.

The SGS system model illustrated in Fig. 4 summarizes the three main components involved in the case study: the mission and vehicle management (*MVM*) part that initiates SGS wing deployment, the *SOFTWARE* part of the *SGS* that based on requests received from the *MVM* executes the corresponding automated procedures and the *HARDWARE* part that models the four physical wings. The communication between components is realized via asynchronous signals transported through ports and connectors. Due to the large number of ports (661) and connectors (504), Fig. 4 presents a simplified architectural view of SGS and only shows a link between two parts where several connectors and ports are involved in the actual model.

Under the hypothesis that at most one hardware failure may occur during a run, which is

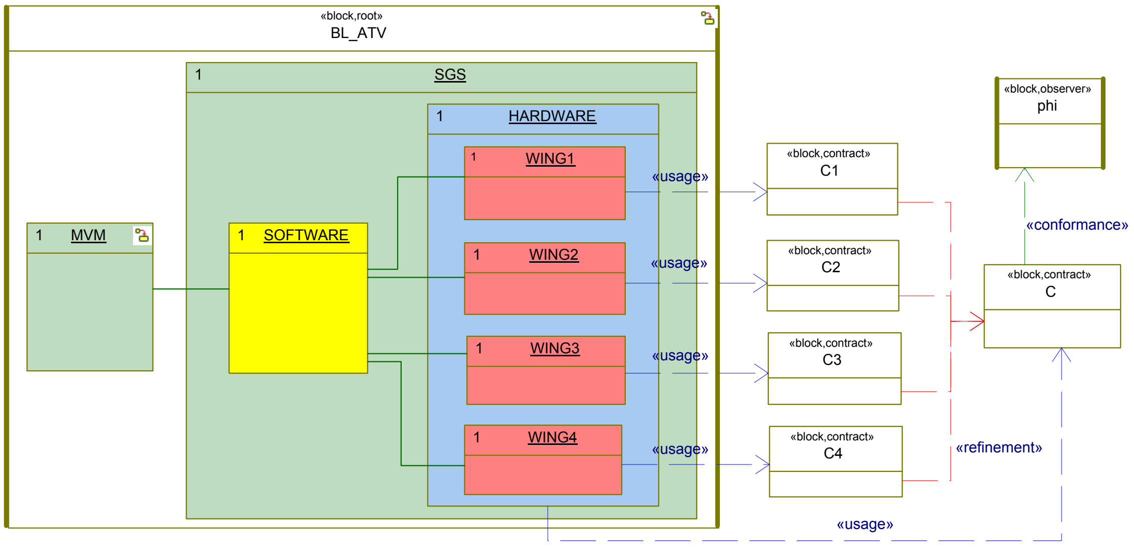

Figure 4: Architecture of the SGS system including contracts (simplified view).

embedded in the *HARDWARE* model, the main goal of the case study is to verify the following property φ:

Property φ: After 10 minutes from system start-up, all four wings are deployed.

Due to the size and the complexity of the model, applying model-checking directly leads to combinatorial explosion and the verification of φ does not finish. We explain in the following how the property φ was verified using the contract-based reasoning methodology. We start by modeling the property φ. This implies identifying what the observer corresponding to the safety property φ must monitor. In our case, the block *phi* must observe the answer that each wing provides with respect to its status (deployed or not deployed) when interrogated by the software. So, the property φ expressed with respect to wing behavior must be satisfied by the *HARDWARE* block instance that contains them. With regard to Fig. 1, the *HARDWARE* is the subsystem S and *WINGi*, $i = \overline{1,4}$, are the components K_i. The environment of the subsystem is given by the parts with which it communicates: bidirectional communication is directly established between *SOFTWARE* and *HARDWARE*, while *SOFTWARE* depends on the behavior of *MVM*. Thus, the environment E of Fig. 1 is represented here by the composition of *MVM* and *SOFTWARE*.

Next, we provide a contract $C = (A, G)$ such that it conforms to φ. In order to comply to the contract methodology, C is implemented by *HARDWARE*'s type and it is used by this part within the proof tree. We use as assumption A the concrete environment of *HARDWARE*, the composition between *MVM* and *SOFTWARE* itself, which thus satisfies by construction the mirror contract. Keeping this composition as assumption is not problematic since its state space has a manageable size. As guarantee G we use the following abstraction derived (manually) from the individual behavior of wings: for each wing status interrogation, the target wing answers either as not deployed for at most 400 seconds or as deployed after at least 130 seconds. In order to ensure Rule 2, since *MVM*$||$*SOFTWARE* sends all possible requests to *HARWARE*, we equip G with all ports defined by *HARDWARE* and we enrich the behavior of G to ignore all other requests. Rule 1 is also satisfied because no refinement of requests is performed. Rule 2 is satisfied by all C_i.

The third step consists in modeling a set of contracts $\{C_1, C_2, C_3, C_4\}$ that refine C and proving that each contract $C_i = (A_i, G_i)$ is implemented by *WINGi*'s type, $i = \overline{1,4}$. The environment for *WINGi* is given by the environment of the subsystem *HARDWARE* and all *WINGj*, $j \neq i$. We use the following abstraction *WAj* for *WINGj*: the wing is either not deployed for at most 400 seconds or deployed from at least 130 seconds while all other received requests are consumed. The assumption A_i is the parallel composition of *MVM*, *SOFTWARE* and *WAj*, $j \neq i$. The guarantee G_i is the projection of G on *WINGi*. Rule 3 is satisfied since the ports of *HARDWARE* with respect to *WINGi* are identical to the ports of *WINGi*. Moreover, this condition also satisfies Rule 1 for each C_i, since refinement of requests is not considered for this case study.

