Evaluation of Architectural Frameworks Supporting Contract-based Specification

Richard Payne and John Fitzgerald
Centre for Software Reliability
School of Computing Science
Newcastle University
United Kingdom NE1 7RU

Abstract

The management and assurance of "systems of systems" (SoSs) are hampered by the difficulty of gaining confidence that a particular architecture of constituent systems will meet a global SoS-level requirement. Research suggests that recording explicit *technical contracts* at the boundaries between the SoS-constituent systems may help with verification of SoS-level properties, including non-functional characteristics. This report surveys approaches to the contract-based specification of SoS-constituent systems and focuses particularly on the description of such non-functional properties.

The report reviews (i) current techniques for the representation of non-functional properties and (ii) current architectural description methods. The aims of the review are to determine the extent to which it would be possible to state and reason (with machine assistance) about non-functional properties and to assess the ability to incorporate contract descriptions at component boundaries in architectural descriptions. A direction for future work in the development of contract specification languages suitable for SoSs is proposed.

1 Introduction

There is a drive towards the use of architectures for software intensive systems that are characterised as "systems of systems"(SoS)[1]. Such architectures are intended to help achieve adaptability and evolution in SoS constructed from reusable heterogeneous systems from disparate sources [1, 2, 3, 4]. However, the lack of homogeneity, central authority and coordination can add to complexity and make it difficult to predict SoS-level properties during design [6]. This in turn can affect dependability assessment and hence put system integration at risk of cost and time overruns.

[1]This is notably the case in the defence sector. The UK Ministry of Defence has published a series of documents [1, 2, 3, 4] highlighting the importance of such architectures in the future system acquisition. The Software Engineering Institute (SEI) conducted a workshop examining architectures of software-intensive systems acquired by the U.S. Army which examined SoS, system and software architectures [5].

A System of Systems (SoS) is a complex structure composed of largely pre-existing systems linked by a (typically networked) infrastructure and delivering a service that the constituent systems could not deliver separately. There has been considerable debate around the definition of SoS. Indeed, as recently as 2008, Sousa-Poza et al. have suggested that the SoS Engineering field is "Foundationally myopic" in its pursuit of knowledge of what exactly constitutes a SoS and that efforts in this direction would be confounded by the features that make SoS Engineeing challenging [7]. However, there is some convergence on the characteristics that make SoSs such an engineering challenge. Maier, for example identifies five key characteristics of SoSs [8]:

- Operational Independence: constituent systems are autonomous, acting to serve their own goals.
- Managerial independence: The constituent systems may be managed independently and so can change functionality or character during the life of the SoS in ways that were not foreseen when they were originally composed.
- Distribution: Components may be distributed and decoupled, with a communications infrastructure supporting collaboration.
- Evolutionary Development: As a consequence of the independence of constituent systems, the SoS as a whole will change over time to respond to changing goals or component characteristics.
- Emergence: the SoS exhibits new behaviour that its components do not exhibit on their own.

To Maier's we may add two further characteristics that pose significant challenges [9]:

- Belonging: the constituent systems serve a common mission and may require modification, for example through wrapping or linking interfaces, in order to achieve integration in the SoS.
- Diversity: the SoS is formed of heterogeneous component systems, many of which were not originally designed for cooperation with the others in the SoS. They will be developed and using a multiplicity of disciplines and so will be described using a wide range of methods and formalisms.

Progress towards goals of making SoS autonomous or scalable require progress in modelling and simulation as well as in verification, validation and assurance. These topics are at the foundation of Kalawsky's "grand challenges" in Systems Engineering [10]. Our work provides an evaluation of techniques that have been proposed for managing the assessment of SoS-level behaviour. These techniques tackle complexity by expressing requirements and guarantees of system functionality in explicit contracts defined at the boundaries of systems. Systems are composed – connecting provided and required functionality – into a SoS by linking *interface contracts* to form an architectural model.

To date, most research has been performed at the system level, that is a system composed of a number of components in some configuration. As such, the properties we may represent in interface contracts at the SoS-level may differ in the level of formality with which they can readily be expressed. We intend to investigate this aspect in our work.

The intention of recording interface contracts of connected systems is to assist the analysis of SoS-level properties, identifying vulnerabilities and performing trade-offs before developing or acquiring individual systems at early stages in architectural design, well upstream of integration. Interface contracts can be expressed in informal or formal notations; the task will, in part, focus on formal description because this brings the additional potential benefit of machine-assisted analysis.

In order to realise this vision, designers need the ability to include explicit contracts in architectural models expressed using established notations, describe functional and non-functional properties formally, and reason about the resulting model. While there have been major advances in techniques and tools for specifying architectures and functionality, these two areas are rarely brought together, and the analysis of non-functional properties has not been addressed in depth.

This report provides an initial assessment of the capability of architectural description frameworks for representing interface contracts. The concepts of contracts and non-functional properties are introduced (Sections 2 and 3). The properties required of existing architectural description languages and frameworks in order to support the exploitation of contracts are identified and a range of significant notations evaluated against them (Section 4). The next step in the work reported here is to develop a proof-of-concept study in which we construct a small contract specification language and apply it to a case study derived from industry application. On the basis of our evaluation in this report, we propose either AADL or SysML to support this work (Section 5).

2 Contracts

In the context of SoS models, contracts are descriptions of the constituent systems of a SoS given in terms of their expectations and the obligations placed on their behaviour. This has much in common with the ideas of *Design by Contract*, a software engineering technique introduced by Meyer [11, 12] in which contracts make explicit the relationships between systems in terms of *preconditions* and *postconditions* on operations and *invariants* on states. The precondition of an operation describes the conditions which must hold prior to execution in order for an operation to be guaranteed to terminate and provide a result. A postcondition denotes the properties of the output state that must hold as the result of operation execution, assuming a valid input satisfying the precondition. The invariant dictates properties of the system's state that must hold before and after (however not necessarily during) every operation execution. A contract on an operation, therefore, asserts that, given a state and inputs which satisfy the precondition, the operation will terminate and will return a result that satisfies the postcondition and respects any required invariant properties.

Contracts may contribute to 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. 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.

In Meyer's approach, contracts mainly specify functionality. Beugnard et al. [13]

expand the notion of a contract to architectures in which components provide services. A four-level structure for contracts is proposed (Figure 1). The first two levels of this

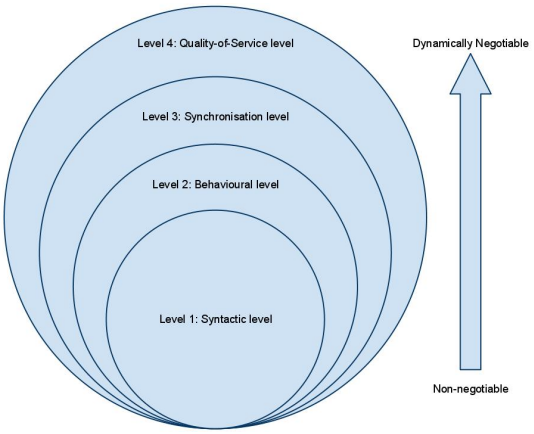


Figure 1: Beugnard et. al's four contract levels. Negotiability increases as the contract level increases. Based on [13]

contract structure relate to the principles outlined by Meyer: the *basic* layer specifies operations, their inputs, outputs and possible exceptions. The *behaviour* layer describes the abstract behaviour of operations in terms of their preconditions and postconditions. The third layer, *synchronisation* corresponds to real-time scheduling of component interaction and message passing. The fourth *quality of service* (QoS) level details non-functional aspects of operations and is particularly relevant to our work.

The contracts proposed by Beugnard et al. are subscribed to prior to service invocation. At subscription time, a service provider and user may negotiate the contract, which may also be altered at runtime. Using the notion of a layered contract, Beugnard et al. propose that the higher the level, the greater the flexibility for negotiation of contract content. The use of contracts in service selection and subscription is an active research field in service-oriented computing. This is especially true with regards to the role of QoS and non-functional properties; this point is discussed further in Section 3.

In situations where systems may operate concurrently, there is the possibility of interference on shared variables. We consider the use of rely/guarantee rules [14] in interface contracts. *Rely* conditions state assumptions about any interference on shared variables during the execution of operations by the system's environment. *Guarantee* conditions state the behaviour of the operation on shared variables during execution.

In this report, we examine the potential for incorporating interface such contracts into an existing architectural specification notation as a means of managing the complexity of gaining assurance in SoS architectures. The incorporation of interface contracts provides two main benefits:

- System-of-systems designers may define expected properties on interface contracts of the constituent systems, which may be provided to system developers. This

provides greater confidence to SoS designers that procured systems should adhere to these expected properties. Interface contracts provide additional guidance on implementation and governance to system developers in that they constrain the properties systems must guarantee to provide and define the properties system implementations may rely upon.

- Interface contracts defined for systems in a SoS architectural design provides the capability of analysing SoS-level properties - by stating reliances and guarantees on interface contracts and composing systems to form a SoS. These analyses may give SoS designers the ability to experiment with consequences of different architectural designs. This will, in part, rely on work on non-functional properties, discussed in Section 3.

We envisage interface contracts using preconditions and postconditions alongside rely/guarantee conditions on the operations of a system component. As discussed in Section 4 communication between components in some architectural notations occurs through ports. We may therefore need to adjust the placing of contracts from software operations to ports attached to system boundaries.

3 Non-Functional Properties

The properties of computing systems are often described as either *functional* or *non-functional*. Functional properties (FPs) are those that pertain to the functional correctness of the system. For example, the relation between system variables before and after a computation may be described as a functional property. Non-functional properties (NFPs) pertain to characteristics other than functional correctness. For example, reliability, availability and performance of specific functions or services are NFPs that are quantifiable. Other NFPs may be more difficult to measure: security or adherence to standards, for example. It is worth stressing that NFPs cannot be divorced from their functional or environmental context, but still relate to specific functions or services.

Many of the dependencies between the constituent systems in SoSs are non-functional in character. Consequently, the formalisation of contracts in architectural models should encompass NFPs. However, the state of the art in the formal expression and analysis of NFPs lags behind that of functional properties. In part this is because the theories needed to allow us to reason about the composition of NFPs are not well worked out, may be complex and in part because many NFPs are not preserved under composition. For example, two deadlock-free systems may well deadlock when composed. It may also be the case that some NFPs may be difficult to express at a system level and compose to SoS-level properties.

In this section we discuss approaches to the use and definition of non-functional properties. Research effort in the use and definition of NFPs is of growing interest in a number of areas of computing. We address a number of these areas and discuss those efforts. The role of NFPs in service-oriented architecture – in particular the service selection phase (Section 3.1), and in component-based computing (Section 3.2) are two areas we aim to address in this section. Along with research into the application of general NFPs in research, there is also effort placed on the representation and analysis

of specific NFPs (Section 3.3). Finally, we draw some observations from our survey of NFPs (Section 3.4).

