Contract-Based Interface Specification Language for Functional and Non-Functional Properties

Richard Payne and John Fitzgerald
Contract-Based Interface Specification Language for Functional and Non-Functional Properties

R. Payne and J. Fitzgerald

Abstract

Projects developing, assessing or maintaining Systems of Systems (SoSs), are hampered by the difficulty of gaining confidence that a particular architecture will indeed achieve global SoS-level requirements. Recording explicit technical contracts at the boundaries of constituent systems could help with with the complex task of determining whether a candidate architecture respects global properties.

The SysML\textsubscript{C} extension of SysML has been defined using UML stereotypes. The definition is accompanied by an informal overview, a formal abstract syntax and static semantics definition, enabling consistency checking of models defined in the language. SysML\textsubscript{C} is at the earliest stages of development, but it appears to: provide a basis for increasing confidence in the substitutability of SoS constituent systems; allow rich contract-based descriptions of functional interfaces to be defined for the provided and required services of constituent systems and SoSs; allow SoS designers to record non-functional properties of system interfaces and services; and provide a means for SoS designers to record contract agreements at the required levels of rigour from informal notes to formal refinement proofs. A small example SoS has been used to demonstrate SysML\textsubscript{C}.
Abstract

Projects developing, assessing or maintaining Systems of Systems (SoSs), are hampered by the difficulty of gaining confidence that a particular architecture will indeed achieve global SoS-level requirements. Recording explicit technical contracts at the boundaries of constituent systems could help with determining whether a candidate architecture respects global properties.

The SysMLC extension of SysML has been defined using UML stereotypes. The definition is accompanied by an informal overview, a formal abstract syntax and static semantics definition, enabling consistency checking of models defined in the language. SysMLC is at the earliest stages of development, but it appears to: provide a basis for increasing confidence in the substitutability of SoS constituent systems; allow rich contract-based descriptions of functional interfaces to be defined for the provided and required services of constituent systems and SoSs; allow SoS designers to record non-functional properties of system interfaces and services; and provide a means for SoS designers to record contract agreements at the required levels of rigour from informal notes to formal refinement proofs. A small example SoS has been used to demonstrate SysMLC.

About the authors

Richard received his BSc (Hons) in Computing Science from Newcastle University in 2005. He has returned to Newcastle University to undertake a PhD under the supervision of Dr. John Fitzgerald, as part of the DIRC project, in the realm of Predictable Dynamic Resilience. Richard is currently researching two main areas. As part of his PhD, Richard is currently researching policy languages for the application in a resilience policy language and its semantics. The resilience policy language will be used in systems where components may enter and leave a system and its environment, all with changing levels of reliability. If the components in use degrade, then a policy (written at design time) will be utilised to reconfigure the system to a reliable state, with predictable results. The policy language will aim to integrate the concepts of component metadata and dynamic resilience mechanisms. His main role is as an RA on the Ministry of Defence funded SSEI project. He is involved in the Interface Contracts for Architectural Specification and Assessment sub task.

John Fitzgerald is a specialist in the engineering of resilient computing systems, particularly in rigorous analysis and design tools. He is perhaps most closely associated with the Vienna Development Method (VDM). A particular area of interest is predictable dynamic resilience: the design of systems that reconfigure in response to threats while retaining predictability. John is currently seconded to the Deploy project, leading its work on achieving and demonstrating dependability through the deployment of formal methods in four industry sectors. He initiated work on resilience-explicit computing in the ReSIST European Network of Excellence on Resilience in Information Society technologies, a concept taken up in the two projects that he jointly leads within the UK Software Systems Engineering Initiative SSEI. His newest project on the use of formal models to support collaborative modelling and simulation in the design of embedded systems (DESTECS), started in January 2010. John studied formal proof (PhD, Manchester Univ.), before joining Newcastle, where he worked on formal design techniques for avionic systems with British Aerospace. He went on to study the potential for industrial application of formal modelling (specifically, VDM and its support tools) as a SERC Fellow and later as a Lecturer at Newcastle. He returned to the University in 2003, having established the design and validation team at Transitive, a successful SME in the embedded processor market. John is Chairman of FME, the main European body bringing together researchers and practitioners in rigorous methods of systems development. He is a Fellow of the BCS, a member of the ACM and a member of the new EPSRC College from 2010.

Suggested keywords

CONTRACTS
ARCHITECTURE
SYSML
INTERFACE SPECIFICATION
SYSTEMS-OF-SYSTEMS
Abstract

Projects developing, assessing or maintaining Systems of Systems (SoSs), are hampered by the difficulty of gaining confidence that a particular architecture will indeed achieve global SoS-level requirements. Recording explicit technical contracts at the boundaries of constituent systems could help with with the complex task of determining whether a candidate architecture respects global properties.

The SysML\(_C\) extension of SysML has been defined using UML stereotypes. The definition is accompanied by an informal overview, a formal abstract syntax and static semantics definition, enabling consistency checking of models defined in the language. SysML\(_C\) is at the earliest stages of development, but it appears to: provide a basis for increasing confidence in the substitutability of SoS constituent systems; allow rich contract-based descriptions of functional interfaces to be defined for the provided and required services of constituent systems and SoSs; allow SoS designers to record non-functional properties of system interfaces and services; and provide a means for SoS designers to record contract agreements at the required levels of rigour from informal notes to formal refinement proofs. A small example SoS has been used to demonstrate SysML\(_C\).

1 Introduction

A system of systems (SoS) is “a set or arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities” [1]. Several characteristics make the engineering and assessment of SoS a considerable challenge. The majority of constituent systems may have been designed independently of the SoS and so were not intended for collaboration. They are managerially and operationally independent, and are often off-the-shelf. SoSs may change as requirements, SoS goals and the environment and infrastructure evolve. In a SoS architectural model, systems are composed into a SoS by linking interfaces defined at the boundaries of systems.
In our earlier work [2], we identified the current capability for expression of contract-based interfaces in industry-strength architectural notations and, based on the findings of the report, made a series of recommendations for future work:

- The development of a formal interface contract specification language as a proof of concept. This language would support the principles of design-by-contract, allowing the specification of pre-, post-, rely- and guarantee-conditions on ports of constituent systems.
- A contract-based interface language should enable a SoS developer to state non-functional properties in contracts.
- The SysML architectural notation provides a suitable basis for such an extension.

In this report we define SysML\(_C\), an extension of a subset of SysML, to support the specification of contract-based interfaces and the integration of functional and non-functional properties. The incorporation of contract-based interfaces in SysML\(_C\) enables developers to increase confidence of substitutability of constituent systems. Through the use of the design-by-contract (DbC) methodology [3], constituent systems may be replaced by alternative systems, or byx assemblies that offer the same or substitutable functionality with weaker or equivalent preconditions and stronger/equivalent postconditions. This property of substitutability aids in ensuring the correctness of evolving SoS whereby components may be upgraded or in reconfiguring SoS where systems may be replaced. SysML\(_C\) allows SoS designers to record non-functional properties of system interfaces and services, allowing analysis of SoS-level properties. Finally, SysML\(_C\) provides a means for SoS designers to record contract agreements that describe SoS designers’ intuition when composing the interfaces of constituent systems. We demonstrate the application of SysML\(_C\) using a SoS example with a range of functional and non-functional properties.