After this step the proof obligation tree is complete. The verification involves 10 intermediate steps: 4 for verifying that each wing satisfies its contract, 5 sufficient conditions for dominance between $\{C_1, C_2, C_3, C_4\}$ and C and one for proving that $A \parallel G \preceq \varphi$. Each verification step is performed by the OMEGA-IFx model checker in a few hours (precise data is available in [9]). The overall effort of the building the contracts and performing the verification steps was of about 5 person*days. This complexity is also due to the fact that some steps have been manually performed like guarantees transformed into observers (timed trace inclusion is verified using observers) or connecting assumptions to components via links. The automation of these steps is currently under development. For further details on contract-based verification of the SGS case study, the reader is referred to [9].

5 Related work

Modeling and verifying contracts for components is a long line of research, whose origins date back to Hoare logic [10]. Syntactical and behavioral contracts, as classified in [2], have been explored for specifying composition constraints and pre/post conditions for operations and also for modeling transformation of models and execution semantics. Contracts as a language construct have emerged with the Eiffel programming language [14] and have since been explored for various programming and specification models. In this section we concentrate on work aiming to introduce contracts in high-level modeling languages. For a discussion of more theoretical works on contracts and contract-based verification the reader is referred to [9].

Weis et al. [19] propose to model a contract for a component in UML by an interface and to specify its role: it can be either a *required* contract on which the component depends or a *provided* contract that is realized by the component. Syntactically, this representation of contracts is similar to ours: we also make the distinction between the required behavior of the environment and the provided behavior of the component by taking into account the assumption over the environment. However, our contracts are richer since they model a behavior that can be used for component validation, while the contracts of [19] can be used only for composability checking during the development phases.

The Kmelia component model [13, 1], based on the previous described work, provides means to verify the functional correctness of behavioral contracts for services: the behavior of an operation is modeled as a Labeled Transition System and formal verification can be realized within different tools via model transformation. Their meta-model defines for a contract the source implementing it as an aggregated element (operation or interface) and models explicitly the contract satisfaction results. But, this formalism does not describe how the order in which services are called by and from a component can be verified, order that can be seen similar to our state machines. Furthermore, it does not provide a connection to high-level modeling language as UML/SysML.

Contracts modeled as pre/post conditions are used in [6] for the verification of model transformation: the assumption is represented by an OCL constraint on the source model and the guarantee is an OCL constraint on the target model. Besides the different purpose, the main difference with respect to the syntax of contracts is that, while our approach considers contracts only for components, [6] models contracts for all model elements. The same contracts are used in [5] to model the execution semantics of UML elements which is seen as a case of model transformation.

To the best of our knowledge, this study is the first to consider behavioral contracts at the component level in UML/SysML and to provide verification relations for property satisfaction by contract-based reasoning. The meta-model we propose is generic enough to represent all the other meta-models previously described, with except of the verification results extension that is based on the dynamical execution of the model.

6 Conclusions

Based on a theory of contracts and on a methodological approach for reasoning with contracts introduced in previous work [9], we have proposed an extension of UML/SysML allowing to model contracts and use them for compositional verification of requirements. The extension is defined as a meta-model, enriched with constraints and well-formedness rules to make contracts verifiable. We have instantiated the extension within the OMEGA UML/SysML Profile to make it usable with standard model editors. The verification method is supported by the OMEGA-IFx toolset and the approach was validated on an industrial-grade system model.

Although an automatic model transformation from OMEGA system models to the input language of the IFx Toolset is already available, some of the steps for generating the intermediate contract-based verification models remain manual. Future work consists in automating all the intermediate model generation steps and in adding functionality for managing the proof obligations and results and for enforcing the rigorous verification methodology described in §2.3.

References

[1] Pascal André, Ardourel; Gilles, and Mohamed Messabihi. Vérification de contrats logiciels a l'aide de transformations de modeles. In *7èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM 2011)*, 2011.

[2] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[3] Sebastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Information and Computation*, 163, 2000.

[4] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 131–132. Springer Berlin / Heidelberg, 2004.

[5] Eric Cariou, Cyril Ballagny, Alexandre Feugas, and Franck Barbier. Contracts for model execution verification. In *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, pages 3–18, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. Ocl contracts for the verification of model transformations. *ECEASST*, 24, 2009.

[7] Eric Conquet, Francois-Xavier Dormoy, Iulia Dragomir, Susanne Graf, David Lesens, Piotr Nienaltowski, and Iulian Ober. Formal Model Driven Engineering for Space Onboard Software. In *Proceedings of Embedded Real Time Software and Systems (ERTS2), Toulouse*. SAE, 2012.

[8] Iulia Dragomir, Iulian Ober, and David Lesens. A case study in formal system engineering with SysML. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th IEEE International Conference on*, july 2012.

[9] Iulia Dragomir, Iulian Ober, and Christian Percebois. Safety Contracts for Timed Reactive Components in SysML. Technical report, IRIT, june 2013. Submitted for publication. Available at `http://www.irit.fr/~Iulian.Ober/docs/TR-Contracts.pdf`.