3.1 Non-Functional Properties in Service Oriented Architectures

Recent research into non-functional aspects of service oriented architectures (SOA) has focused on the use of non-functional requirements (NFRs) in the selection phase of service deployment. In [15], the authors survey a number of approaches to service selection using NFRs, comparing the approaches to a number of requirements. Also a request for information (RFI) has been released to the Object Management Group (OMG) standards group [16] for NFR in SOAs.

In order to support reasoning, NFRs should have some underlying theory of non-functional attributes upon which the requirements are based. Therefore for the remainder of this section we investigate some approaches to incorporating NFRs in SOAs.

3.1.1 Taxonomy for Non-Functional Requirements

Galster and Bucherer address non-functional requirements (NFRs) in SOAs, acknowledging the fact that in SOAs NFRs have had less attention than the functional aspects of services [17]. They present a taxonomy of non-functional requirements, potentially useful as a checklist of NFRs that one may wish to ensure are specified. The taxonomy divides NFRs into:

Process Requirements placed on the service-oriented development lifecycle: *design*, *discovery*, *composition* and *runtime*;

Non-Functional External Requirements derived from the environment in which a system is developed and deployed; and

Non-Functional Service Requirements considered the 'actual' NFRs placed directly in the service or system.

Figure 2 presents an extract of the taxonomy, with examples of types of requirement from each of the three classes above. The taxonomy lists NFR types with an indication as to their quantifiability (a ticked box denotes a quantifiable NFR, blank box denotes a non-quantifiable NFR and two boxes denote a NFR is met or not met), a brief informal description and an example. For instance, the Non Functional Service Requirement *performance* is deemed to be quantifiable and is described as being the "response time, throughput and timeliness" of a service.

Although the taxonomy is described in terms of requirements, one can identify the types of properties upon which the requirements are based. The authors identify several areas which require further work. They state that there are a number of dependancies between NFRs which are not specified in the taxonomy and that some are difficult to differentiate. The work does not explicitly address composition of NFRs across systems with multiple services.

C0	Non-functional requirements		Quantifiability	Explanation and example
	C1	**Process requirements**		
		C1.1 Delivery requirements	☐■	Requirements to the delivery conditions of the service or the service-oriented system (e.g., services have to be delivered on customer site rather than made accessible in a de-centralized service repository which is not under the control of the customer).
		C1.2 Implementation requirements	☐■	Constraints on the implementation technology or methodology (e.g., services have to be implemented as web services using IBM WebSphere® technology).
		C1.3 Composition requirements	☐	Requirements on the composability of a service (e.g., services have to be composable with internally developed web services).
		C1.4 Standard requirements	☐■	Standards which a development process has to follow or the process has to comply with (e.g., service-oriented development process has to be ISO9000 conformant).
	C2	**NF External requirements**		
		C2.1 Legal constraints	☐■	Legal constraints on development process and services (e.g., a data protection officer must certify that the new service-based system is conform to the data protection legislation).
		C2.2 Economic constraints	n/a	Economic constraints on process and service / service based system (e.g., market conditions).
	C3	**NF Service requirements**		
		C3.1 Usability	☑	Measure of the quality of a user's experience in interacting with the service application (e.g., the product must help the user to avoid making mistakes, quantified as the ratio between number of functions understood by the user and the total number of functions).
		C3.2 Reliability	☑	Ability to keep operating over time (e.g., mean-time-to failure).
		C3.3 Performance	☑	Response time, throughput, and timeliness (e.g., messages exchanged per time-unit).
		C3.4 Capacity	☑	Ability to handle different capacities (e.g., max. number of service-users at a time).
		C3.5 Portability	☑	Ability to be ported to other environments (e.g., number of supported interface specifications for different target systems, such as WSDL).
		C3.6 Safety	☑	Capability of service / system to achieve acceptable levels of risk of harm to people, business or environment in a specified context (e.g., incidence of damage as the ratio between damage occurrences and total number of usage situations).

Figure 2: Extracts from taxonomy of non-functional properties [17]

3.1.2 A QoS Ontology Language

In the web services domain, Papaioannou et al. present a quality of service (QoS) ontology [18] which allows a number of QoS attributes, collectively referred to as the *QoS vocabulary*, to be represented using the *QoS language*. The main constructs of the QoS language ontology are:

- *QoSParameter* – the non-functional property identifier;
- *Metric* – the way in which a QoSParameter is assigned a value;
- *QoS Impact* – the manner in which a QoSParameter contributes to service quality;
- *Type* – specific category of vocabulary ontology the QoSParameter belongs to;
- *Nature* – a QoSParameter may be static (remain unchanged) or dynamic (may change at runtime); and
- *Aggregation* – indicates whether a QoSParameter is composed of multiple other QoSParameters.

QoSParameters are defined in XML, with metrics defined using a value (given as a string), datatype and unit. A small example of the NFP throughput is given in Figure 3 as an extract from the example in [18].

```
<QoSParameter rdf:ID= ‘‘Throughput">
  <hasType>
    <Type rdf:ID=‘‘Performance">
  </hasType>
  <hasMetric>
    <Metric rdf:ID=‘‘ThroughputMetric">
      <Value rdf:datatype=string> 5000 </Value>
      <MetricType rdf:datatype=string> Long </MetricType>
      <hasUnit>
        <Unit rdf:ID=‘‘ThroughputUnit">
          ...
        </Unit>
      </hasUnit>
    </Metric>
  </hasMetric>
</QoSParameter>
```

Figure 3: Extract from definition of “Throughput” NFP [18]

Using XML markup tags, we see the QoSParamter *Throughput* has the type $Performance$ and is a $ThroughputMetric$ with the value 5000. The type denotes a property category for the grouping of properties, not the inheritance of type attributes. The paper provides no reference to any theory to support the metrics described. No analysis is performed on the QoSParameters nor is a semantics defined for the composition of properties of a multi-service system.

3.1.3 Service Selection Based on Non-Functional Properties

Reiff-Marganiec et al. propose a notation for the representation of non-functional properties for use in SOAs [19]. In their approach, a service offers *operations* to which one or more *categories* are attached. A category contains *criteria* representing the NFPs. Each criterion is a tuple $\{Name, Type, Weight, Value\}$ where $Name$ is the unique criterion identifier, $Type$ is one of three main types (*numerical*, *boolean* or *string*), $Weight$ signifies the strength of the requirement (a weight of 1 signifies a hard requirement, 0 a soft requirement), and $Value$ represents the current value. For example, a criterion for cost may be represented as in the equation below.

$$\{Name = \text{“}cost\text{”},\ Type = \text{“}currency\text{”},\ Weight = \text{“0.5”},\ Value = \text{“100”}\} \quad (1)$$

The authors suggest the use of *evaluation functions* which are used to standardise criterion values. Ranking formulas are defined for use of service selection in SOAs. The notation is defined informally and the approach to criterion analysis is very simple. No semantics are given nor are any theories for the criteria types.

3.2 Component-Based and Architectural Approaches to Non-Functional Properties

3.2.1 Non-Functional Properties in Software Architecture

Van Eenoo et al. have discussed the lack of support for non-functional properties in architecture description languages (ADLs) [20]. They suggest that NFPs are often too abstract and informal, the separation of functional and non-functional properties is not simple and that NFPs are often an afterthought of the development process. The authors propose a new construct to an existing ADL – Wright (discussed in Section 4.5) – defined using a mixture of natural language and the process calculus CSP.

The new construct uses the notion of ‘required’ and ‘ensured’ non-functional properties at the interface of the components and connectors of a system. NFPs are defined using the $Non\text{-}Functional$ extension consisting a unique identifier and a collection of required and ensured properties defined in natural language. The extension dictates which NFPs are required by a given architectural entity (and from which entity they are required) and also the NFPs an entity ensures (and to which entity they are ensured). In the paper, a small example system is provided with some NFPs attached. An extract from that example, the component GP, is given in Figure 4.

The NFP *Security* states that the connector $Remote\_Procedure\_Call$ must encrypt data sent to the GP component. The GP component ensures that data passed to the *auth* port is authenticated. Further NFPs are defined which provide some basic numerical expressions. The authors state that ‘compliance’ checking can be performed to verify that all NFP requirements are met, however there is no indication of how this is to be done.

The authors acknowledge the challenge of formalising the construct for machine reasoning but state that the use of natural language for property definition allows for

Component GP

Port $auth = authenticate?x \rightarrow auth$

Port $write = start \rightarrow accept!x \rightarrow write$

Computation $auth.authenticate?x \rightarrow write.start \rightarrow computation$

Non-functional $Security =$

$(REQUIRES \subseteq Remote\_Procedure\_Call.encryption,$
$ENSURES \subseteq auth.authentication)$

Figure 4: Extract of example specification of GP component with non-functional property extension [20]

greater readability and accessibility. This trade-off between readability and formality limits the machine-assisted formal reasoning of system-level properties using this notation.

3.2.2 Non-Functional Properties of Component-Based Systems

In [21] Franch presents **NoFun**, a notation for the representation of non-functional properties of component-based software systems. A NoFun definition uses three elements. **Non-Functional Attributes** correspond to the non-functional datatypes such as time, efficiency or reliability. **Non-Functional Behaviours** describe the actual property - the assignment of a value to a Non-Functional Attribute. Finally, **Non-Functional Requirements** are constraints over the attributes.

Attributes are bound to individual components, component groups or to the whole system and are defined over the operations of the bound components. Figure 5 shows the attribute module $ERROR\_RECOVERY$ which contains two attributes; *op_error_recovery* and *error_recovery*. The attributes specify their type (both are defined as boolean types), whether they are bound to components or a subset of the component operations (*op_error_ recovery* is bound to all operations of a component, *error_recovery* is bound to a component itself), whether the attribute is derived from other attributes (the *error_recovery* attribute depends upon the *op_error_recovery* attribute) and finally its definition (*op_error_recovery* is not defined, *error_recovery* is defined – stating that a component has an error recovery mechanism if all its operations have error recovery).

The non-functional behaviours of components, defined separately to their implementation, form a component specification which the implementation must respect. The extract in Figure 6 shows a specification of the component implementation $IMP\_LIBRARY\_1$ which states that all operations of the component have error recovery. This is stated by the boolean expression $error\_recovery(ops(LIBRARY))$. The collection of operations in the $LIBRARY$ component ($op(LIBRARY)$) are supplied to the *error_recovery* attribute which, as described in Figure 5, requires all operations to have error recovery. Finally, the Non-Func requirements are defined as ordered predicates over the component behaviours; the intent of the requirements appears to be in the selection of components.

```
attribute module ERROR_RECOVERY
  attributes
    boolean op_error_recovery bound to all_ops
    boolean error_recovery
      bound to components derived
      depends on op_error_recovery(all_ops)
      defined as
        error_recovery =
          for all op in all_ops it holds
            op_error_recovery(op)
end ERROR_RECOVERY
```

Figure 5: Extract of the Non-Functional Attribute 'Error Recovery' [21]

```
behaviour module for IMP_LIBRARY_1
  behaviour
    error_recovery(ops(LIBRARY));
    ...
end IMP_LIBRARY_1
```

Figure 6: Extract of the Non-Functional Behaviour of 'Library' component [21]

NoFun has a 'well-defined' syntax and semantics, however the level of formality is not clear.