Outline of Report

In Section 2 we set out requirements that will guide the design of the language extension and serve as a basis for its evaluation. Section 3 provides a brief overview of SysML, in particular the subset of the language that we wish to extend. In Section 4 we detail our approach in developing SysML\(_C\), outlining the features that we wish to include, a discussion of the formality of the intended extension, and finally the language extension. An example SoS (Section 5) demonstrates the use of SysML\(_C\). In Section 6, we draw conclusions from the work to date and evaluate SysML\(_C\) against the requirements set out in Section 2. Finally, recommendations based on the findings of this report are detailed in Section 7.

2 Language Extension Requirements

The following requirements identify the essential features of a contract-based interface language extension. These requirements are based on our investigations, recommendations from [2] and from communications with the SysML community.

**R1** The language must extend the SysML architectural notation. Further, any addition should not contradict existing elements of the notation.
The extended language should allow the application of the design-by-contract (DbC) methodology in architectural specification. This includes the use of preconditions, postconditions and shared variable invariants.

Rely/Guarantee conditions should be used to avoid interference of shared variables.

Interfaces on system boundaries may be reused between SoSs. Contracts, therefore must be stated in the context of a given SoS and there can be multiple SoSs.

Non-functional properties should be represented—a SoS developer must be able to include informal theories for their representation and manipulation in models.

Existing language elements should be reused and extended where possible and appropriate.

Language extensions should be defined and documented in a complementary fashion to the existing definition as in the SysML language guide [4].

Language must support large-scale SoS model definitions.

Language must allow SoS developers the ability to reason about and record architectural models including the non-functional properties models.

3 Current SysML Features

An aim of our current work is to extend an existing industry-strength modelling notation with the capability to record contracts on constituent systems. Our previous report [2], reviewed the state of the art in architectural description notations, including their ability to support contract-based interface specifications. We concluded that SysML is the most suitable choice of notation into which support for contracts could be embedded. In Section 3.1 we briefly review the SysML notation and in Section 3.2 we identify the subset of SysML that is of particular relevance to contractual modelling.

3.1 SysML Background

SysML [4] is a notation for system modelling devised by the Object Management Group (OMG). SysML allows for the representation of SoSs, systems, hardware, software, information and processes. The language is a subset of UML 2.0 [5, 6], with extensions defined as a UML profile.

Like UML, SysML provides a number of diagrams to support the description of complementary aspects of a system. These diagrams are divided into three categories: structure, behaviour and requirements. The SysML diagrams use data defined in the metamodel, which may be captured in an XMI document.

The SysML standard defines a ‘precise natural language’ semantics for the different concrete syntactic diagram elements. It is stated that although a formal semantic definition is not provided for SysML (or any subset) at present, future versions may include a more formal definition.
3.2 SysML Subset of Interest

We focus on a subset of SysML that supports the modelling of SoS architectures. In particular, we concentrate on the block definition diagram (BDD) and internal block diagram (IBD). In [2] we gave an example architectural definition using BBD and IBD, which we repeat here. The example system contains three components connected through ports. Figure 1 depicts an extract of the illustrative example defined using a BDD. The System block contains Application, Operating_System and Hardware_Sensor blocks which are in turn composed of a number of blocks. As this figure is not intended to fully define the example system, we do not specify properties and operations of the individual blocks.

![Figure 1: SysML block definition diagram of example](image)

The example is elaborated further in the IBD of the System block, shown in Figure 2. This expands on the BDD, defining how the Application, Operating_System and Hardware_Sensor blocks which compose the System are connected. Blocks may have named ports, with connections between those ports. The connectors are local to the block System.

The SysML metamodel subset corresponding to BBDs and IBDs may be defined using XMI, as in the language specification of SysML [4], supporting model interchange. The information presented in the diagrams of SysML (such as block identifiers, operations and ports) is defined in the underlying XMI document – the diagrams simply present this data in a human-readable form. We take the same approach as the designers of SysML in that we identify first the elements of SysML that we require, followed by the elements of UML 2.0. These are summarised below.

SysML

The base elements that we use from SysML are:
Figure 2: SysML internal definition diagram of example

**Block** The base element of IBDs and BDDs, blocks may represent SoSs, constituent systems or components. Blocks extend the UML 2.0 notion of a Class, repurposing it to represent more than simply software classes.

**ValueType** Extending the DataType element of UML 2.0, a ValueType refers to any variable that may be identified by its values. A ValueType may also have an associated **Unit** and **QuantityKind**.

**UML 2.0**

The base elements that we use from UML 2.0 are:

**Package** A package acts as a container for model elements. A model may contain a number of packages and may reference elements in other UML 2.0 packages.

**Model Element** A entity present in any model. All base elements we use from UML 2.0 are model elements.

**Class** As described above, the class element is extended by the SysML block. For this reason, we must include it in the imported XMI document. The class element shall not be explicitly referenced by the SysML extension.

**Port** Renamed in SysML as standard ports, ports denote the interaction points of blocks. A port may describe the interactions through provided and required interfaces.

**Interface** Interfaces describe the services that may be provided or required by a block, in terms of operations.

**Operation** An operation describes a behavioural feature of the interface in terms of a signature (parameters and return type), and optional precondition and postcondition.
**Connector** A connector specifies the link that enables communication between two or more ports.

**DataType** As with a ValueType, a DataType is a type whose instances are identified by its values. A DataType may have operations and attributes.

**Association** In the subset we consider, an association describes the ownership and/or composition relationship between classes/blocks.

**State Machine** A state machine may express the behaviour of a model as a traversal of a graph of **States** connected by one or more **Transitions**.

### 4 SysML$_C$: a Contract-Based Interface Specification Language

In this section we first describe the general approach taken to extending SysML (Section 4.1) before giving the more detailed definition of SysML$_C$ (Section 4.2).

#### 4.1 Approach to Language Extension

Our extension to SysML adds three features: **contract-based interfaces**, **contract agreements** and **non-functional properties**. Sections 4.1.1 to 4.1.3 describe each of these in turn.

#### 4.1.1 Contract-Based Interfaces

The interfaces of a constituent system describe the services that the constituent system may provide or require. In SysML, such service descriptions typically show the available operations and their signatures, as well as any shared variables with initial values. The UML 2.0 language definition [6] allows operations to be defined (optionally) by means of preconditions and postconditions in the form of **constraints** which can be expressed in a language of the modeller’s choice (although the Object Constraint Language (OCL) is mentioned as a candidate). UML therefore supports the Design by Contract (DbC) methodology [3] in which each class defines a contract that will be respected by any valid implementation. The contract consists of preconditions and postconditions for each (public) operation of the class, and invariants over any global variables. The user of the class may therefore know under what conditions an operation may be executed (defined by the precondition) and the condition to expect (defined by the postcondition) if the operation is invoked when the precondition is satisfied.