[10] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata - Second Edition*. Morgan & Claypool Publishers, 2010.

[12] RTCA Inc. Software Considerations in Airborne Systems and Equipment Certification. Document RTCA/DO-178C, 2011.

[13] Mohamed Messabihi, Pascal André, and Christian Attiogbé. Multilevel Contracts for Trusted Components. In Javier Cámara, Carlos Canal, and Gwen Salaün, editors, *Proceedings International Workshop on Component and Service Interoperability*, volume 37 of *EPTCS*, pages 71–85, 2010.

[14] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, October 1992.

[15] Iulian Ober and Iulia Dragomir. Omega2: A new version of the profile and the tools. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 373–378. IEEE Computer Society, 2010.

[16] Object Management Group. Systems Modelling Language (SysML) v1.1, 2008.

[17] Object Management Group. Unified Modelling Language (UML) v2.2, 2009.

[18] Sophie Quinton. *Design, vérification et implémentation de systèmes à composants*. PhD thesis, Université de Grenoble, 2011.

[19] Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau. A uml meta-model for contract aware components. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML'01, pages 442–456, London, UK, UK, 2001. Springer-Verlag.

A Contract meta-model: Formalization of constraints (in UML 2.3)

```
-- HELPER FUNCTIONS

context Namespace

-- Computes recursively the set of dependencies contained in a namespace
def: getDependenciesRec : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

context Model

-- Computes recursively the set of dependencies of a model
def: getDependencies : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

context Class

-- Verifies if the class is a contract
def: isContract : Boolean = self.getAppliedStereotypes()->select(name='
    contract')->size()<>0

-- Verifies if the class is an assumption
def: isAssumption : Boolean = self.getAppliedStereotypes()->select(name='
    assumption')->size()<>0

-- Verifies if the class is a guarantee
def: isGuarantee : Boolean = self.getAppliedStereotypes()->select(name='
    guarantee')->size()<>0

context Dependency

-- Verifies if the dependency is an implementation
def: isImplementation : Boolean = self.getAppliedStereotypes()->select(name=
    'implementation')->size()<>0

-- Verifies if the dependency is a usage
def: isUsage : Boolean = self.getAppliedStereotypes()->select(name='usage')
    ->size()<>0

-- Verifies if the dependency is a refinement
def: isRefinement : Boolean = self.getAppliedStereotypes()->select(name='
    refinement')->size()<>0

-- Verifies if the dependency is a conformance
def: isConformance : Boolean = self.getAppliedStereotypes()->select(name='
    conformance')->size()<>0



-- CONSTRAINT 1
```

```
context Class

-- Verifies if the assumption does not model any class type properties not
    composite
def: assumptionHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
      name<>'Timer' and not a.isComposite)->size() = 0

-- Verifies if the assumption has properties well formed
def: assumptionPropertiesWellFormed : Boolean =
  if self.isAssumption
    then self.assumptionHasNoPropertiesClassType
  else
    true
  endif

-- Verifies if the assumption has no generalizations
def: assumptionHasNoGenerals : Boolean =
  if self.isAssumption
    then self.general->size() = 0
  else
    true
  endif

-- Verifies if the assumption has no dependencies (except interface
    realization)
def: assumptionHasNoDependencies : Boolean =
  if self.isAssumption
    then self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization
        ))->size() = 0
  else
    true
  endif

-- Constraint 1
inv assumptionWellFormed : self.assumptionPropertiesWellFormed and self.
    assumptionHasNoGenerals and self.assumptionHasNoDependencies



-- CONSTRAINT 2

context Class

-- Verifies if the guarantee does not model any class type properties not
    composite