3.2.3 Formal Specification of Non-functional Properties of Component-Based Software Systems

Zschaler presents Quality-Modelling Language for Component-based Systems (QML/CS), a formal specification language for non-functional properties [22]. The language follows the concept of component-based systems closely: Zschaler describes a component-based system composed of a *container* which uses *components* and *resources*, provides *services* and has a *container strategy*. A formal semantic framework is provided in the extended Temporal Logic of Actions (TLA^+).

Zschaler introduces a notion of *component networks* to deal with systems using more than one component and also allows for component hierarchy. Component networks – as in architecture description languages (ADLs) – allow components to be connected, resulting in a configuration.

Non-functional properties are defined as constraints over *measurement* values. Measurements describe 'non-functional dimensions' of systems – synonymous with our notion of non-functional types – and are defined by state-based specifications of the system. Measurements do not influence system behaviour. Figure 7 demonstrates the specification of the measurement *response_time*. The definition shows that the mea-

surement *response_time* is of type *real*. *Probes* are defined (not shown in the example) as *end* and *start* to represent measurements referring to end time and start time of the last invocation of operation *op*. From this, *response_time* is defined.

```
in context RTContext
declare measurement real response_time
                                (ServiceOperation op) {
  spec op.invocations-> last.end -
       op.invocations-> last.start;
}
```

Figure 7: Example measurement definition of response time in QML/CS [22]

A component can then be specified referencing these measurements in the form of temporal logic expressions. Figure 8 shows a simple definition of the component *Counter* which provides one operation *getData*() the response time for which is always less than 60.

```
declare component Counter {
  provides int getData();

  always response_time(getData) < 60;
}
```

Figure 8: Example application model with definition of response time of getData() operation, defined in QML/CS. Based on [22]

Zschaler acknowledges there may be side effects on service non-functional properties when components are composed in a system configuration. Although measurements may be defined and non-functional properties specified for components and services in Zschaler's notation, there is no way to analyse system-level properties out of the composition of properties of the components.

3.3 Representation of Individual Non-Functional Properties

Alongside the research on generic methods of representing non-functional properties, significant work has been undertaken in the representation of individual types of NFP. Such work may provide a basis for the theories of NFPs required for the assessment of system-level properties. A theory for a given property should state how the property may be represented, relevant typing details and a semantics detailing the behaviour of the property and non-functional expressions. The semantics could also describe how properties may be composed.

The state of the art in formal theories of NFPs is somewhat inconsistent, often being conducted at the code level rather than the design level of systems engineering.

In particular the use of NFPs in system architecture and interface contracts has not been addressed in depth. In this section, we address approaches to the description of different classes of NFP. In Section 3.3.1, we address the measurement and analysis of dependability properties including availability and reliability. Section 3.3.2 details efforts to formally model and analyse timing and performance properties.

3.3.1 Evaluation of System Dependability Properties

The taxonomy of non-functional properties of service-oriented architectures presented in Section 3.1.1 identifies *reliability* and *availability* as examples of service NFPs. Aviziensis et al. describe dependability as encompassing availability, reliability, safety, integrity and maintainability – all of which may be considered non-functional types in our terminology [23]. There is also considerable research on measuring and evaluating these non-functional types in implemented systems [24], including the use of probabilistic evaluation methods including *reliability block diagrams* and *petri nets*. These evaluation methods are also described in a survey report by Gamble [25].

The measurements of availability and reliability properties and the evaluation of such properties in multi-component systems shows that a formal theory of these NFPs is an ongoing research issue, including the compositionality of reliability properties using reliability block diagrams. It should be noted, however, that the research on dependability properties has not been linked to architectural interface contracts and it is not clear how this research may apply in architectural models.

3.3.2 Modelling and Analysis of Timing Properties

The timing properties of software systems, highlighted in the taxonomy in Section 3.1.1, may constitute a number of non-functional types. We may consider *response time*, *throughput* and other real-time aspects as examples. In this section we detail an attempt at incorporating timing as first class properties in the verification of formal models and a detailed survey of methods for performance evaluation.

In [26], Fitzgerald et al. present an approach to validate system-level timing properties in formally defined VDM++ [27] models. The authors model a distributed embedded system on a number of virtual CPUs. The example described illustrates a number of performance requirements related to the response times of operations of the modelled system. The performance requirements, stated as validation conjectures, describe the temporal relationships between system events in a trace obtained by interpreting the system model. Patterns of such conjectures are described illustrating common relationships between system events. For example the requirement: *"a volume change must be reflected in the display within 35ms"* is expressed using the validation conjecture in Figure 9. The approach has been embedded into tool support that allows management of the simulations to verify timing conjectures and graphical presentation of the outcomes.

Balsamo et al. compare 15 attempts at modelling and analysing software performance [28]. The approaches examined centre on early stage development from architecture design to more detailed design. The approaches are compared on the basis of *level of automation*, *integration in development lifecycle* and *integration of software and*

```
deadlineMet(#fin(Radio‘VolumeUp), (#fin(MMI‘AdjustScreen), 35)
```

Figure 9: Example validation conjecture stating defined using VDM++ extension, based on [26]

performance models. Most of the approaches have distinct software and performance models. Typically software models are defined using UML diagrams – many of the approaches take an architectural component-based approach to the model definition. In contrast, performance models are defined by queueing networks, stochastic petri nets, process algebra and simulation models. While use of a common framework would allow seamless analysis, it would also require expertise across the divide between performance and software modelling. The majority of the approaches use existing tools for the specification of software models (e.g. UML tools) and in the performance analysis (queueing network analysers) ensuring a higher probability of acceptance of such analyses. However, the lack of feedback information is noted by the authors.

The performance evaluation approaches we have highlighted in this section acknowledge the need for analysis of non-functional properties at an early stage of software development, the need for tool support, integration with existing tools and methodologies and provide insight into formal validation of NFPs. It should be pointed out, as with dependability properties discussed in the previous section, the incorporation of formal performance verification and analysis has not been linked to interface contracts in architectural specifications and as stated in Section 3.3.1, it is unclear whether the such verification and analysis is possible in architectural models.

3.4 Observations

In this section, we have reported on several different approaches to the representation of non-functional properties. We have also identified a number of classifications and taxonomies of NFPs. We finish this section by drawing conclusions on the state of the art and propose how we wish to build upon existing work in defining a language for interface contracts.

A number of attempts have been made to classify NFPs [17, 29]. These classifications, whilst comprehensive in their coverage, lack formality. They focus on identifying properties which may be of interest and hierarchies of properties. For the formal verification of non-functional properties, we need to "plug in" theories of NFPs. For this to be a realistic achievement, significant effort must be placed in understanding the composition and relationships between NFPs. The classifications may be of use in identifying commonalities between the different properties.

Notations have been proposed in the service-oriented, component-based, architectural and safety case domains to represent NFPs. Some of the notations take a formal approach [22], a number attempt to achieve generality [21, 22] and existing specification languages are extended [20]. No analysis or verification of emerging properties is presented and all attempts are still far from industry readiness.

As we have identified, considerable effort is required to identify NFPs in systems and SoSs. Perhaps more difficult, however, is in the composition of NFPs. This is apparent

in our survey of approaches for the specification of NFPs. Of the approaches described in Sections 3.1.2, 3.1.3, 3.2.1 and 3.2.3, none address the problem of composition. We also see that efforts are made in a number of the SOA approaches in Section 3.1 to address utility of NFPs with regard to a weighting or preference on NFPs. This utility is not at a mature level, using informal notations and simple evaluation functions. Given this complexity, it is our belief that the effort required in determining NFP composition rules and guidance for identifying NFPs in systems/SoS is beyond the scope of our work. It is also our belief that stating NFPs informally in interface contracts – assuming compositionality is beyond the state of the art – will nonetheless provide benefits to system designers.

It is apparent from this survey of approaches to represent NFPs and perform analyses that each has negative aspects. We intend, in future work, to define a set of requirements for the representation of NFPs. These requirements shall depend on the interface contract language definition we develop and we intend to utilise the findings of this report to justify those requrements.

4 Architecture Description Languages/Frameworks

An *architecture description notation* allows a system modeller to describe the structure and behaviour of a system in terms of discrete elements of computation and hence to perform analyses in the design phase of system development. Medvidovic et al. surveyed a number of early Architecture Definition Languages (ADLs), describing the architecture of software systems [30, 31]. The advancement of embedded systems and systems-of-systems has strengthened the need for notations modelling both hardware and software elements.

In this section we review technically and industrially significant notations for the description of system architectures. We consider architectural notation to contain abstractions which describe elements of a software or system architecture. Commonly these are *system*, *components* and *connectors*. Architectural notations may have an underlying semantic definition; given in either natural language or a formal logic.

Since we are considering the incorporation of contracts into architectural descriptions, we begin by identifying properties that architectural description notations should possess in order to support this (Section 4.1). An illustrative example is introduced in Section 4.2 and carried through Sections 4.3-4.8 which describe specific architectural notations. In Section 4.9, we draw general conclusions about the potential of architectural languages to support contracts and select notations on which to develop experimental contract extensions.

4.1 Criteria for Architecture Description

An aim in this work is to incorporate a contract language into an existing architectural description notation. Below we detail our criteria by which we judge the ability of architecture description languages and frameworks to support contract extension. The list below introduces the criteria with a description and justification for the choice of each criterion.

Semantic Strength The semantics of an architectural description language defines the meanings of constructs (such as components, connectors and assemblies). The term "semantic strength" refers to the way in which the semantics is defined. A weak semantic definition is one that is relatively imprecise, usually given in natural language, and which leaves the meanings of some constructions open to human interpretation. At the other extreme, a strong semantic definition states precisely the meaning of each construct with little or no ambiguity. The stronger a semantic definition, the greater the range of analyses that can be performed consistently and with machine support.

If a high degree of confidence is required in the verification of system-level properties, a strong semantics is required. In future work, we intend to formally define a contract language and provide a formal theory for (a subset of) non-functional properties. The semantic strength of architectural description language should be strong to support this task. However, a weaker semantics in the architectural description language does not preclude the use of contracts on the basis of guidelines – such contracts are simply analysed less formally, for example in accordance with guidelines rather than by means of an automatic tool (with the attendant risks of ambiguity). Indeed, for many cases, this may be a more attractive option than a fully formal approach.

Support for Non-Functional Properties Non-functional properties, discussed in Section 3, allow a system architect to specify properties about the system and its constituent elements that are not directly related to their functionality.

We wish to define interface contracts which allow us to analyse system properties which may be both functional and non-functional. The notation should support the representation of such FPs and NFPs - ideally supporting theories of NFP. As such properties are often application- or domain-specific, the notation should also allow NFP theories to be defined.

Extendability The extendability of a language denotes the ability of a language to accommodate application-specific extensions. The extendability should form part of the language definition and its semantics.