Our previous report recommended the application of DbC to the definition of interfaces of constituent systems [2]. Such a specification style is optional in SysML, but we suggest that pre/postconditions should be compulsory attributes of each interface operation in SysML$_C$. In return for the effort of stating contracts explicitly, the SoS developer gains the ability to provide the contracts to developers of constituent systems, or use them as a basis for selection and assessment of off-the-shelf constituents. Such a DbC approach permits the analysis of SoS-level properties by composing constituent interface specifications, and enables assessment of alternative constituent system specifications may be compared or different allocations of functionality to constituents. Finally, DbC may promote system substitutability – systems may be replaced by alternative systems or assemblies that offer the
same or substitutable functionality with weaker or equivalent preconditions and stronger/equivalent postconditions. If pre/postconditions remain optional, these benefits may not be available across the whole of a SoS, reducing the potential value of the model.

Our previous report also recommended including rely/guarantee conditions in interface definitions. Rely/guarantee conditions [7] aim to address the interference on shared variables in situations where systems may operate concurrently. Rely conditions state assumptions about any interference on shared variables during the execution of system operations. Guarantee conditions state the effect on shared variables during operation execution. The benefits are similar to those of pre/postconditions and as such, are compulsory attributes of interface operations.

We will build on the Rely Guarantee Contract Language (RGCL) [8] by introducing shared variables on interfaces and the rely-guarantee notation on operations. RGCL presented an interface definition for component-based systems, for use in modular certification. RGCL was not applied to an architectural notation, but the DbC and rely-guarantee principles employed in RGCL interfaces are similar to those we envisage in SysML.

So far, our notion of contractual specification has been limited to behavioural description of operations. UML 2.0 supports the use of protocol state machines (PSMs) to describe the response of an interface to specified sequences of events, constraining the order of operations. PSMs are state transition diagrams, in which each transition is optionally labelled with pre/postconditions and an operation invoked on the transition. SysML removes PSMs in order to simplify the notation, on the basis that standard state machines are sufficient [4]. To an extent, we agree – the \{trigger, guard, activity\} tuple syntax of a SysML state machine transition may correspond loosely to the \{precondition, operation, postcondition\} syntax of a PSM transition. However, there is a semantic difference in that a SysML transition states that if an event occurs (when in the correct state) which matches the trigger and the guard evaluates to true, then the transition is ‘fired’ and the activity executed. This is different to the semantics of the PSM transition which only dictates that if the precondition is true then an operation may be executed which will result in the postcondition being true (assuming the postcondition is correctly specified). There is no obligation to execute an operation.

It is our opinion that enabling a SoS developer/analyst the ability to record DbC preconditions, postconditions, rely and guarantee conditions and PSMs (and thus allow a ‘may’ semantics in protocol state machines) in interfaces adds to the richness of interface definitions and increases the range of analyses available.

4.1.2 Contract Agreements

In the IBDs of SysML, blocks are linked by connectors. The ends of a connector, defined in SysML as ConnectorEnds, have an attribute roles. The semantics of roles is imprecise, however the UML standard indicates that roles denote the Port to

---

1 There is a discussion to be had regarding applying rely/guarantee conditions on interfaces as well as operations. We shall consider this in future work on SysML.

2 There is also a discussion to be had in applying operation-wide rely/guarantee conditions on interfaces themselves. We do not address this inclusion in this report, however it remains an issue for future work.

3 We say “loosely” as the types specified in SysML transitions are different to those of a PSM transition.
which a connector is attached. Note that there is no mention of the interface being connected.

The basic SysML syntax allows the construction of models that are ambiguous in some respects. Consider the IBD in Figure 3, which contains three parts $b_1$, $b_2$ and $b_3$. Part $b_1$ requires interfaces of two kinds: $i_1$ and $i_2$. Part $b_2$ provides interfaces $i_1$ and $i_2$, and part $b_3$ provides only interface $i_1$. If we consider that $b_1$ is connected to $b_2$ and $b_3$ as shown, there is no way to determine which part – $b_2$ or $b_3$ – is providing the interface $i_1$ to $b_1$. We may wish to choose, for example, between parts due to some non-functional property of each interface (we discuss non-functional properties further in Section 4.1.3). We therefore propose allowing a SoS developer to record the interface that is being connected, along with the port and optional part identifiers.

Figure 3: SysML example demonstrating need for recording interfaces in connector definition

The SysML standard states that each required interface must be linked with a provided interface. For such a link to be valid, either the required and provided interfaces must refer to the same interface type (as in the $i_2$ interfaces in Figure 3), or the provided interface must be a specialisation of the required interface in that it must contain at least the same operations and properties as the required interface. In modelling a SoS, greater flexibility may be required than is afforded by this definition. For example, two provided interfaces in combination may be sufficient to provide a required interface, or the level of formality or abstraction in interfaces may differ. In such cases, we wish to allow SoS developers to record the intuition behind linked interfaces.
We may also consider a situation where interfaces are not compatible. In such situations mismatches may occur between system interfaces and wrapper/bridge interfaces may be needed. Although this is beyond the scope of this work, the research into mismatches by Gamble [9] may provide some insight.

We define an additional construct in SysML\(_C\) in order to support the recording of rationale for interface links. We take inspiration from the SysML rationale construct which is a single text field in which the designer may record text in natural language, or refer to external documents or models. We will include in SysML\(_C\) a contract agreement for recording the designer’s reasoning as to how a given composition ensures that a required interface is provided. The contract agreement could feasibly include a natural language description or a more complex refinement or satisfaction justification.

4.1.3 Non-Functional Properties

Our previous report concluded that non-functional properties (NFPs) could be incorporated into the definitions of constituent system interfaces [2]. In order to support a range of NFP representations, SysML\(_C\) should allow each property description to contain, at a minimum, a unique name or identifier, a type, and a value. It is also advisable to record unit types, impact on other NFPs, operations on NFPs and algorithms for deriving NFPs from existing metadata in a constituent system’s definition. The SysML ValueType element (introduced in Section 3) has attributes for the definition of operations, sub-properties (for complex types), the unit and the kind of value.

The NFPs specified in an interface definition may be interface-wide, relating to the system or service as a whole (such as service availability), or relate to individual operations (such as response time). SysML\(_C\) therefore supports the statement of properties at both the interface and operation levels.

4.2 SysML\(_C\) Language Definition

In this section we define SysML\(_C\), an extension of SysML incorporating features motivated in Section 4.1. In Section 4.2.1 we introduce the extensions in the form of UML stereotypes, and provide an abstract syntax in the VDM\(^4\) notation. Given the syntactic definitions of the new constructs, in Section 4.2.2 we provide a static semantics for the language extension and the subset of SysML we consider.