def: guaranteeHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
      name<>'Timer' and not a.isComposite)->size() = 0

-- Verifies if the guarantees' properties are well-formed
def: guaranteePropertiesWellFormed : Boolean =
  if self.isGuarantee
    then self.guaranteeHasNoPropertiesClassType
  else
    true
  endif

-- Verifies if the guarantee has no generalizations
def: guaranteeHasNoGenerals : Boolean =
  if self.isGuarantee
    then self.general->size() = 0
  else
    true
  endif

-- Verifies if the guarantee has no dependencies (except interface
    realization)
def: guaranteeHasNoDependencies : Boolean =
  if self.isGuarantee
    then self.clientDependency->reject(oclIsTypeOf(uml::InterfaceRealization
        ))->size() = 0
  else
    true
  endif

-- Constraint 2
inv guaranteeDependenciesWellFormed : self.guaranteePropertiesWellFormed and
     self.guaranteeHasNoGenerals and self.guaranteeHasNoDependencies



-- CONSTRAINT 3

context Class

-- Verifies if the contract does not model any predefined type properties
def: contractHasNoPropertiesPredefinedType : Boolean =
  self.ownedAttribute->select(a | not a.type.oclIsTypeOf(uml::Class))->size
      () = 0

-- Verifies if the contract does not model any class type properties beside
    assumption and guarantee
def: contractHasNoPropertiesClassType : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and
    ((not a.type.oclAsType(uml::Class).isAssumption) or (a.type.oclAsType(
        uml::Class).isAssumption and not a.isComposite)) and
    ((not a.type.oclAsType(uml::Class).isGuarantee) or (a.type.oclAsType(uml
        ::Class).isGuarantee and not a.isComposite)))->size() = 0

-- Verifies if the contract has properties well formed
def: contractPropertiesWellFormed : Boolean =
  if self.isContract
    then self.contractHasNoPropertiesPredefinedType and self.
        contractHasNoPropertiesClassType
  else
    true
  endif

-- Verifies if the contract does not own any operations
def: contractHasNoOperations : Boolean =
  if self.isContract
    then self.ownedOperation->size() = 0
  else
    true
  endif

-- Verifies if the contract does not own any state machine
def : contractHasNoStateMachine : Boolean =
  if self.isContract
    then self.ownedBehavior->size() = 0
  else
    true
  endif

-- Verifies if the contract has no generalizations
def : contractHasNoGenerals : Boolean =
  if self.isContract
    then self.general->size() = 0
  else
    true
  endif

-- Constraint 3
inv contractWellFormed : self. contractPropertiesWellFormed and self.
    contractHasNoOperations and self.contractHasNoStateMachine and self.
    contractHasNoGenerals



-- CONSTRAINT 4

context Dependency

-- Verifies if the target of a refinement is not a member of the source set
def: refinementTargetIsNotSource : Boolean =
  if self.isRefinement
    then not self.client->includes(self.supplier)
  else
    true
  endif

-- Constraint 4
inv refinementTargetWellFormed : self.refinementTargetIsNotSource



-- CONSTRAINT 5

context Dependency

-- Computes the set of classes that implement the target contract of a usage
def: getImplementationsForTarget : Set(Class) =
  let ut:Class = self.supplier->asOrderedSet()->at(1).oclAsType(uml::Class)
      in
  ut.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.
      isImplementation and d.supplier->asOrderedSet()->at(1).oclAsType(uml::
      Class) = ut).client.oclAsType(uml::Class)->asSet()

-- Verifies if the type of the source of a usage is a class implementing the
     contract
def: canContractBeUsed : Boolean = self.getImplementationsForTarget->
    includes(self.client->asOrderedSet()->at(1).oclAsType(uml::Property).
    type.oclAsType(uml::Class))

-- Verifies if the source of a usage can use the contract
def: contractUseWellFormed : Boolean =
  if self.isUsage
    then self.canContractBeUsed
  else
    true
  endif
```

```
-- Constraint 5
inv contractUseWellFormed : self.contractUseWellFormed
```