As stated earlier, our goal is to extend an existing architectural description language or framework with a contract language and the ability to represent and reason over non-functional properties. Allowing extensions in the semantics of the language or framework provides a mechanism for the incorporation of the interface contracts and also may aid in the compliance of the extension for existing users. If a language or framework does not have support for extensions, we must ensure that any additions we make are in keeping with the existing language definition.

Industrial Usage For our work to be of practical benefit, the architectural description method should ideally be in use by industry, or be at a readiness level such that it is viable for use in an industrial context. If this is not the case then clear guidance as to how our work may be incorporated into industry standard tools must be provided.

Tool Support Linked to the industrial usage, tool support for the chosen description language or framework will ensure the ability for system designers to easily define

system architectures. Such tools should be both available to industrial partners and ourselves in the support, development and demonstration of how the contract language may be used in context with systems engineering. Tools should be sufficiently robust and well enough supported to be used by industry.

Tool Support Extendability The extendability of tool support refers to the ease with which additional features can be added to a tool. Such extensions may be application-specific or may allow for additional analysis of the architectural model.

Extendable tool support would allow us, given language extensions, to provide formal verification within the tool support currently available for the given language/framework. This extension will allow us to perform additional analysis on the architectural model and interface contract.

External Analysis The external analysis of architectural models will allow us to perform analysis using tools external to the main tool support of the notation used. This external analysis may be invoked from current tool support, or may parse architectural models either developed using current tools or by hand if no current tools exist.

If extending existing tool support is not viable, analysis of architectural models in external tools is required. Support for such external analysis from within current tools would be most beneficial as it would allow current users of the notation to use tools they are familiar with.

Scope A notation for the description of an architecture may represent a single system – including its components and connectors. Similarly a notation may represent multiple systems – at a system-of-system level whereby many systems are combined to provide some capability. Alternatively both levels of abstraction may be supported.

There is significant interest in both the definition of system architectures and also in system-of-systems. The ability for a notation to support both levels of abstraction increases the ability for the definition of such interface contracts to be adopted at both levels. However there may be issues which may be raised at one level which may not appear in the other.

We review several significant notations for architectural description against these criteria. We have selected a range of languages from both "academic" and industrial sources. Acme, Darwin and Wright are three academic architecture description languages. Although a large number of ADLs have been developed, we chose these particular examples because of the volume of research work that accompanies them, and the formality of their language definitions. The UML and SysML design notations have been included because of their wide industrial usage. AADL has gained some industrial use.

In the remainder of this section, we provide further details of each approach. For each we provide a brief overview, demonstrate how the example presented in Section 4.2 may be defined and address the criteria described above.

4.2 Illustrative Example

In this section, we present a small example system in order to illustrate the different architectural notations, the concept of interface contracts and how the different notations may be extended to include contracts. This example has a layered architecture with a (very) abstract similarity to an IMS architecture. It consists of a single system the aim of which is to obtain sensed data from its environment. The system contains a number of subcomponent types and we are concerned with three: the *Application*, *Operating System* and *Hardware Sensor* types. The system communicates with its environment through the hardware sensor component, which is connected to the system boundary. Data obtained from the environment may then be used by other subcomponents.

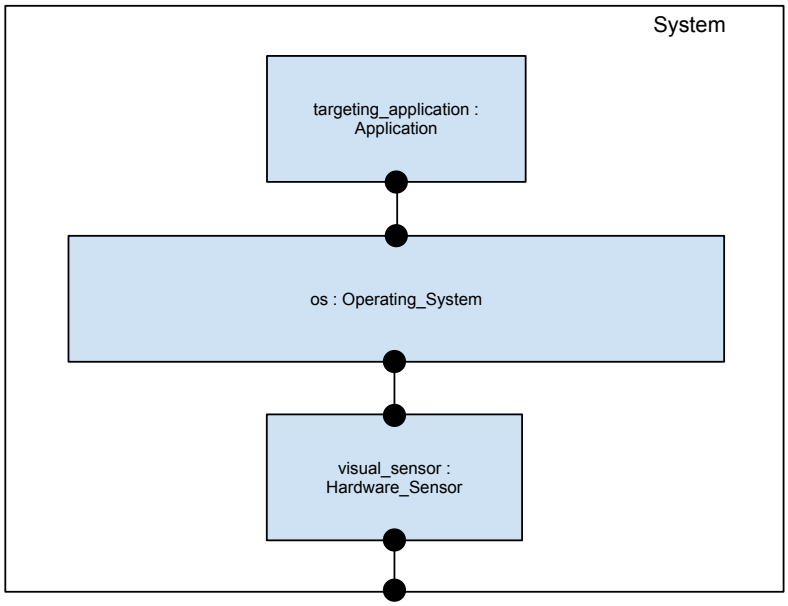


Figure 10: Example system with layered architecture

Figure 10 shows the abstract connections between the component instances. The *targeting_application* component of type *Application* requests visual information from the lower layers of the system, providing coordinates of the area to be monitored. The *os* component of type *Operating_System* acts as a relay in that it receives a request from the targeting application and passes it to the relevant underlying hardware-specific sensors. The OS subsequently returns the resultant visual data to the application. The *visual_sensor* component of type *Hardware_Sensor* acts as a driver for the hardware, interfacing with the environment of the software system. The component supplies coordinates and hardware commands and receives the relevant visual data.

4.3 Acme

4.3.1 Acme Overview

Acme[2] is an academic architecture description language. Initially developed as an interchange language, Acme is a generic language with little direct support for analysis of architectural models. An Acme model has the following entities to describe architectural structure: *component*, *connectors*, *ports roles* and *systems*. Components and connectors are attached using ports and roles. Hierarchies of components may be defined using representations. Acme models may be annotated with properties. These properties, consisting of a name, type and value, may be interpreted by the Armani constraint checker within the AcmeStudio toolset[3].

Acme models may be represented using a graphical or textual notation, a syntax is defined for the textual representation and diagrammatical rules are defined. Figure 11 depicts the example system, with the large boxes denoting components, small boxes on their edges the ports and the circles with arrows depicting the connectors and their roles. The *app* and *vis* components are further defined using representations, the Acme diagrams of these components are presented in Appendix A.1.

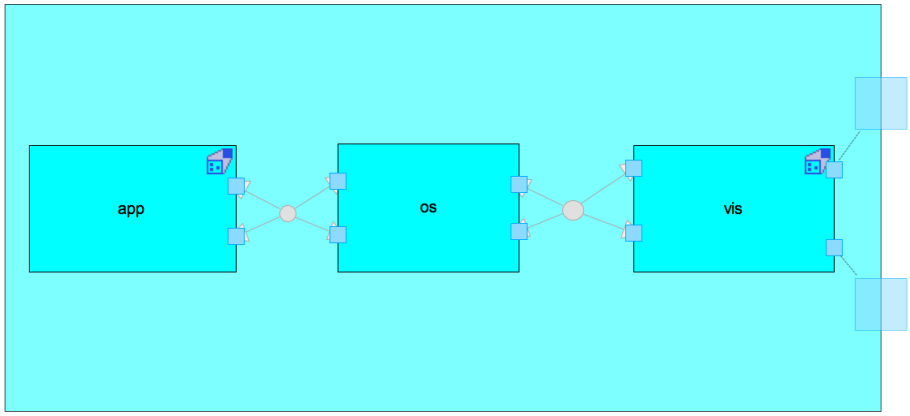


Figure 11: Acme graphical diagram of example system

Figure 12 presents an extract of the textual definition of the example system. In defining the example system, we have also defined an Acme *family* which defines the types of component, connector and other architectural entities. In Figure 12 the component *SystemExample* is of the *System* type, with the components and connectors making up the system component also of predefined types. As with the textual example, the *app* and *vis* components are further defined using the representation keyword. These are presented in Appendix A.2.

[2] `http://www.cs.cmu.edu/~acme/`
[3] `http://www.cs.cmu.edu/~acme/AcmeStudio/index.html`

```
Component SystemExample : System = new System extended with {
  Representation System_Rep = {
    Component os : Operating_System = new Operating_System {}
    Component app : Application = new Application extended with {
      Representation app_Rep = { ...}
    }
    Component vis : HW_Sensor = new HW_Sensor extended with {
      Representation visual_sensor_Rep = { ...}
    }
    Connector app-os : App_OS = new App_OS {}
    Connector os_hwd : OS_Hwd = new OS_Hwd {}

    Attachment app.response to app-os.caller_resp_in;
    Attachment os.resp_out to app-os.callee_resp_out;
    Attachment os.req_in to app-os.callee_req_in;
    Attachment app.request to app-os.caller_req_out;
    Attachment vis.resp_out to os_hwd.hwd_resp_out;
    Attachment vis.req_in to os_hwd.hwd_req_in;
    Attachment os.resp_in to os_hwd.os_resp_in;
    Attachment os.req_out to os_hwd.os_req_out;
  }
  Bindings {
    hardware_in to vis.cmd_out;
    hardware_out to vis.sense_in;
  }
}
```

Figure 12: Acme textual definition of System component. Full definition in Appendix A.2

4.3.2 Acme Applicability

Acme has a weak semantics. A basic 'open semantic framework' is defined which may be used to reason over basic structural abstractions. The values of properties do not have a defined semantics, so we may develop syntactic extensions for Acme properties to which we define our own semantics. Thus both contracts and non-functional properties may be incorporated into the Acme using external tools to interpret the properties.

As Acme is an academic tool, its level of industry use is low. However, since Acme is intended as a generic ADL and interchange language, the basic structural entities may be replicated from any ADL. Hence, cross-notation development is possible with industrial strength notations. Tool support is in the form of AcmeStudio – an Eclipse-based tool – which has benefited from continual academic development. AcmeStudio supports external analysis of Acme properties through the combination of Eclipse plugins and external tools.

Acme is intended as a software ADL. However, due to the abstract nature of component definition and the weak semantics, the notation allows both software system and SoS representation.

4.4 Darwin

4.4.1 Darwin Overview

The Darwin ADL [32] is concerned with the representation and analysis of structure and communication flow in distributed component-based systems. Darwin models are defined with either a declarative textual notation or a graphical representation, specifying basic components with the services they provide and require. Components are bound, linking services provided and required to form composite components and systems.

Figure 13 depicts the example from Section 4.2 in Darwin's graphical notation. The components are denoted by boxes, the ports by circles. Provided ports are shown as filled circles, required ports as empty circles.