4.2.1 Extension Stereotypes and Syntax

In this section we introduce the extensions made to SysML. We first provide UML stereotypes which define how the language elements of the SysML subset are extended and also illustrates relationships between the new elements where appropriate. The stereotypes are also reproduced for reference in Appendix A. New SysML\(_C\) language elements are prefixed with \(SysML_C\). Given the stereotype definition, we provide an abstract syntax of the constructs using the VDM notation. Only the main features of the abstract syntax are presented here; the definitions are given in full in Appendix B.

\(^4\)www.vdmportal.org
Below we provide a definition for an existing SysML language element. The first line of the definition gives the element’s name, in this case Unit. Defined after the element name and on subsequent lines are the attributes of the element – in this example symbol, description and quantityKind. Each attribute has a name, to the left of the colon, and a type to the right. In this case, a Unit contains a symbol which is of type String, a description which is also a String, and a quantityKind of type QuantityKind. Our convention is that attribute names are in lower case, while type names begin with an upper case letter.

\[
\text{Unit} :: \quad \text{symbol} : \text{String} \\
\quad \text{description} : \text{String} \\
\quad \text{quantityKind} : \text{QuantityKind}
\]

Language elements may be referenced in the definition of attributes of other elements. For example, we may consider the language element ValueType:

\[
\text{ValueType} :: \quad \text{quantityKind} : \text{QuantityKind} \\
\quad \text{unit} : \text{Unit}
\]

This has an attribute unit which is of the type Unit defined above.

**Package**

The top-level element of the model that we consider is the SysML Packages. We extend the existing language definition as shown in the stereotype in Figure 4, allowing the explicit introduction of non-functional characteristics (nftypes). The remainder of the definition remains unchanged from SysML. We consider a SysML architectural model to contain locally defined functions and datatypes, block types, the composition relationship of those blocks and definitions of the interfaces used by the blocks.

The SysML BDD mainly utilises the block and association elements to visualise the compositional structure of the system/SoS model. We intend the IBDs to use the details of the blocks, local types and interfaces in describing the communication relationships. This ties with the description of SysML we provide in Section 3.

![Figure 4: Stereotype depicting abstract syntax extension for SysML Packages](image)
Throughout the abstract syntax of SysML\textsubscript{C} and the SysML subset we consider, we have used mappings to define a number of the syntactic elements, ensuring the use of unique identifiers for the defined model elements.

The Package syntax states that the existing \textit{valuetypes}, \textit{assoc}, \textit{block} and \textit{interfaces} attributes remain, although the interfaces definition shall be modified. A new attribute is added – \textit{nftypes} – which is a mapping from an element identifier to a \textit{SysML\textsubscript{C}NFType} definition. Note the \textit{assoc} association mapping refers to 'global' links – that is the composition relationship between blocks. It does not relate to the communication connections between constituent blocks, which are referred to by the \textit{SysML\textsubscript{C}Connector} element discussed later in the document.

\begin{align*}
\text{SysML\textsubscript{C}Package} ::& \quad \text{valuetypes} : Id \xrightarrow{m} \text{ValueType} \\
& \quad \text{nftypes} : Id \xrightarrow{m} \text{SysML\textsubscript{C}NFType} \\
& \quad \text{blocks} : Id \xrightarrow{m} \text{Block} \\
& \quad \text{assoc} : Id \xrightarrow{m} \text{Association} \\
& \quad \text{interfaces} : Id \xrightarrow{m} \text{SysML\textsubscript{C}Interface}
\end{align*}

\textbf{Contract-Based Interfaces}

Interface definitions in SysML typically define only the signatures of those operations forming an interface. SysML\textsubscript{C} extends the \textit{Interface} and \textit{Operation} definitions as shown in Figure 5 and as outlined in Section 4.1.1. Figure 5 also shows that the extended \textit{Interface} definition also contains a \textit{SysML\textsubscript{C}ProtocolStateMachine}, a UML 2.0 language element omitted from SysML.

A \textit{SysML\textsubscript{C}Interface} contains five attributes: \textit{name}, \textit{vars}, \textit{nf}, \textit{ops} and \textit{psm}. The name we consider replicates the identifier associated with an instance of the element. The \textit{vars} element is considered a collection of shared variables local to the interface, accessible by all operations of the interface and is defined as a mapping from an identifier to a \textit{ModelVar} – which describes the variable type (may be a...
predefined primitive type or model-specific `ValueType`) and an initial value. NFPs which apply to the interface and operations may be defined by the `nf` attribute – an identifier is mapped to a `NFVar` which describes the type of the non-functional property (defined in the `nftypes` attribute of the `SysML.C.Package` element) and allows the system designer to define a value of the property. The `ops` attribute maps an identifier to an operation definition, as in the existing language definition. Finally, `psm` is an optional attribute (denoted by the square brackets around the attribute type) which corresponds to an instance of a `SysML.C.ProtocolStateMachine` element.

We extend the SysML definition of an operation to include five additional attributes: `nf`, `pre`, `post`, `rely` and `guar`. The operation-specific NFPs, `nf`, are defined in the same way as in the `SysML.C.Interface` definition. The precondition and rely conditions describe those conditions we expect to hold before and during the execution of the operation respectively. The guarantee and postconditions describe the obligations that the operation must respect during and after execution respectively. The `pre`, `post`, `rely` and `guar` attributes are defined as expressions and have access to shared variables and non-functional properties of the interface, parameters of the operation and non-functional properties of the operation. The operation definition does not contain any algorithm as to how a result is obtained. Note, the parameter list may be empty in the sense that an empty mapping may be provided.

The protocol state machine is a UML 2.0 variant of the state machine included in SysML. We have defined a simple syntax for the `SysML.C.ProtocolStateMachine` element, with three attributes: `states`, `transitions` and `initial`. The `states` attribute is a mapping of an identifier to a `SysML.C.State`. We consider only ‘simple’ states (this is the term used in UML) – the contents of which are not of concern. The attribute `initial` refers to the identifier of a state defined in the `states` attribute. Finally, a collection of `SysML.C.ProtocolTransitions` are defined in the `transitions` attribute.
The *SysML* ProtocolTransition element differs slightly from the syntax of the UML syntax. The element has three attributes: `startState`, `endState` and `operation`. Two states are referenced, the identifier of the state from which the transition arises, `startState` and the state which the transition leads to, `endState`. The `operation` attribute refers to an identifier of a defined operation. In the UML syntax, the precondition and postcondition of the operation are also defined as attributes of the transition. As we define these in the definition of the operation, they are omitted from the *SysML* ProtocolTransition syntax.

\[
\text{SysML}\_\text{Connector} :: \text{name} : \text{token} \\
\text{parties} : \text{Id} \xrightarrow{m} \text{SysML}\_\text{Party} \\
\text{contracts} : \text{Id} \xrightarrow{m} \text{SysML}\_\text{ContractAgreement}
\]

The *SysML* Party element retains the `port` and optional `part` attributes of the ConnectorEnd element, adding the `interface` attribute. This refers to an identifier of an interface attached to the port denoted in the `port` attribute.
As stated in Section 4.1, we consider a new element to be added to the SysML language, based on the existing Rationale element. This new element, SysML_C_ContractAgreement, contains no additional elements, but we wish this construct to be distinct from a Rationale element – specific to connectors. The existing body attribute remains as a String, this allows an SoS designer to record contract satisfaction in natural language. Recording the body as a String also enables SoS designers to reference artefacts such as satisfaction or refinement proofs if a formal verification approach is to be taken.