B Making contracts verifiable: Formalization of well-formedness rules (in UML 2.3)

```
-- HELPER FUNCTIONS

context Namespace

-- Computes recursively the set of dependencies contained in a namespace
def: getDependenciesRec : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

-- Computes recursively the class owning the namespace
def: getOwningClass : Class =
  if self.oclIsTypeOf(uml::Class)
    then self.oclAsType(uml::Class)
  else
    self.namespace.getOwningClass
  endif

context Model

-- Computes recursively the set of dependencies of a model
def: getDependencies : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

context Classifier

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

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

context Class

-- Verifies if the class is a contract
def: isContract : Boolean = self.getAppliedStereotypes()->select(name='
    contract')->size()<>0

-- Verifies if the class is an assumption
def: isAssumption : Boolean = self.getAppliedStereotypes()->select(name='
    assumption')->size()<>0

-- Computes the assumption of a contract
def: itsAssumption : Class = self.ownedAttribute->select(a | a.type.
    oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).isAssumption)->
    at(1).type.oclAsType(uml::Class)

-- Verifies if the class is a guarantee
def: isGuarantee : Boolean = self.getAppliedStereotypes()->select(name='
    guarantee')->size()<>0

-- Computes the guarantee of a contract
def: itsGuarantee : Class = self.ownedAttribute->select(a | a.type.
    oclIsKindOf(uml::Class) and a.type.oclAsType(uml::Class).isGuarantee)->
    at(1).type.oclAsType(uml::Class)

-- Verifies if a contract is implemented by a class
def: isImplemented : Boolean = self.oclAsType(uml::Classifier).getModel().
    getDependencies->select(d | d.isImplementation and d.implTarget = self)
    ->size() > 0

-- Verifies if a contract is used within a context
def: isUsed : Boolean = self.oclAsType(uml::Classifier).getModel().
    getDependencies->select(d | d.isUsage and d.useTarget = self)->size() >
    0

-- Verifies if the class is an observer
def: isObserver : Boolean = self.getAppliedStereotypes()->select(name='
    observer')->size()<>0

-- Verifies if the class is a composite structure
def: isComposite : Boolean = self.ownedAttribute->select(a | a.type.
    oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).name <> 'Timer'
    )->size() <> 0

-- Computes the set of parts of a composite class
def: getPart : Set(Property) =  self.ownedAttribute->select(a | a.type.
    oclIsTypeOf(uml::Class) and a.isComposite)

context Port

-- Computes the set of interfaces that type a port
def: interfaces : Set(Interface) = self.provided->reject(isInterfaceGroup)

-- Computes the direction of a port
def: direction: String =
  if self.getValue(self.getAppliedStereotypes()->select(name='RhpPort')->
      asOrderedSet()->at(1),'isReversed').oclAsType(Boolean)
    then 'required'
  else
    'provided'
  endif

-- Verifies if two ports have the same name, direction and type
def: isIdenticalTo(p:Port) : Boolean = self.name = p.name and self.direction
     = p.direction and self.interfaces = p.interfaces

-- Verifies if two ports have different direction and the same type
def: isConjugated(p:Port) : Boolean = self.direction <> p.direction and self
    .interfaces = p.interfaces

-- Verifies if two ports have different directions and that the current is a
     subtype of the parameter port
def: isSubtypeConjugated(p:Port) : Boolean = self.direction <> p.direction
    and p.interfaces->includesAll(self.interfaces)

context Dependency

-- Verifies if the dependency is an implementation
def: isImplementation : Boolean = self.getAppliedStereotypes()->select(name=
    'implementation')->size()<>0

-- Computes the source of an implementation
def: implSource : Class = self.client->asOrderedSet()->at(1).oclAsType(uml::
    Class)

-- Computes the target of an implementation
def: implTarget : Class = self.supplier->asOrderedSet()->at(1).oclAsType(uml
    ::Class)

-- Verifies if the dependency is a usage
def: isUsage : Boolean = self.getAppliedStereotypes()->select(name='usage')
    ->size()<>0

-- Computes the source of a usage
def: useSource : Property = self.client->asOrderedSet()->at(1).oclAsType(uml
    ::Property)

-- Computes the target of a usage
def: useTarget : Class = self.supplier->asOrderedSet()->at(1).oclAsType(uml
    ::Class)

-- Verifies if the dependency is a refinement
def: isRefinement : Boolean = self.getAppliedStereotypes()->select(name='
    refinement')->size()<>0

-- Computes the source of a refinement
def: refSource : Bag(Class) = self.client.oclAsType(uml::Class)

-- Computes the target of a refinement
def: refTarget : Class = self.supplier->asOrderedSet()->at(1).oclAsType(uml
    ::Class)

-- Verifies if the dependency is a conformance
def: isConformance : Boolean = self.getAppliedStereotypes()->select(name='
    conformance')->size()<>0

-- Computes the source of a conformance
def: confSource : Class = self.client->asOrderedSet()->at(1).oclAsType(uml::
    Class)

-- Computes the targets of a conformance
def: confTarget : Set(Class) = self.supplier.oclAsType(uml::Class)->asSet()



-- RULE 1

context Dependency

-- Verifies if the ports of a guarantee are defined in the component
    implementing the contract
def: guaranteePortsSubsetPartPorts : Boolean =
  self.implTarget.itsGuarantee.ownedPort->forAll(p1 | self.implSource.
      ownedPort->select(p2 | p2.isIdenticalTo(p1))->size() = 1)