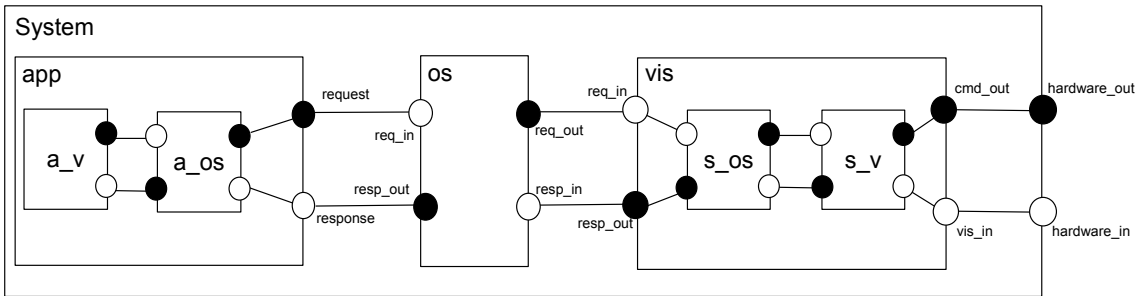


Figure 13: Darwin graphical diagram of example system

The definition of the system component is given in Figure 14 with the full textual definition given in Appendix B. The system component provides a *hardware_out* service

and requires a *hardware_in* service. The system component is a composite component containing three component instances: *app*, *os* and *vis*. The components are bound together and to the system ports. Notice from both textual and graphical notations, there is no provision for the representation of properties.

```
component System{
  provide hardware_out;
  require hardware_in;
  inst app: Application;
        os : Operating_System;
        vis : Hardware_Sensor;
  bind app.request -- os.req_in;
        os.resp_in -- app.response;
        os.req_out -- vis.req_in;
        vis.resp_out -- os.resp_in;
        hardware_out -- vis.cmd_out;
        hardware_in -- vis.sense_in;
}
```

Figure 14: Darwin textual definition of System component. Full definition in Appendix B

Darwin has been given an operational semantics in the π-calculus, providing explicit meaning to the architectural abstractions of the language. In [32] the π-calculus semantic definition is introduced for the Darwin syntactic constructs for required and provided services, and the bindings. The authors then expand this to include hierarchical and composite components.

4.4.2 Darwin Applicability

The formal semantics defined in the π-calculus gives the language a strong semantic basis, opening up the possibility of formal verification of structural properties of an architectural model. As Darwin does not allow the representation of non-functional properties of systems, components or connectors, formal verification of system-level properties is not supported.

The Darwin language has no support for extendibility – thus any additional syntactic features of the language required for the support of interface contracts must incorporate and extend the underlying semantics of the language. The tool support provided for Darwin is limited to a Java-based compiler and development appears to have halted. Tools extendibility is therefore low. We have found little or no record of industrial use.

As with Acme, Darwin is a software ADL – however, as the semantics of Darwin models component ports as software processes, the notation does not support SoS.

4.5 Wright

4.5.1 Wright Overview

Wright [33] is a formal software architecture description language, again from an academic background. The Wright ADL places its emphasis on the formal specification of a system architecture – in particular the semantics of connectors. As with Acme, Wright has three structural entities: a *system* is composed of *components* and *connectors*. A component is defined in terms of its ports and computation, connectors in terms of roles and a *glue* protocol.

Together with the architectural abstractions, Wright uses the formally defined mathematical process algebra CSP to define the semantics of architectural models. Wright is a textual notation, employing a subset of the CSP process algebra in the definition of components, connectors and the glue which joins the two to form a protocol of component interaction.

Figure 15 depicts the example system in the Wright notation. The definition of the Application component type states that there are two ports and the expected events and data flows between the component and its environment. Defined in CSP, the request port states that the protocol of interaction is a series of *appReq* events, where data is output, denoted by $appReq!x$. The response port denotes input of data given the *appResp* event. The computation of the component states that there is an interleaved series of the *appReq* and *appResp* events of the request and response ports respectively.

The *A-OS_Conn* connector definition of the example system is similar to that of the *Application* component. However, notice the *glue* contains the deterministic choice between a series of events and the $SKIP$ process which denotes successful termination. The remainder of the example shows the definition of the component and connector types, followed by combining the ports and roles of the components and connectors of the system.

Wright is primarily concerned with detailing the semantics of communication and checking properties of connections, for example deadlocks. As with the Darwin language, Wright does not support modelling of component properties or interfaces. Hierarchical systems are not supported in Wright.

4.5.2 Wright Applicability

Wright has an inherently strong semantics since models are defined using a formally defined mathematical process algebra. Wright models may, therefore, be formally verified using the FDR model checker[4] for connection-related and computational-related properties – namely deadlock. In spite of this, Wright does not appear to enjoy industry use.

Tool support is available for the Wright ADL but is far from industrial readiness. A command-line interface is provided for a tool which may parse Wright models, translate models to Acme and to the FDR model checker for formal verification.

[4] http://www.fsel.com/software.html

```
System Example
  Component Application
    Port request = appReq!x -> request
    Port response = appResp?x -> response
    Computation request.appReq -> response.appResp -> Computation
  Component Operating_System ...
  Component Hardware_Sensor ...

  Connector A-OS_Conn
    Role AppMakeRequest = appReq?x -> AppMakeRequest
    Role OSRecRequest = osReqIn!x -> OSRecRequest
    Glue AppMakeRequest.appReq?x -> OSRecRequest.osReqIn!x -> Glue
          [] SKIP
  Connector OS-A_Conn ...
  Connector OS-HWD_Conn ...
  Connector HWD-OS_Conn ...

Instances
  app : Application
  os : Operating_System
  vis : Hardware_Sensor
  a-os : A-OS_Conn
  os-a : OS-A_Conn
  os-hwd : OS-HWD_Conn
  hwd-os : HWD-OS_Conn

Attachments
app.request as a-os.AppMakeRequest;
app.response as os-a.AppRecResp;
os.req_in as a-os.OSRecRequest;
os.req_out as os-hwd.OSSendReq;
...
end Example
```

Figure 15: Wright definition of System component. Full definition in Appendix C

The Wright language has no built-in facilities for extensions and, like Darwin, definition of properties is not supported. However, as seen in Section 3.2, we could envisage extending the language – any additional syntactic features such as contracts and properties must extend the underlying semantics of the language. If this were done, tool support would also need to be extended to parse the new features.

Like Darwin, the use of a process algebra to specify component and connector semantics limits Wright to software systems.

4.6 UML 2.0

4.6.1 UML Overview

The Unified Modelling Language (UML) [34, 35] aims to provide a general modelling framework for the design of software systems. UML models consist of diagrams describing aspects of a software system from the point of view of the different stakeholders. UML consists of two parts: an infrastructure [34] and a superstructure [35]. The infrastructure (or meta-metamodel) describes the core metamodel upon which the superstructure is based. The meta-metamodel defines the entities used in the UML diagrams defined in the superstructure. The diagram definition in the superstructure defines notations and semantics of the diagrams which make up the UML.

The diagrams of UML fall into two categories: *Structure* and *Behaviour*. The *component* diagram – as with the academic ADLs – describes how components are connected to form software systems. Figure 16 depicts the illustrative example as a UML component diagram.

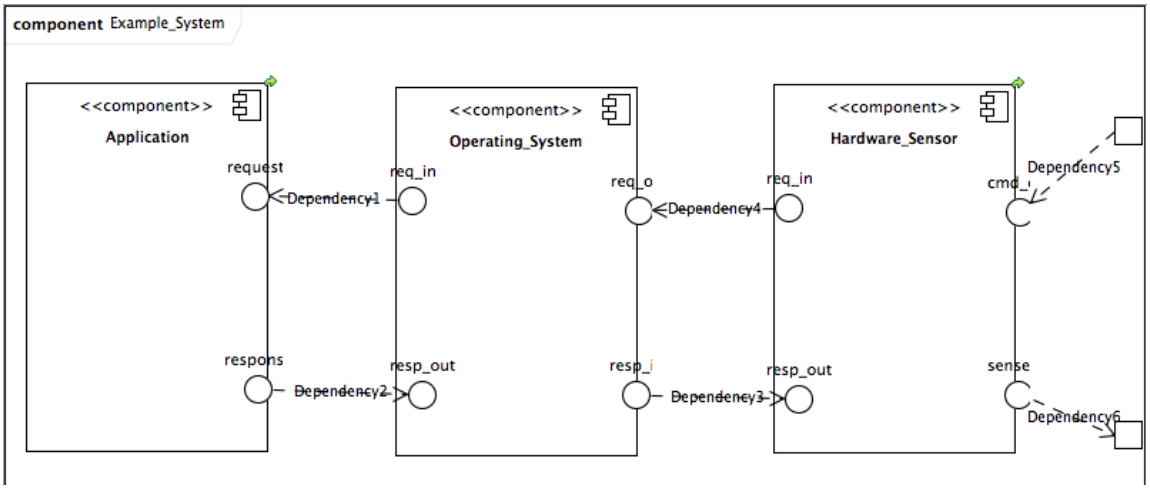


Figure 16: UML component diagram of example system

The component diagram allows components to be modelled and interfaces defined[5].

[5]The Topcased tool used to create Figure 16 does not differentiate between required and provided

Dependency links define the relationship between required and provided interface – required interfaces depend upon provided interfaces. Components may be further decomposed using additional component diagrams (the *Application* and *Hardware_Sensor* components are defined in component diagrams in Appendix D). Component diagrams do not support property definition, however modellers could use other diagram types such as class diagrams to capture this information. However, due to the relatively weak semantics of UML, there may not be concrete links between these diagrams.

As UML has been designed as being a generic modelling language, there are no application- or domain-specific aspects to it. UML does, however, support the use of stereotypes and profiles to extend the language. Profiles use a subset of the UML metamodel, stereotypes and a natural language semantics to define such extensions. One example of which, SysML, is addressed in Section 4.7.

4.6.2 UML Applicability

The semantics of UML diagrams, given in [35], are specified in natural language. This weak semantics requires tool vendors to encode the meaning of the diagrams in their software tools, and there is therefore a risk of inconsistency and incompatibility. There have, however, been a number of initiatives – independent of the OMG standard – to provide a more precise semantic language semantics.

The Precise UML group[6] have attempted to formally define UML and to map UML diagrams to formal notations – however, UML 1.0 was used in this research and the group no longer focus solely on UML. Executable UML [36] (xUML) has an execution semantics defined for a subset of the UML language including the class and statechart diagrams along with an action language. Formal treatments of xUML have been attempted, Hansen et al. translate a subset of xUML into the mCRL2 process algebra [37] and Turner et al. translate a subset of xUML to CSP||B (a notation combining the CSP process algebra and the formal modelling notation B) [38]. Modellers may therefore consider using a subset of UML (metamodel and diagrams) and extend the relevant parts to include contracts through the use of UML profiles and stereotypes. These syntactic extensions may be given a formal semantics for formal verification and analysis.

UML has become almost 'de rigueur' in industry, due to its generality, extensibility and strong tool support. Several commercial tools are available from vendors such as IBM, Sparx and No Magic. Open source tools are in development based on the Eclipse framework including Papyrus and Topcased. Commercial and open source tools may be extended to reflect language extensions in the form of stereotypes and profiles. UML models may also be serialised as XMI files, which enables exporting of models using an abstract syntax for external analysis. A thorough analysis of commercial tools has not been performed. From our initial investigations, the Topcased tool appears to be the further developed and a more mature open source tool (the example presented is this document was prepared using the Topcased tool).

interfaces.