SysML_C_ContractAgreement ::

name : token
body : String

NFType

The final extensions we present are the stereotypes for the NFProperty, NFType and NFOp elements – presented in Figure 7.

![Stereotype depicting abstract syntax extension for SysML_C NFTypes and NFOps](image)

SysML_C extends the existing LiteralSpecification element of UML to include non-functional values – NFProperty. The NFProperty element has a value and a NFType, an extension of NFType. SysML_C extends the ValueType definition largely to separate the definitions of the functional and non-functional properties of a SoS. The ValueType element also contains concepts we require for NFTypes such as operations, units and generalisation. As such, the syntax for introducing NF types to a SoS definition remains largely the same as a ValueType. A NFType has an option generalisation attribute gen, a mapping of identifier to NFOp – operations which may be performed on the NF type, a property attribute which defined the underlying type - which may be either a simple predefined type or a complex, compound type.
and the unit and quantityKind attributes which state the measurement units of the type. Note, NFOps is optional in the sense that an empty mapping may be provided.

\[
\text{SysML}_C\text{-NFType} :: \begin{align*}
\text{name} & : \text{token} \\
\text{gen} & : [\text{Id}] \\
\text{ops} & : \text{Id} \xrightarrow{m} \text{SysML}_C\text{-NFOp} \\
\text{prop} & : \text{PredefinedType} | \text{Id} \xrightarrow{m} \text{PredefinedType} \\
\text{unit} & : [\text{token}] \\
\text{quantityKind} & : [\text{token}]
\end{align*}
\]

The NFOp syntax simply records the operation signature for each operation which may be performed on a NFType.

\[
\text{SysML}_C\text{-NFOp} :: \begin{align*}
\text{name} & : \text{token} \\
\text{param} & : \text{Id} \xrightarrow{m} \text{SysML}_C\text{-NFType} \\
\text{returnt} & : [\text{SysML}_C\text{-NFType}]
\end{align*}
\]

Given a NFP with a specific NFType, the NFOps of that type allow the calculation of the value of a NFP based on the input of a number of different NFTypes. For example, consider a NFP with a NFType availability. This NFType may have a NFOp which calculates system availability from a failure rate attribute and a recovery time attribute. As NFPs may be added at the interface of an SoS, we may consider NFOps which calculate SoS-level NFPs given NFPs of constituent systems.

### 4.2.2 Extension Static Semantics

A static semantics is defined for SysML\(_C\). This static semantic definition is given in the form of well-formedness functions which restricts models to those which may have a valid meaning. These functions correspond to the type checking of a model. The functions are lengthy and we do not intend to describe them in the body of the report. The static semantics is defined in full in Appendix C.

### 5 SysML\(_C\) Example

In this section, we apply the contract-based interface notation SysML\(_C\) to a simple example SoS – illustrating the extensions we have proposed. In Section 5.1, we outline the example with a SysML\(_C\) architectural model and an overview of the non-functional properties we feel are of interest. Section 5.2 details the application of SysML\(_C\) contract-based interfaces to the example.

#### 5.1 Outline of SysML\(_C\) Architectural Model

A Global Navigation Satellite System (GNSS) is the standard term for satellite navigation systems. In this example, we are considering an abstraction of the European GNSS which aims to provide position data with some overlay data to increase the accuracy and integrity of the position of users of the GNSS.

The simplified GNSS example is a SoS, of two constituent systems: a position system and an overlay system. The BDD (Figure 8) shows that the GNSS is composed of a Position system block and an Overlay system block – the directional arrows with filled diamonds show the compositional relationships between blocks.
BDDs do not provide details of the communication relationship between the Position and Overlay blocks, nor of the block ports or other properties.

In this example we are considering the US Global Position System (GPS) as an instance of a position system (the European Galileo positioning system is not yet operational – however we could envisage the design of changing systems a good application of the this work), and the European Geostationary Navigation Overlay Service (EGNOS) as an instance of an overlay system. Given the basic BDD composition diagram in Figure 8, we can consider the definition of the ports and the provided and required services on the block instances. The GPS position system has only one port and provides one service – the broadcast of position data (this data contains timestamp, orbit and satellite health information which we shall consider later in the example). The EGNOS overlay system has two ports, one requiring a position service and the other providing an overlay service (which includes data to increase position accuracy and data pertaining to satellite integrity). The GNSS SoS in turn has one port providing two services – a position service and an overlay service – those provided by the constituent systems. Figure 9 below illustrates the blocks, their ports and the connections between them. The provided and required services are illustrated as interfaces – the notation used in SysML is to represent provided interfaces as a ball, and required interfaces as a socket.

5.1.1 Non-Functional Properties in Example

In a SoS such as this example, there are a large number of NFPs a system or SoS designer should consider when specifying contract-based interfaces and also during architectural design. In our previous report [2], we surveyed a number of approaches in the identification, specification and verification of NFPs. The report
identifies a number of classifications of NFP, many of which may be considered in this example. As this example is intended to demonstrate the application of the notation introduced in Section 4, we consider a small subset of the applicable NFPs. Those NFPs we consider are detailed below, with a brief description, the type of the NFP and, where applicable, any unit used to describe the value.

**Accuracy** – the distance deviation from the actual (true) response, *primitive type*: natural number, *unit*: metre.


**Users** – the classification of users, *primitive type*: enumeration.

**Availability** – the proportion of time a service is functioning, *primitive type*: natural number (0-100).

**Integrity** – an indication as to whether integrity data is provided, *primitive type*: Boolean.

**SafetyOfLife** – an indication as to whether a service may be used in safety critical applications, *primitive type*: Boolean.

The NFPs detailed above include a mixture of broad classifications: *dependability* (availability, integrity), *performance* (response time), *precision* (accuracy) and *example-specific* (SafetyOfLife, users). The selection also has a number of different underlying primitive types (natural numbers, Boolean and enumeration). Complex types are not represented in this example.
The SysML package diagram in Figure 10 is used to depict the NF types detailed above. Each NF type has a defined underlying type - presented in angle brackets as per the syntax in Appendix B. Optional unit and quantityKind values are given where appropriate. In this small example, we do not define operations for the types.