-- Verifies is an implementation relation is well formed with respect to
    request refinement
def: guaranteePortsWellFormed : Boolean =
  if self.isImplementation
    then self.guaranteePortsSubsetPartPorts
  else
    true
  endif

-- Rule 1
inv implementationGuaranteePortsWellFormed : self.guaranteePortsWellFormed



-- RULE 2

context Class

-- Verifies if the ports of the assumption have a correspondent within the
    ports of the guarantee
def: assumptionPortsSubsetGuaranteePorts : Boolean =
  self.itsAssumption.ownedPort->forAll(p1 | self.itsGuarantee.ownedPort->
      select(p2| p1.isConjugated(p2))->size() = 1)

-- Verifies if the ports of the guarantee have a correspondent within the
    ports of the assumption
def: guaranteePortsSubsetAssumptionPorts : Boolean =
  self.itsGuarantee.ownedPort->forAll(p1 | self.itsAssumption.ownedPort->
      select(p2 | p1.isConjugated(p2))->size() = 1)

-- Verifies if the contract defines a closed system
def: contractAGPortsWellFormed : Boolean =
  if self.isContract
    then self.assumptionPortsSubsetGuaranteePorts and self.
        guaranteePortsSubsetAssumptionPorts
  else
    true
  endif

-- Rule 2
inv contractClosedSystem : self.contractAGPortsWellFormed



-- HELPER FUNCTIONS

context Class

-- Computes the source set of contracts in a refinement relation within a
    given context
def: getUsedContractsOfParts(target:Class) : Set(Class) =
  self.getPart->iterate(p:Property; res:Set(Class)=Set{} | res->union(p.
      clientDependency->select(d1:Dependency | d1.isUsage and
    d1.useTarget.clientDependency->select(d2:Dependency | d2.isRefinement
        and d2.refTarget = target)->size() > 0).useTarget.oclAsType(uml::
        Class)))

-- Computes the set of ports of the guarantees of the source set of a
    refinement relation within a given context
def: getPortsFromUsedContractsOfParts(target:Class) : Set(Port) =
  self.getUsedContractsOfParts(target)->iterate(c:Class; res:Set(Port)=Set{}
       | res->union(c.itsGuarantee.ownedPort))

-- Verifies that a contract is refined only once within a context
def: refinementUniqueWithinContext(target:Class) : Boolean =
  self.getPart->forAll(p:Property | p.clientDependency->select(d1:Dependency
       | d1.isUsage and d1.useTarget.clientDependency->select(d2:Dependency
      | d2.isRefinement and d2.refTarget = target)->size() > 0)->size() = 1)

context Dependency

-- Computes the parts that implement the target of a refinement
def: getPartsUsingRefinementTarget : Set(Property) =
  self.refTarget.oclAsType(uml::Classifier).getModel().getDependencies->
      select(d:Dependency | d.isUsage and d.useTarget = self.refTarget).
      useSource->asSet()

-- Computes the set of required ports from a larger set
def: getRequiredPorts(sp:Set(Port)) : Set(Port) =
  sp->select(p:Port | p.direction = 'required')

-- Computes the set of provided ports from a larger set
def: getProvidedPorts(sp:Set(Port)) : Set(Port) =
  sp->select(p:Port | p.direction = 'provided')



-- RULE 3

context Dependency

-- Computes the set of ports that do not have a match within the set
def: nonMatchedPorts(sp:Set(Port)) : Set(Port) =
  let spr: Set(Port) = self.getRequiredPorts(sp),
      spp: Set(Port) = self.getProvidedPorts(sp) in
  spr->iterate(p1:Port; res:Set(Port)=Set{} | if not spp->exists(p2 | p1.
      isSubtypeConjugated(p2)) then res->union(p1->asSet()) else res endif)
      ->union(
  spp->iterate(p1:Port; res:Set(Port)=Set{} | if not spr->exists(p2 | p1.
      isSubtypeConjugated(p2)) then res->union(p1->asSet()) else res endif))

-- Verifies that the set of ports of the target are also defined in the
    union of non matched ports of the source of a refinement
def: refTargetPortsSubsetSourcesPorts(p:Property) : Boolean =
  let r:Class = self.refTarget,
      sp:Set(Port) = self.nonMatchedPorts(p.type.oclAsType(uml::Class).
          getPortsFromImplementedContractsOfParts(r)) in
  r.itsGuarantee.ownedPort->forAll(p1 | sp->select(p2 | p1.isIdenticalTo(p2)
      )->size() = 1)

-- Verifies the refinement of requests of a refinement
def: targetGuaranteePortsWellFormed : Boolean =
  if self.isRefinement
    then self.getPartsUsingRefinementTarget->forAll(p:Property | self.
        refTargetPortsSubsetSourcesPorts(p))
  else
    true
  endif

-- Rule 3
inv refinementTargetGuaranteePortsWellFormed : self.
    targetGuaranteePortsWellFormed



-- RULE 4
-- Verifies that a contract is refined only once in a context
def: refinementUniqueWithinContext : Boolean =
  if self.isRefinement
    then self.getPartsUsingRefinementTarget->forAll(p:Property | p.type.
        oclAsType(uml::Class).refinementUniqueWithinContext(refTarget))
  else
    true
  endif

-- Rule 4
inv refinementUniqueWithinContext : self.refinementUniqueWithinContext




-- RULE 5

context Class

-- Verifies if an implemented contract is used
def: contractIsImplemented : Boolean =
  if self.isContract
    then if self.isImplemented then self.isUsed else true endif
  else
    true
  endif

-- Rule 5
inv contractIsImplemented : self.contractIsImplemented


context Class

-- Verifies if a safety property is a target of a conformance relation
def: isVerified : Boolean =
  self.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.
      isConformance and d.confTarget->includes(self))->size() > 0

-- Verifies that a conformance has within the target the safety property
def: spIsVerified : Boolean =
  if self.isObserver
    then self.isVerified
  else
    true
  endif

-- Rule 5
inv safetyPropertyIsVerified : self.spIsVerified
```