[6] http://www.cs.york.ac.uk/puml/

4.7 SysML

4.7.1 SysML Overview

SysML [39] is a modelling language for system specification in systems engineering. The focus on system engineering allows for the representation of *systems*, *hardware*, *software*, *information* and *processes*. This expands on the software-oriented focus of UML. SysML uses a subset of UML 2.0 and provides further extensions to the UML superstructure in the form of a UML profile.

Like UML, SysML has a number of diagrams defined in the meta-model. These describe the system being modelled, and fall into broad categories of *Behaviour*, *Requirement* and *Structure* diagrams. The structure diagrams are those of most interest to architectural description. SysML is defined using UML modelling techniques (i.e. diagrammatically) with the use of 'precise natural language' semantics, however the specification document [39] does suggest that future versions may feature a formal semantics.

The *structural block diagrams* describe the relationships between components - *block definition diagrams* describe the relationships in terms of associations, generalisations and dependencies, and *internal block diagrams* define the internal structure of a block in terms of references to other blocks and ports.

Figure 18 depicts an extract of the illustrative example defined using the block definition diagram. The System block contains *Application*, *Operating_System* and $Hardware\_Sensor$ blocks which are in turn are composed of a number of blocks. As this figure is not intended to fully define the example system, we do not fully specify properties and operations of the individual blocks, however, the three blocks which compose the system do have top level operations defined. The types $OSPacket$ and $HWPacket$ given as parameter types are system specific types defined in SysML.

The example is elaborated further in the internal block diagram of the System block, shown in Figure 18. This diagram expands on the block definition by 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.

Constraints may be added to blocks for analysis and constraint of properties of a system. SysML identifies and names such constraints, however does not specify a computer interpretable language for their representation. Interpretation must be provided by system engineers.

4.7.2 SysML Applicability

As it is based on UML 2.0, SysML has a weak semantics. As with UML the emphasis is on tool vendors to encode the semantics in their tools. Being based on UML allows SysML to be extended through the use of stereotypes and profiles. As SysML is system- rather than software-oriented, the notation is more clearly suited to the use of interface contracts in system-of-system applications.

SysML is becoming widely used in industry due to UML tool vendors such as Sparx Systems and IBM providing support for the SysML profile in their UML 2.0 tools. The SysML standard received contributions across industry, US government

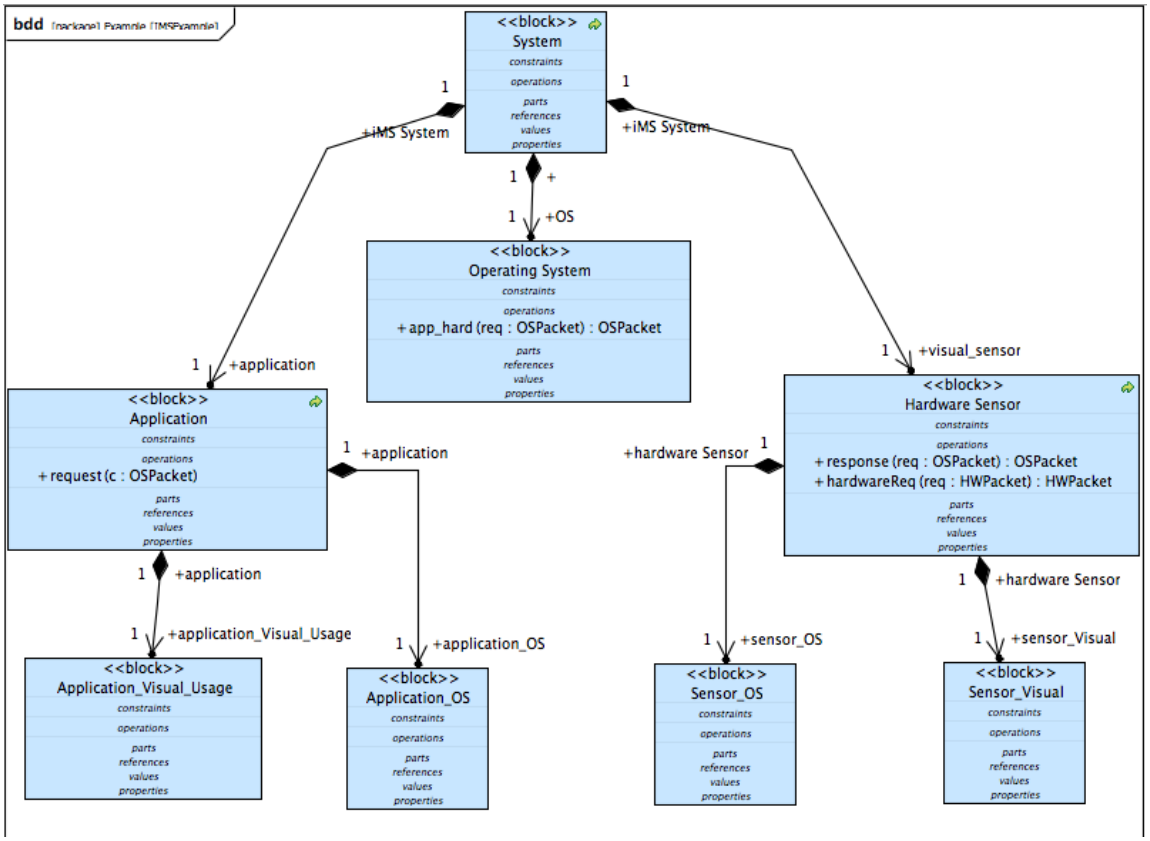


Figure 17: SysML block definition diagram of example

and tool vendors. A number of commercial and open source UML tools, including those mentioned in Section 4.6, offer support for SysML through the compliance of the SysML profile. Our findings on the tool support for UML also apply for the SysML profile.

Designed as an extension to UML 2.0 to represent system-level development, SysML supports both system and system-of-system architectural models – both software and hardware.

4.8 AADL

4.8.1 AADL Overview

The Architecture Analysis and Design Language (AADL) [40, 41] is an ADL designed to describe complex real time and embedded systems. The language was formerly known as the Avionics Architecture Description Language and as such had an avionic domain bias. However, in its current form AADL is not domain-specific and allows

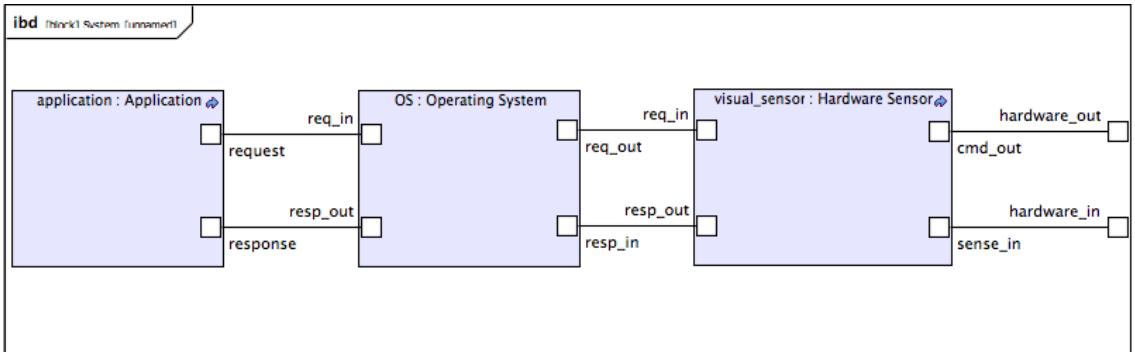


Figure 18: SysML internal definition diagram of example

modelling at both the system and system-of-systems level. AADL is derived from the MetaH ADL and was developed by both industrial and academic organisations.

AADL models may be represented graphically, textually or as XML. The main elements of AADL consist of *component types*, *component implementations*, *property sets*, *packages* and *annexes*.

Three categories of components – *software*, *hardware* and *composite* – provide different abstraction levels in AADL models. Software components may be either a *Thread*, *Thread Group*, *Process* or *Data*. Hardware components comprise a *Processor*, *Memory*, *Device* or *Bus*. Finally, the composite component is a *System*.

Figure 19 depicts the *example_system* instance *example* defined using AADL. The three subcomponents of the system are defined as instances of software processes. The processes are connected by named data ports. Each process may be further decomposed into software threads, Figure 20 depicts the *application* software process *ta*. Separate diagrams depict the types as defined in AADL and also each of the processes, which are presented in Appendix F.1.

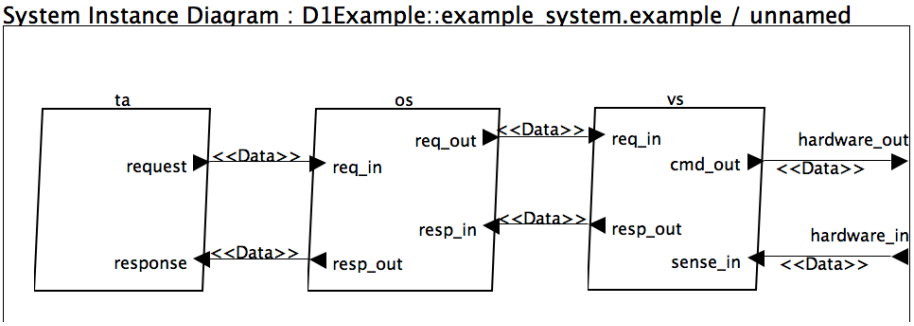


Figure 19: AADL graphical diagram of example system

The textual definition of the example system, presented in full in Appendix F.2,

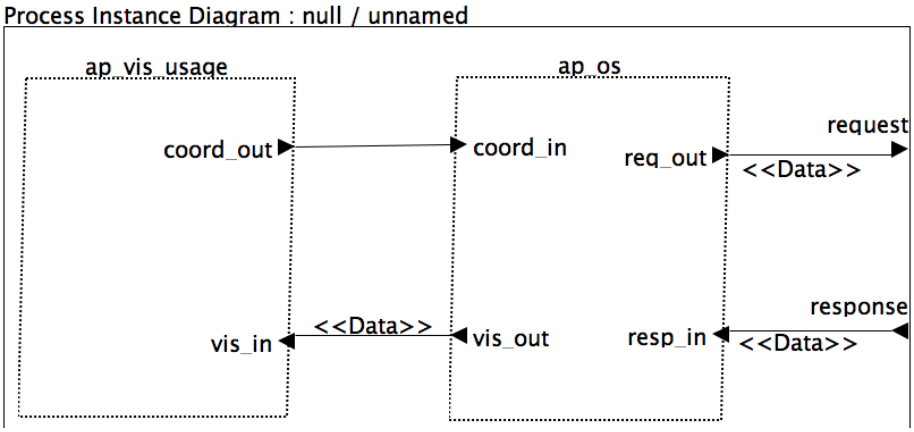


Figure 20: AADL graphical diagram of application process

requires the definition of component types and their implementation. The textual representation of the *example* component – discussed earlier – is shown in Figure 21.

```
system implementation example_system.example
  subcomponents
    ta: process application.targeting_application;
    os: process operating_system.os;
    as: process hardware_sensor.visual_sensor;
  connections
    data port ta.request -> os.req_in;
    data port os.resp_out -> ta.response;
    data port os.req_out -> as.req_in;
    data port as.resp_out -> os.resp_in;
    data port as.cmd_out -> hardware_out;
    data port hardware_in -> as.sense_in;
end example_system.example;
```

Figure 21: AADL textual definition of System component. Full definition in Appendix F.2

AADL has a set of standard types, including boolean, real, integer and enumerated lists. A large number of pre-defined functional and non-functional properties exist in the AADL standard. AADL also has the capability for system modellers to define system-specific properties using *property sets*. We are not aware of a formal theory for properties in AADL.