Figure 10: Global Navigation Satellite System NF Type definitions

5.2 Application of Contract-based Interfaces

The IBD in Figure 9 depicts four interfaces in the GNSS example: overlay, egnosOverlay, position and gpsPosition. In this report we present interface definitions of two of these to illustrate SysML C. Figure 11 presents a BDD which depicts the overlay and egnosOverlay interfaces.

The overlay interface has one global variable, overlayDevice, a Boolean variable, one global NFP – av of type Availability and value 99. A single operation is included in the interface, also named overlay. The operation has the expectation stating that an overlay-capable device must be used and that the device remains overlay-capable during the time of operation execution. This is a slightly artificial expectation in that the actual operation is broadcast, however it illustrates the idea of an operation expectation. The obligation of the operation states that the overlay result contains position overlay data – this is assuming that the Overlay ValueType contains optional attributes. Three NFPs are stated for the operation, the accuracy of the result, response time and the users which may utilise the operation.

Finally the overlay interface has the PSM overlayPSM, shown in Figure 12, associated with it. We see that there is a single Standby state and a single protocol transition from this state. The protocol transition contains the overlay operation.
precondition and postcondition and returns to the original state after operation execution.

The egnosOverlay interface is similar to the overlay interface. The interface contains two operations – the first is available to all users, the other a commercial service. This is determined by the NFP users. The commercial operation has significantly better NFPs – the accuracy in the commercial operation is 1m (normal operation is 2m) and the response time is 10s (normal operation is 20s).

The egnosOverlayPSM, shown in Figure 13 details the PSM for the egnosOverlay interface. We see that there is a single Standby state and a two protocol transitions from this state. The two protocol transitions relates to the two operations of the interface, both return to the original state.

Finally, we present the contract agreement between the two interfaces. Figure 14 presents a subsection of the IDB in Figure 9 with an added contract agreement on the connector between the egnosOverlay and overlay interfaces. This connector links a provided interface of a system (EGNOS) to a provided interface of the SoS.
(GNSS) containing the EGNOS system. The contract, named OverlayContract, has a natural language body. The agreement in this example is simple in that we propose (without providing evidence) that the egnosOverlay interface provides a ‘better’ service than is provided at the SoS level in the overlay interface. Because of this, we may consider that the egnosOverlay interface satisfies the overlay interface specification – the egnosOverlay interface is a refinement of the overlay interface. The justification provided is three-fold:

- The interface-level NFP – availability – is higher in the egnosOverlay interface (99%) than the overlay (95%).
- The overlay interface has only one operation – overlay(). This is matched to the egnosOverlay() operation on the egnosOverlay interface. The signatures and preconditions are the same, the postcondition is stronger on egnosOverlay. Finally, all operation-level NFPs – accuracy, response time and users – are equal or better in the egnosOverlay interface.
- The overlayPSM is a subset of the egnosOverlayPSM.

This contract agreement example is very simple and justifications are not evidenced. We make assumptions and simplifications to help us illustrate the language construct rather than the content one should expect. In future work we shall investigate a more detailed case study where more complex and complete contract agreement definitions shall be defined.

6 Conclusions

In this section we evaluate SysML\(_C\) in relation to the requirements defined in Section 2.

The SysML modelling language has been extended, as proposed in our previous report and stated in requirement R1. The extensions have been described in an informal overview, and are provided with UML stereotypes, an abstract syntax defined in VDM (and in the UML stereotypes) and static semantics in the form of well-formedness functions defined in VDM – this meets requirement R7.
Meeting requirements R2, R3 and R5, the extension of interfaces and their operations realises the Design by Contract methodology, rely/guarantee rules with shared variables and includes non-functional properties. NFPs may be defined in individual operations and interface-wide.

In defining contract-based interfaces, contract agreements and NFPs, we have extended existing elements of SysML/UML where appropriate. The T16Interface, ContractAgreement and NFType elements, for example, extend the UML Interface and SysML Rationale and ValueType elements respectively. The extensions maintain the intention of the existing elements, thus meeting requirement R6. Meeting requirement R4, SysMLC allows contract agreements to be defined as SoS-specific entities allowing interfaces to be used differently in individual SoSs.

Requirements R8 and R9 cannot be fully evaluated in this work package. R8 states that SysMLC must support large-scale SoS models. Although we can not satisfy this requirement at this stage, the small example in Section 5 leads us to believe that larger models can readily be expressed using the extended language. The example illustrates how NFPs may be defined using the language extensions, but we have not considered formal analysis of models at this stage in the work, and so cannot judge the extent to which we can address requirement R9. SysMLC allows the specification of composition operation signatures of NFPs. However, we acknowledge the need for comprehensive future research in the specification and analysis for algorithms for NFP composition between different NFTypes and at different levels of abstraction in a SoS. In future work, we plan to consider a larger-scale example which we feel will allow us to evaluate these requirements.

The example presented in Section 5 aims to introduce the SysMLC language definition constructs – we have been able to define interfaces with the new language elements and introduced an example contract utilising the interfaces. In future work,
we aim to provide a realistic assessment of the capability of the contract language and to indicate where further enhancements to the language are necessary. Because of this, case studies should exhibit representative properties and be of sufficient scale to exercise the contract language. Therefore a suitable candidate case study should meet the following requirements:

**Realistic Complexity** - Although a case study may contain a number of relevant abstractions, the complexity of the case study should be representative of real world SoSs.

**Clear SoS structure** - The architecture of a suitable case study should identify a number of systems and system types with clearly defined connectivity.

**Non-functional properties** - We aim to represent both functional and non-functional properties in predicates of interface contracts. As such, a case study should exhibit, ideally, properties of different underlying types (e.g. numeric, boolean, collections).

In future work, we intend to develop realistic case studies with input from members of SOSA, SysML communities and other stakeholders.
7 Recommendations for Future Work

We propose the following recommendations for future work aimed at advancing the state of practice in contract-based modelling and analysis of SoS.

- SysML$_C$ has potential to allow SoS designers to define and analyse contract-based interfaces, contract agreements and NFPs. We recommend it as a basis for future case study work.
- The SysML$_C$ language definition presented in this report should be subjected to review and revision by stakeholders in the SoS development and assessment community. Any changes should be documented and issued where appropriate.
- A case study should be developed in order to evaluate SysML$_C$ and the contract-based approach, as stated in Section 6. We aim to engage with members of SOSA and SysML communities in order to define SoS case studies adequate for the work of this task.
- The requirements R7 and R8 set out in Section 2 should be addressed: the ability to support large-scale models and the ability to reason about and analyse architectural models and NFPs.
- Discussions with participants in the OMG SysML effort are ongoing. We will maintain contact and aim to inform OMG of developments in this work.

Acknowledgements

The work described here was funded under the UK Software Systems Engineering Initiative (SSEI)$^5$ within Task 16 on Interface Contracts for Architectural Specification and Assessment.