C OMEGA Contract Profile: Formalization of well-formedness rules

```
-- HELPER FUNCTIONS

context Namespace

-- Computes recursively the set of dependencies contained in a namespace
def: getDependenciesRec : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

-- Computes recursively the class owning the namespace
def: getOwningClass : Class =
  if self.oclIsTypeOf(uml::Class)
    then self.oclAsType(uml::Class)
  else
    self.namespace.getOwningClass
  endif

context Model

-- Computes recursively the set of dependencies of a model
def: getDependencies : Set(Dependency) =
  self.member->iterate(m:NamedElement; res:Set(Dependency)=Set{} | if m.
      oclIsTypeOf(uml::Dependency) then res->union(m.oclAsType(uml::
      Dependency)->asSet())
    else if m.oclIsKindOf(uml::Namespace) then res->union(m.oclAsType(uml::
        Namespace).getDependenciesRec) else res->union(Set{}) endif endif)

context Classifier

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

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

context Class

-- Verifies if the class is a contract
def: isContract : Boolean = self.getAppliedStereotypes()->select(name='
    contract')->size()<>0

-- Verifies if the class is an assumption
def: isAssumption : Boolean = self.getAppliedStereotypes()->select(name='
    assumption')->size()<>0

-- Verifies if the class is a guarantee
def: isGuarantee : Boolean = self.getAppliedStereotypes()->select(name='
    guarantee')->size()<>0

-- Verifies if the class is an observer
def: isObserver : Boolean = self.getAppliedStereotypes()->select(name='
    observer')->size()<>0
```

```
-- Verifies is the class is an interfaceGroup
def: isPortType : Boolean = self.getAppliedStereotypes()->select(name='
    portType')->size()<>0

-- Verifies if the class is the root of the system
def: isRoot : Boolean = self.getAppliedStereotypes()->select(name='root')->
    size()<>0

-- Verifies if the class is a composite structure
def: isComposite : Boolean = self.ownedAttribute->select(a | a.type.
    oclIsTypeOf(uml::Class) and a.type.oclAsType(uml::Class).name <> 'Timer'
    )->size() <> 0

-- Computes the sequence of receptions defined in all ports of a class
def: portRequests : Sequence(Reception) = self.ownedAttribute->select(p | p.
    oclIsTypeOf(uml::Port)).oclAsType(uml::Port).interfaces.ownedReception

context Port

-- Computes the set of interfaces that type a port
def: interfaces : Set(Interface) = self.provided->reject(isInterfaceGroup)

-- Computes the set of receptions defined in the port's type
def: requests : Bag(Reception) = self.interfaces.ownedReception

context Trigger

-- Computes the class owning the trigger
def: getOwningClass : Class = self.namespace.getOwningClass

-- Computes the signal for which the trigger is defined
def: getSignalName : String = self.event.oclAsType(uml::SignalEvent).signal.
    name

context State

-- Verifies if the state is a  SendAction
def: isSendAction : Boolean = not self.getAppliedStereotype('
    RhapsodyStandardModel::RhapsodyProfile::RhpSendAction').oclIsUndefined()

-- Computes the signal sent via the SendAction
def: sendActionSignal : Signal = self.getValue(self.getAppliedStereotype('
    RhapsodyStandardModel::RhapsodyProfile::RhpSendAction'), 'event').
    oclAsType(uml::SignalEvent).signal

-- Computes the target port of the SendAction
def: sendActionTarget : Port = self.getValue(self.getAppliedStereotype('
    RhapsodyStandardModel::RhapsodyProfile::RhpSendAction'), 'target').
    oclAsType(uml::Port)

context Dependency

-- Verifies is the dependency is an implementation
def: isImplementation : Boolean = self.getAppliedStereotypes()->select(name=
    'implementation')->size()<>0

-- Verifies if the dependency is a usage
def: isUsage : Boolean = self.getAppliedStereotypes()->select(name='usage')
    ->size()<>0

-- Verifies if the dependency is a refinement
```

```
def: isRefinement : Boolean = self.getAppliedStereotypes()->select(name='
    refinement')->size()<>0

-- Verifies if the dependency is a conformance
def: isConformance : Boolean = self.getAppliedStereotypes()->select(name='
    conformance')->size()<>0



context Port

-- Verifies if the port is typed
def: portHasType : Boolean = self.interfaces->size() >= 1

-- Verifies that the port has requests to transfer
def: portTransfersRequests : Boolean = self.requests->size() <> 0

-- Rule: All ports transfer requests
inv portTransfersRequests :  self.portTransfersRequests



context Trigger

-- Verifies that the trigger defined in a class has a corresponding
    reception in a port type
def: signalDefinedInPortType : Boolean =
  self.getOwningClass.portRequests.name->includes(self.getSignalName)

-- Rule: The signal of a trigger is defined within a port's type
inv triggerCompleteness : self.signalDefinedInPortType


context State

-- Verifies if the target port of a SendAction can transfer the request (i.e
    . a reception is defined in the port's type)
def: signalDefinedInPortType : Boolean =
  if self.isSendAction
    then self.sendActionTarget.requests.name->includes(self.sendActionSignal
        .name)
  else
    true
  endif

-- Rule: The signal of a SendAction is defined within a port's type
inv sendActionCompleteness: self.signalDefinedInPortType



context Class

-- Rule: Stereotypes are disjoint
inv correctDefinition : (isContract and not isAssumption and not isGuarantee
     and not isObserver and not isPortType and not isRoot) or
  (not isContract and isAssumption and not isGuarantee and not isObserver
      and not isPortType and not isRoot) or
  (not isContract and not isAssumption and isGuarantee and not isObserver
      and not isPortType and not isRoot) or
  (not isContract and not isAssumption and not isGuarantee)
```

```
context Dependency

-- Rule: Stereotypes are disjoint
inv correctDefinition : (isImplementation and not isUsage and not
    isRefinement and not isConformance) or
  (not isImplementation and isUsage and not isRefinement and not
     isConformance) or
  (not isImplementation and not isUsage and isRefinement and not
     isConformance) or
  (not isImplementation and not isUsage and not isRefinement and
     isConformance) or
  (not isImplementation and not isUsage and not isRefinement and not
     isConformance)



context Class

-- Verifies if the contract has a composite assumption
def: contractHasAssumption : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
     oclAsType(uml::Class).isAssumption and a.isComposite)->size() = 1

-- Verifies if the contract has a composite guarantee
def: contractHasGuarantee : Boolean =
  self.ownedAttribute->select(a | a.type.oclIsTypeOf(uml::Class) and a.type.