Extensions to the AADL notation may be developed through the use of *annexes*. To

date, four annexes have been approved and added to the AADL standard[7], including an error model annex and an annex for XML/XMI interchange. Annexes describe sublanguages – analysis-specific notations that may be associated with components types and implementations in an AADL system definition.

4.8.2 AADL Applicability

Although not formally defined, the semantics of AADL is described using a precise execution and communication semantics for the analysis of embedded execution properties of AADL models, achieved though hybrid automata descriptions of software threads.

As mentioned, AADL has support for extending the core language to allow for both application-specific and new notations in the form of annexes. There is a large community for the support of AADL and guidance is available to aid in annex development. Alongside the predefined properties supported by AADL, it allows the definition of property sets in a system architecture and annexes for project- and domain-specific properties.

MetaH, the forerunner of AADL was developed at Honeywell Technology Centre, under sponsorship of DARPA and US Army Aviation and Missile Command [40]. A number of defence and aviation companies have investigated the use of AADL for architectural modelling and analysis[8], however the use of the notation is not pervasive. The tool support – in the form of the open source OSATE[9] – is a mature tool, which may feasibly be widely used. Due to its open source nature and the ability to extend the language using annexes, extensions to tool support may be developed. Guidance also exits on tool extension.

AADL models may define systems containing software and hardware components, and the notation may also support system-of-systems.

4.9 Architecture Description Assessment Conclusions

Table 1 summarises the characteristics of the languages against the criteria described in Section 4.1.

Based on the criteria and assessment in this Section, we make some general observations on the state of the art in architecture specification. Semantically strong architecture description languages such as Darwin and Wright enable machine-assisted formal verification but appear to have weak tool support and are limited to academic use. Semantically weaker notations lack automated verification, but appear to have more robust and extensive tool support, allow application and domain-specific extensions and tend to have stronger records of use in both academia (Acme, AADL) and industry (UML, SysML).

Given the findings of this study, we feel that AADL and SysML are candidates for extending with an interface contract specification language. The two notations boast the ability for language extension, representation of properties, strong extendable open-source tool support (and extendable commercial tools for SysML) and also support

[7] http://www.aadl.info/aadl/currentsite/start/about-standard.html
[8] http://www.aadl.info/aadl/currentsite/start/who.html
[9] http://www.aadl.info

Language/ Framework	**Criteria**						
	Semantics	Extendability	NFP Support	Industrial	Tool Support	TS Extend./ Ext. Analysis	Scope
Acme	Weak	Properties	None-interpreted properties	Low	Academic - semi-mature	Eclipse plugins and ext. analysis	Software and SoS
Darwin	Formal - π calculus	None	None	Low	Academic - poor	None	Software
Wright	Formal - CSP	None	None	Low	Academic - poor	None	Software
UML 2.0	Natural language - weak	Stereotypes and profiles	User-defined and profiles	High	Commercial - mature, Open source - semi-mature	Commercial - ext. analysis (XMI model), extendable Open source - Eclipse plugins	Software
SysML	Natural language - weak	Stereotypes and profiles	User-defined and profiles	High	Commercial - mature, Open source - semi-mature	Commercial - ext. analysis (XMI model), extendable Open source - Eclipse plugins	System and SoS
AADL	Execution semantics	Annexes	Built-in properties and Annex support	Low	Open source - mature	Open source - Eclipse plugins	System and SoS

Table 1: Summary of architecture description approaches in relation to criteria

model definition at the system and system-of-system level. Although the use of AADL in industry is not widespread, the notation does have a precise execution semantics. SysML, as a standardised profile of UML, has stronger industrial support. Although the language has only weak semantic strength, formal treatments of UML-based notations (Section 4.6) indicate the suitability of applying interface contracts to a formally defined subset of SysML. Given the aims and potential benefits we envisage, and the desire of industry application, we propose in future work to define contract extensions to SysML.

Using the illustrative example in Section 4.2, we envisage how contracts may be described by extending SysML. The interface contract language definition is a matter for the next work package and thus the syntax used in the contract definition in this Section should be taken as a means to demonstrate an interface contract specification in AADL. The approach we propose for incorporating interface contracts in SysML is to first identify a subset of SysML relevant to contracts. In Section 4.7, we suggested that the structural block diagrams are of greatest relevance to this task. Given this, we propose extending a subset of SysML – namely those aspects related to the block definition diagram and internal block diagram.

As an example of this extension, we propose a SysML profile, *SysMLContract*, as depicted in Figure 22. The profile extends an existing element of the SysML notation – referred to as a *metaclass* – with a *stereotype*. In the illustrative example, the *Property* metaclass of the metamodel is extended to include a new syntactic entity *ContractProperty*.

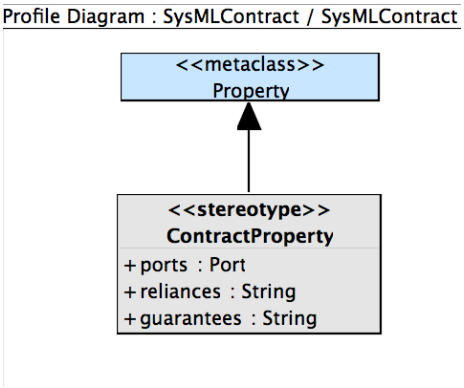


Figure 22: Example SysML profile – extending the property metaclass to define an interface contract

The ContractProperty stereotype has three elements - *ports*, *reliances* and *guarantees* – with port identifier and string types. Given this syntactic extension, consider the illustrative example defined in Section 4.2 and illustrated using SysML in Section 4.7. Applying the SysMLContract profile to the original model definition, we extend the *Application* component in the block definition diagram by defining the property *interface_contract* defined as the stereotype <<*ContractProperty*>>. An extract of the

extended block definition diagram is depicted in Figure 23.

The system may be further defined in the internal block diagram as shown in Figure 24. The interface_contract ContractProperty is depicted with the defined values within the property block. The contract states which component ports are associated with the contract, followed by properties which the component relies upon and those the component guarantees. The reliance expression states that the output of the request must be a member of the instance variable set *reachableCoords*. The guarantee states that the response image must have a resolution greater than 6 (assuming the use of the National Imagery Interpretability Scale (NIIRS) [42]) and the response time of the response will be less than 500ms. Notice that a mixture of functional properties (for example the guarantee) and non-functional properties (in the reliances) within the contract.

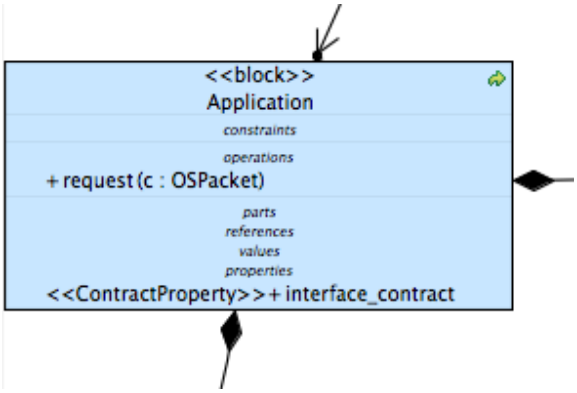


Figure 23: Extract of SysML block definition diagram

As SysML and AADL use components and ports for the individual elements of computation and communication, a future direction of this task may consider and provide guidance as to how the concepts and language definition of interface contracts defined in SysML notation may be incorporated into AADL.

5 Future Work and Conclusions

We have reviewed aspects of the state of the art in architectural modelling, contract specificaton and the description of non-functional properties. We have identified the importance of representing non-functional properties in interface contracts. However, our findings in Section 3.4 have also identified the fact that the rules of composition and the identification of NFPs in SoS and systems is a complex task.

Given our findings in Section 2, we aim in future work to evaluate the potential for contract-based specification in SoS by developing a language to support interface contracts and the definition of theories of functional and non-functional properties. This language incorporates the principles of design-by-contract alongside rely/guarantee on ports of systems composing system-of-systems. Based on our findings in Section 3.4,

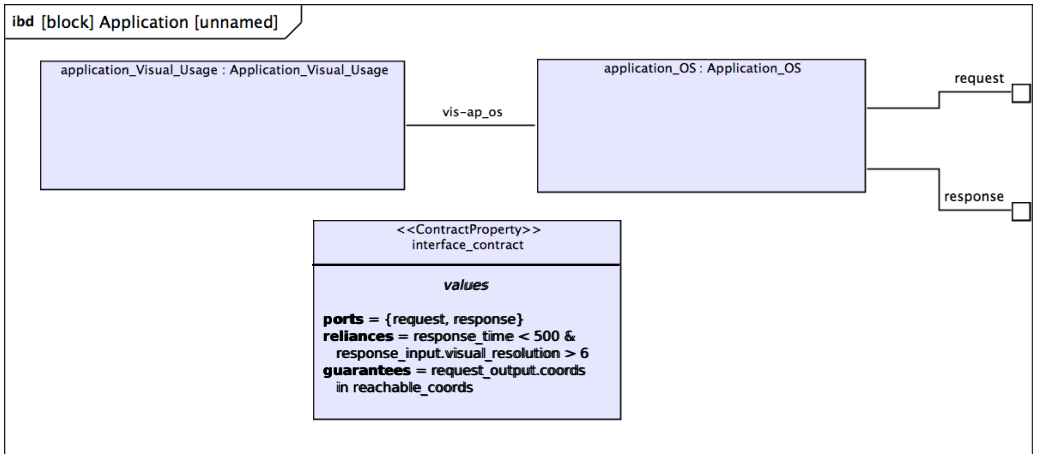


Figure 24: SysML internal definition diagram of example with interface contracts

we note that non-functional properties are of great importance. An interface contract language should enable a SoS designer to state NFPs in contracts. We propose to carry out a proof-of-concept case study in which we assess the contract language. Below we highlight our requirements for a case study to achieve these aims.

- **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 systems-of-systems. This ensures the results of any assessments are deemed realistic and that the contract language will scale to such real world SoS.
- **Clear system-of-system 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, a number of different properties of different underlying types (e.g. numeric, boolean, collections).
- **Composition of system properties** - Properties introduced in interface contracts of systems may be composed in an architecture. The result of composition may differ between property types as has been stated in Section 3. A suitable case study should make have a number of properties with differing composition rules.
- **Verification of SoS-level properties** - The proposed exploitation of the interface contracts in future work includes the analysis of SoS-level properties. The proof-of-concept should identify such SoS-level properties to demonstrate this exploitation path.