The authors are grateful to colleagues in SSEI, Dstl, BAESystems and the Centre for Software Reliability at Newcastle for their helpful input: John McDermid, Neil Bowman, Jane Fenn, Carl Gamble, Steve Riddle, Kevin Dockerill Alan Grigg and Steven Beard.

8 List of References

References


$^5$http://ssei.org.uk/


9 List of Abbreviations

SysML System Modelling Language
SysML\textsubscript{C} System Modelling Language for Contracts
UML Unified Modelling Language
OCL Object Constraint Language
OMG Object Management Group
XMI XML Metadata Interchange
SoS System of Systems
DbC Design by Contract
RGCL Rely Guarantee Contract Language
BDD Block Definition Diagram
IBD Internal Block Diagram
GNSS Global Navigation Satellite System
EGNOS European Geostationary Navigation Overlay Service
NFP Non-Functional Property
PSM Protocol State Machine
VDM Vienna Development Method
A SysML\textsubscript{C} Stereotypes

Package
Stereotype for package element of extension

Connector
Stereotype for connector, contract and party elements of extension

Interface
Stereotype for interface, operation, protocol state machine and protocol transition elements of extension
NFType
Stereotype for NFType and NFOp element of extension
B SysML\textsubscript{C} Abstract Syntax

----- Package -----  
public SysMLc\_Package :: valuetypes : map Id to ValueType  
nftypes : map Id to SysMLc\_NFType  
blocks : map Id to Block  
assoc : map Id to Association  
interfaces : map Id to SysMLc\_Interface;

----- Block -----  
public Block :: name : token  
ports : map Id to Port  
parts : map Id to Part  
conns : map Id to SysMLc\_Connector;

public Port :: name : token  
provided : set of Id  
required : set of Id;

public Part :: name : token  
type : Id;

----- Associations - Block relationships -----  
public Association :: name : token  
owner : Id  
composite : Id;

----- Connector -----  
public SysMLc\_Connector :: name : token  
parties : map Id to SysMLc\_Party  
contracts : map Id to  
SysMLc\_ContractAgreement;

public SysMLc\_Party :: part : [Id]  
port : Id  
interface : set of Id;

----- Contract -----  
public SysMLc\_ContractAgreement :: name : token  
body : seq of char;

----- Interface -----  
public SysMLc\_Interface :: name : token  
vars : map Id to SysMLc\_ModelVar  
nf : map Id to SysMLc\_NFProperty  
ops : map Id to SysMLc\_Operation  
psm : [SysMLc\_ProtocolStateMachine];

----- Operation -----
public SysMLc_Operation ::
    name : token
    param : map Id to SysMLc_ModelType
    return : [SysMLc_ModelType]
    nf : map Id to SysMLc_NFProperty
    pre : Expression
    post : Expression
    rely : Expression
    guar : Expression;

----- Protocol State Machine -----  
public SysMLc_ProtocolStateMachine ::
    name : token
    states : map Id to SysMLc_State
    transitions : map Id to SysMLc_ProtocolTransition
    initial : Id;

public SysMLc_State = token;

public SysMLc_ProtocolTransition ::
    startState : Id
    operation : Id
    endState : Id;

----- Type and assignment for functional variables -----  
public SysMLc_ModelVar ::
    type : SysMLc_ModelType
    value : Value;

----- Type definitions for functional datatypes -----  
public SysMLc_ModelType = Id | PredefinedType;

public ValueType ::
    name : token
    gen : [Id]
    prop : PredefinedType |
    map Id to PredefinedType
    unit : [token]
    quantityKind : [token];

public PredefinedType = EnumerationType | PrimitiveType;

public EnumerationType ::
    enums : set of token;

public PrimitiveType = <Bool> | <Nat> | <Int> | <String>;

----- Value definitions for functional datatypes -----  
public Value = PredefinedValue | ComplexValueTypeValue;

public PredefinedValue = PrimitiveVal | EnumerationValue;
public PrimitiveVal = bool | nat | int | seq of char;

public EnumerationValue = token;

public ComplexValueTypeValue = map Id to PredefinedValue;

----- Type and assignment for non functional variables -----
public SysMLc_NFProperty :: type : Id
val : SysMLc_NFValue;

----- Type definitions for non-functional datatypes ------
public SysMLc_NFType :: name : token
  gen : [Id]
  ops : map Id to SysMLc_NFOp
  prop : PredefinedType |
    map Id to PredefinedType
  unit : [token]
quantityKind : [token];

public SysMLc_NFOp :: name : token
  param : map Id to SysMLc_NFType
return : [SysMLc_NFType];

----- Value definitions for non functional datatypes ------
public SysMLc_NFValue = PredefinedValue | ComplexNFValue;

public ComplexNFValue = map Id to PredefinedValue;
C SysML\(_C\) Static Semantics

Static Environments
--User defined types (F/NF)
public ModelEnv = map Id to Ref;

public Ref = TypeRef | BlockRef | AssocRef | PartRef | InterRef | ArchRef;

public TypeRef = ValTypeRef | NFTypeRef;

-- base/field types of complex valuetypes
public ValTypeRef :: prop : PredefinedType |
map Id to PredefinedType;

-- base/field types of complex valuetypes
public NFTypeRef :: prop : PredefinedType |
map Id to PredefinedType;

--Block env, block port members
public BlockRef :: ports : set of Id;

--Association env, owner/composite relationship of blocks
public AssocRef :: oid : Id
    cid : Id;

--Part env, referencing block type of part
public PartRef :: block : Id;

--Port env, details interfaces on ports
public InterRef :: p : set of Id
    r : set of Id;

public ArchRef = <Part> | <Interface> | <Operation> | <State>;

-- Variable env, states the type of a variable
public VarEnv = map Id to ModelType;

--NF env, states the NF type of a NFP
public NFEnv = map Id to Id;

SysML\(_C\) Well Formedness Functions
-- This function ensures all attributes of a SysMLc_Package are well formed.
-- Environments are constructed using the Package attributes - constructing
-- relevant Ref and Env static environment objects.

public wfPackage : SysMLc_Package \to \text{bool}
wfPackage(mk_SysMLc_Package(valuetype, nftype, blocks, assoc,
let env = \{v |-> mk_ValTypeRef(valuetype(v).prop) | v in set dom valuetype\} in
forall v' in set dom valuetype & wfValType(valuetype(v'), env) and
let env' = env ++ \{n |-> mk_NFTypeRef(nftype(n).prop) | n in set dom nftype\} in
forall n' in set dom nftype & wfNFType(nftype(n'), env') and
let env'' = env' ++ \{b |-> mk_BlockRef(dom blocks(b).ports) | b in set dom blocks\} in
forall i in set dom interfaces &
wfInterface(interfaces(i), env')
and let env''' = env'' ++ \{i' |-> <Interface> | i' in set dom interfaces\} in
forall a in set dom assoc &
wfAssociation(assoc(a), env''') and
let env'''' = env''' ++ \{a' |-> mk_AssocRef(assoc(a').owner, assoc(a').composite) | a' in set dom assoc\} in
forall b in set dom blocks &
wfBlock(b, blocks(b), env''''');
let venv = \{s' |-> vars(s').type | s' in set dom vars\} in
forall n in set dom nf & cnftp(nf(n), env) and
let nfenv = \{n' |-> nf(n').type | n' in set dom nf\} in
forall op in set dom ops & wfOperation(ops(op), env, venv, nfenv)
and let env' = env ++ \{o |-> <Operation> | o in set dom ops\} in
wfProtocolStateMachine(psm, env');