     oclAsType(uml::Class).isGuarantee and a.isComposite)->size() = 1

-- Verifies if the contract is well formed
def: contractPropertiesWellFormed : Boolean =
  if self.isContract
    then self.contractHasAssumption and self.contractHasGuarantee
  else
    true
  endif

inv contractPropertiesWellFormed: self.contractPropertiesWellFormed



context Dependency

-- Verifies if the implementation has one source
def: implementationHas1Source : Boolean =
  self.client->size() = 1

-- Verifies if the source of an implementation is a class
def: implementationHasClassSource : Boolean =
  self.client->asOrderedSet()->at(1).oclIsTypeOf(Class)

-- Verifies if the source of an implementation is well-formed
def: implementationSourceWellFormed : Boolean =
  if self.isImplementation
    then self.implementationHas1Source and self.implementationHasClassSource
  else
    true
  endif

inv implementationSourceWellFormed : self.implementationSourceWellFormed
```

```
-- Verifies if the implementation has one target
def: implementationHas1Target : Boolean =
  self.supplier->size() = 1

-- Verifies if the implementation’s target is a contract
def: implementationHasContractTarget : Boolean =
  self.supplier->asOrderedSet()->at(1).oclIsTypeOf(uml::Class) and self.
      supplier->asOrderedSet()->at(1).oclAsType(uml::Class).isContract

-- Verifies if the target of an implementation is well-formed
def: implementationTargetWellFormed : Boolean =
  if self.isImplementation
    then self.implementationHas1Target and self.
        implementationHasContractTarget
  else
    true
  endif

inv implementationTargetWellFormed : self.implementationTargetWellFormed



context Dependency

-- Verifies if the usage has 1 source
def: usageHas1Source : Boolean = self.client->size() = 1

-- Verifies if the source of a usage is a part
def: usageHasPropertySource : Boolean = self.client->asOrderedSet()->at(1).
    oclIsTypeOf(uml::Property)

-- Verifies if the source of a usage is well-formed
def: usageSourceWellFormed : Boolean =
  if self.isUsage
    then self.usageHas1Source and self.usageHasPropertySource
  else
    true
  endif

inv usageSourceWellFormed : self.usageSourceWellFormed

-- Verifies if the usage has 1 target
def: usageHas1Target : Boolean = self.supplier->size() = 1

-- Verifies if the target of a usage is a contract
def: usageHasContractTarget : Boolean = self.supplier->asOrderedSet()->at(1)
    .oclIsTypeOf(uml::Class) and self.supplier->asOrderedSet()->at(1).
    oclAsType(uml::Class).isContract

-- Verifies if the target of a usage is well-formed
def: usageTargetWellFormed : Boolean =
  if self.isUsage
    then self.usageHas1Target and self.usageHasContractTarget
  else
    true
  endif

inv usageTargetWellFormed : self.usageTargetWellFormed
```

```
context Dependency

-- Verifies if the refinement has a source set
def: refinementHasSources : Boolean =
  self.client->size() >= 1

-- Verifies if all of the refinement sources are contracts
def: refinementHasContractSources : Boolean =
  self.client->forAll(c | c.oclIsTypeOf(uml::Class) and c.oclAsType(uml::
      Class).isContract)

-- Verifies if the refinement source is well-formed
def: refinementSourceWellFormed : Boolean =
  if self.isRefinement
    then self.refinementHasSources and self.refinementHasContractSources
  else
    true
  endif

inv refinementSourceWellFormed: self.refinementSourceWellFormed


-- Verifies if the refinement has one target
def: refinementHas1Target : Boolean =
  self.supplier->size() = 1

-- Verifies if the refinement target is a contract
def: refinementHasContractTarget : Boolean =
  self.supplier->asOrderedSet()->at(1).oclIsTypeOf(uml::Class) and self.
      supplier->asOrderedSet()->at(1).oclAsType(uml::Class).isContract

-- Verifies if the refinement target is well-formed
def: refinementTargetWellFormed : Boolean =
  if self.isRefinement
    then self.refinementHas1Target and self.refinementHasContractTarget
  else
    true
  endif

inv refinementTargetWellFormed: self.refinementTargetWellFormed



context Dependency

-- Verifies if the conformance has one source
def: conformanceHas1Source : Boolean =
  self.client->size() = 1

-- Verifies if the conformance source is a contract
def: conformanceHasContractSource : Boolean =
  let s:NamedElement = self.client->asOrderedSet()->at(1) in
  s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::Class).isContract

-- Verifies if the conformance source is well formed
def: conformanceSourceWellFormed : Boolean =
  if self.isConformance
    then self.conformanceHas1Source and self.conformanceHasContractSource
  else
```

```
    true
  endif

inv conformanceSourceWellFormed : self.conformanceSourceWellFormed


-- Verifies if the conformance has a target set
def: conformanceHasTargets : Boolean =
  self.supplier->size() > 0

-- Verifies if the conformance has only in the target set
def: conformanceHasObserverTargets : Boolean =
  self.supplier->forAll(s | s.oclIsTypeOf(uml::Class) and s.oclAsType(uml::
      Class).isObserver)

-- Verifies if the conformance target is well-formed
def: conformanceTargetWellFormed : Boolean =
  if self.isConformance
    then self.conformanceHasTargets and self.conformanceHasObserverTargets
  else
    true
  endif

inv conformanceTargetWellFormed : self.conformanceTargetWellFormed
```

Abstract

A compositional approach based on components and driven by requirements is a common method used in the development of critical real-time embedded systems. Since the satisfaction of a requirement is subject to the composition of several components, defining abstract and partial behaviors for components with respect to the point of view of the requirement allows for a manageable design of systems. In this paper we consider such specifications in the form of contracts. A contract for a component is a pair (assumption, guarantee) where the assumption is an abstraction of the component's environment behavior and the guarantee is an abstraction of the component's behavior given that the environment behaves like the assumption. In previous work we have defined a formal contract-based theory for Timed Input/Output Automata with the aim of using it to express the semantics UML/SysML models. In this paper we propose an extension of the UML/SysML language with a syntax and semantics for contracts and for the relations they must satisfy. Besides the important role that contracts have in design, they can also be used for the verification of requirement satisfaction and for their traceability.

Keywords

component, contract, compositional reasoning, UML/SysML, meta-model, design, well-formedness, V&V