-- This function ensures all attributes of a SysMLc_ProtocolStateMachine are
-- well formed. The function also ensures that the initial state is in the
-- 'states' mapping.
public wfProtocolStateMachine : SysMLc_ProtocolStateMachine *
               ModelEnv -> bool
wfProtocolStateMachine(
    mk_SysMLc_ProtocolStateMachine(-, states, trans, init), env) ==
    init in set dom states and
    forall s in set dom states & wfState(states(s), env) and
    let env' = env ++ \{s' |-> <State> | s' in set dom states\} in
    forall t in set dom trans & wfProtocolTransition(trans(t), env');

-- This function is not yet specified. As we consider the state to be an
-- abstract token type, we do not provide a definition of the well-
-- formedness. This may change in future work.
public wfState : SysMLc_State * ModelEnv -> bool
wfState(-, env) ==
is not yet specified;

-- This function ensures a SysMLc_ProtocolTransition is well formed. The
-- function ensures that the identifiers provided for the start and end
-- states map to States in the Model environment and that the operation
-- identifier maps to an Operation in the Model environment.
public wfProtocolTransition : SysMLc_ProtocolTransition * ModelEnv -> bool
wfProtocolTransition(mk_SysMLc_ProtocolTransition(s, op, e), env) ==
    \{s, e\} subset dom env and is_(env(s), ArchRef) => env(s) = <State>
    and is_(env(e), ArchRef) => env(e) = <State> and
    op in set dom env and is_(env(op), ArchRef) => env(op) = <Operation>;

-- This function ensures a SysMLc_Operation is well formed. The function
-- ensures that the parameter and return types exist and are of an
-- acceptable type. Given this, the function ensures all attributes of a
-- SysMLc_Operation are well formed.
public wfOperation : SysMLc_Operation * ModelEnv * VarEnv * NFEnv-> bool
wfOperation(mk_SysMLc_Operation(-, ps, rtp, nf, pre, post, rely, guar),
            env, venv, nfenv) ==
    forall tp in set dom ps & is_(tp, Id) =>
        tp in set dom env and is_(env(tp), ValueType) and
is_(rtp, Id) => rtp in set dom env and is_(env(rtp), ValueType) and
forall n in set dom nf & cnftp(nf(n), env) and
let nfenv' = nfenv ++ {n' |-> nf(n').type | n' in set dom nf} in
wfExpression(pre, env, venv, nfenv') and
wfExpression(post, env, venv, nfenv') and
wfExpression(rely, env, venv, nfenv') and
wfExpression(guar, env, venv, nfenv') and

-- The function ensures the well-formedness of a Port. It ensures that the
-- identifiers given for provided and required interfaces exist and refer to
-- Interface reference objects in the Model environment.

public wfPort : Port * ModelEnv -> bool
wfPort(mk_Port(-, prov, req), env) ==
forall i in set prov union req &
  i in set dom env and is_(env(i), ArchRef) => env(i) = <Interface>;

-- The function ensures the well-formedness of a Part, ensuring the part is
-- of a defined Block type.

public wfPart : Part * ModelEnv -> bool
wfPart(mk_Part(pid, type), env) ==
pid in set dom env and
  is_(env(pid), PartRef) => env(pid).block = type and
  type in set dom env;

-- This function ensures a SysMLc_Connector is well formed, ensuring all
-- attributes of a SysMLc_Connector are well formed.

public wfConnector : SysMLc_Connector * Id * ModelEnv-> bool
wfConnector(mk_SysMLc_Connector(-, parties, contr), bid, env) ==
forall p in set dom parties & wfParty(parties(p), bid, env) and
forall c in set dom contr & wfContractAgreement(contr(c), rng parties, env);

-- This function ensures a SysMLc_Party is well formed. Depending on if a
-- party is a constituent part or a block boundary, the function ensures all
-- attributes have been predefined in the Model environment and are of the
-- correct type.

public wfParty : SysMLc_Party * Id * ModelEnv -> bool
wfParty(mk_SysMLc_Party(part, port, i), bid, env) ==
cases part:
  nil -> (bid in set dom env) and is_(env(bid), BlockRef) =>
    (port in set env(bid).ports) and
    (port in set dom env) and is_(env(port), InterRef) =>
      i in set env(port).p or i in set env(port).r,
      others -> part in set dom env and is_(env(part), PartRef) =>
        (env(part).block in set dom env) and
        is_(env(env(part).block), BlockRef) =>

  others -> part in set dom env and is_(env(part), PartRef) =>

port in set env(env(part).block).ports and
(port in set dom env) and is_(env(port), InterRef) =>
i in set env(port).p or i in set env(port).r
end;

-- As the contract agreement is a simple string at this time, we do not
-- provide a definition for it’s well-formedness function.

public wfContractAgreement : SysMLc_ContractAgreement * set of Party *
ModelEnv -> bool
wfContractAgreement(mk_SysMLc_ContractAgreement(-, -), parties, env) ==
is not yet specified;

-- This function ensures an Association is well formed – ensuring the owner
-- and composition identifiers map to Block references in the Model
-- environment.

public wfAssociation : Association * ModelEnv -> bool
wfAssociation(mk_Association(-, owner, composite), env) ==
{owner, composite} subset dom env and
is_(env(owner), BlockRef) and is_(env(composite), BlockRef);

-- The well-formedness of ValueTypes and NFTypes ensure that any links to
-- generalisations exist. Further work is required on the NFType definition
-- as SysMLc improves.

public wfValueType : ValueType * ModelEnv -> bool
wfValueType(v, env) ==
cases v.gen:
  nil -> true,
  others -> v.gen in set dom env and
            forall p in set dom v.prop & p in set dom env(v.gen).prop
end;

public wfNFType : SysMLc_NFType * ModelEnv -> bool
wfNFType(n, env) ==
cases n.gen:
  nil -> true,
  others -> n.gen in set dom env and
            forall p in set dom n.prop & p in set dom env(n.gen).prop
end;

-- We do not provide the body for the well-formedness function of
-- expressions here. We do not envisage the definition to be complex and
-- aim to provide a definition as SysMLc improves.

public wfExpression : Expression * ModelEnv * VarEnv * NFEnv-> bool
wfExpect(expr, env, venv, nfenv) ==
is not yet specified;