Acknowledgements

The work described here was funded under the UK Software Systems Engineering Initiative (SSEI)[10] within Task 16 on *Interface Contracts for Architectural Specification and Assessment.*

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

6 List of References

References

[1] Ministry of Defence. Defence technology strategy. `http://www.science.mod.uk/strategy/dts.aspx`, 2005.

[2] Ministry of Defence. Defence industrial strategy. `http://www.science.mod.uk/strategy/dis.aspx`, 2005.

[3] Ministry of Defence. Innovation strategy. `http://www.science.mod.uk/strategy/inno_strat.aspx`, 2005.

[4] Ministry of Defence. Technology partnership in defence. `http://www.science.mod.uk/strategy/technology_partnership.aspx`, 2005.

[5] John K. Bergey, Jr. Stephen Blanchette, Paul C. Clements, Michael J. Gagliardi, Rob Wojcik, William G. Wood, and John Klein. U.S. army workshop on exploring enterprise, system of systems, system, and software architectures. Technical Report CMU/SEI-2009-TR-008, Carnegie Mellon Software Engineering Institute, 2009.

[6] Martin Hall-May. *Ensuring Safety of Systems of Systems A Policy-based Approach.* PhD thesis, Department of Computer Science, University of York, UK., 2007.

[7] A. Sousa-Poza, S. Kovacic, and C. Keating. System of systems engineering: an emerging multidiscipline. *Intl. Journal of Systems Engineering*, 1(1,2), 2008.

[8] M.W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[9] B. Sauser and J. Boardman. Taking hold of system of systems management. *Engineering Management Journal*, 20(4), 2008.

[10] R. S. Kalawsky. Grand challenges for systems engineering research. In *Proc. 7th. Intl. Conf. on Systems Engineering Research*, April 2009.

[11] Bertand Meyer. *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, 1988.

[12] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

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

[10] `http://ssei.org.uk/`

[14] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332, 1983.

[15] Hong Qing Yu and Stephan Reiff-Marganiec. Non-functional property based service selection: A survey and classification of approaches. In *Non Functional Properties and Service Level Agreements in Service Oriented Computing Workshop (NFPSOA)*, volume 411. CEUR-Proceedings, 2008.

[16] Object Management Group. Handling non-functional properties in a service oriented architecture request for information. http://www.omg.org/cgi-bin/doc?mars/2008-12-26, December 2008.

[17] Matthias Galster and Eva Bucherer. A taxonomy for identifying and specifying non-functional requirements in service-oriented development. In *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*, pages 345–352, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Ioannis V. Papaioannou, Dimitrios T. Tsesmetzis, Ioanna G. Roussaki, and Miltiades E. Anagnostou. A QoS ontology language for web-services. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 101–106, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Stephan Reiff-Marganiec, Hong Qing Yu, and Marcel Tilly. Service selection based on non-functional properties. In *International Conference on Service Oriented Computing Workshops*, pages 128–138. Springer, 2007.

[20] Christopher Van Eenoo, Osama Hylooz, and Khaled M. Khan. Addressing non-functional properties in software architecture using ADL. In *Sixth Australasian Workshop on Software and System Architectures*, pages 6–12, 2005.

[21] Xavier Franch. Systematic formulation of non-functional characteristics of software. In *Proc. 3rd Int'l Conf. on Requirements Engineering*, pages 174–181. IEEE Computer Society, 1998.

[22] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 2009.

[23] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[24] Mohamed Kaâniche. Resilience evaluation with regards to accidental and malicious threats. In *Proceedings of the ReSIST Summer School*, pages 213–260, 2007.

[25] Carl Gamble and Steve Riddle. Dependability metadata acquisition and assessment: A state of the art survey. Technical Report CS-TR-1232, School of Computing Science, Newcastle University, Newcastle upon Tyne, UK, 2010.

[26] John S. Fitzgerald, Simon Tjell, Peter Gorm Larsen, and Marcel Verhoef. Validation support for distributed real-time embedded systems in VDM++. *IEEE International Symposium on High-Assurance Systems Engineering*, 0:331–340, 2007.

[27] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag, 2005.

[28] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[29] Phillipa Conmy and Tim Kelly. Classifying and understanding safety dependencies in integrated modular avionics. In *25th International System Safety Conference (ISSC '07)*, August 2007.

[30] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[31] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, Janruary 2007.

[32] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[33] Robert Allen and David Garlan. A formal basis for architectural connection. *IEEE Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[34] Object Management Group. OMG Unified Modelling Language infrastructure version 2.2. http://www.omg.org/spec/UML/2.2/Infrastructure, February 2009.

[35] Object Management Group. OMG Unified Modelling Language superstructure version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure, February 2009.

[36] Stephen J. Mellor and Marc J. Balcer. *Executable UML, A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[37] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, MohammadReza Mousavi, and Jaco van de Pol. Towards model checking executable UML specifications in mCRL2. In *Proceedings of the 2nd IEEE International workshop UML and Formal Methods (UML&FM 2009)*, 2009.

[38] Edward Turner, Helen Treharne, Steve Schneider, and Neil Evans. Automatic generation of CSP || B skeletons from xUML models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 364–379. Springer-Verlag, 2008.

[39] Object Management Group. OMG Systems Modelling Language version 1.2. http://www.omg.org/spec/SysML/1.2, June 2010.

[40] SAE Aerospace. Architecture Analysis and Design Language AADL. SAE Standard: AS5506, November 2004.

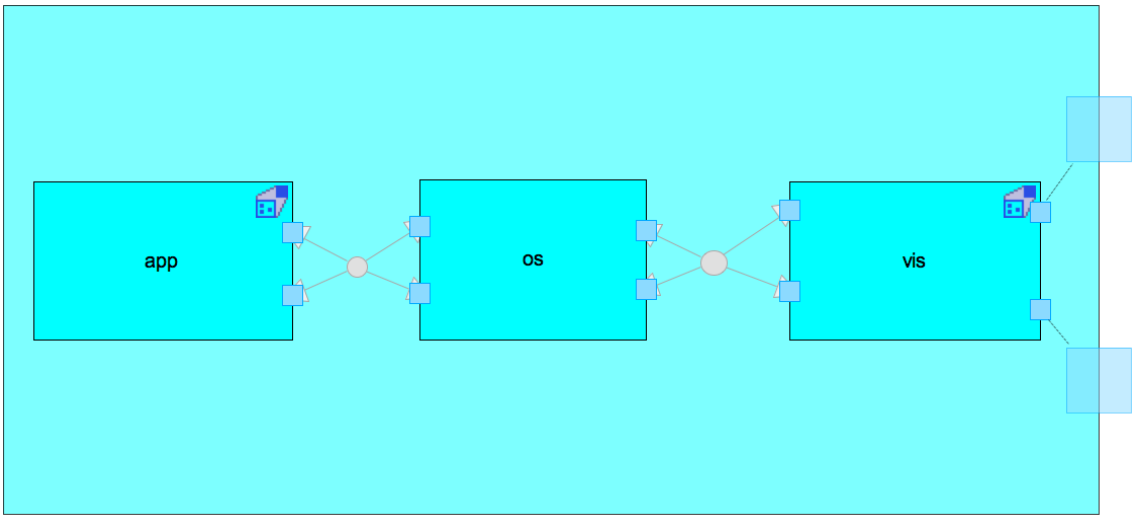
[41] Peter Feiler, David Gluch, and John Hudak. The architecture analysis and design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, 2006.

[42] Ian Moir and Allan G. Seabridge. *Military Avionics Systems*. John Wiley & Sons, ltd, 2006.

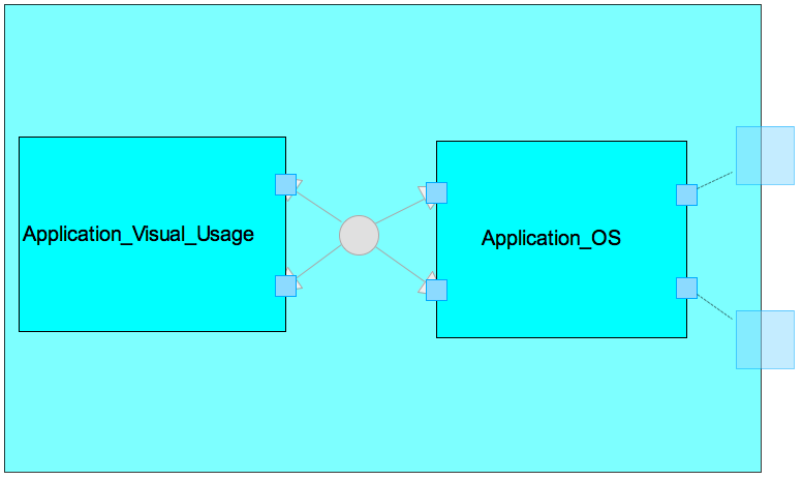
A Acme Example

A.1 Graphical Representation

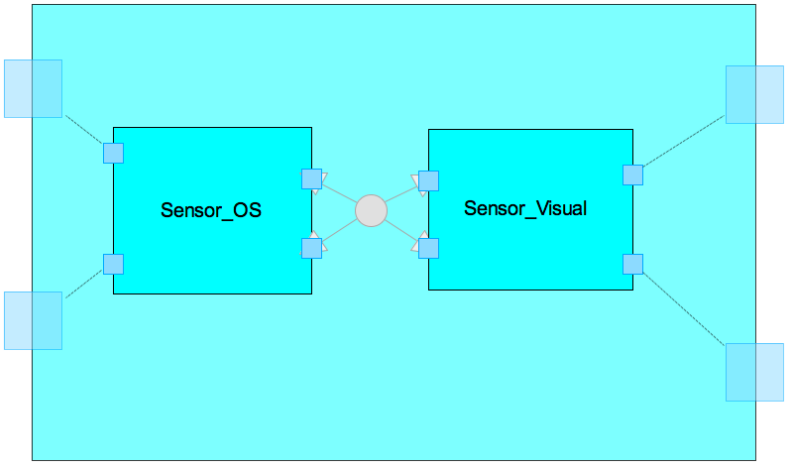
A.1.1 SystemExample Component



A.1.2 Application Component



A.1.3 Hardware Sensor Component



A.2 Textual Representation

```
import families/Example.acme;

System example : Example = new Example extended with {

  Component SystemExample : System = new System extended
                                                          with {
    Representation System_Rep = {

      Component os : Operating_System = new Operating_System  {}

      Component app : Application = new Application extended
                                                          with {
        Representation app_Rep = {
          Component Application_OS  = {
            Port vis_out ;
            Port req_out;
            Port resp_in;
            Port coord_out;
          }
          Component Application_Visual_Usage  = {
            Port coord_out;
            Port vis_in;
          }
          Connector int_app_os  = {
            Roles role1, role3, role4, role0;
          }
          Attachment Application_Visual_Usage.coord_out to
```

```
                                        int_app_os.role0;
    Attachment Application_OS.vis_out to int_app_os.role4;
    Attachment Application_Visual_Usage.vis_in to
                                        int_app_os.role3;
    Attachment Application_OS.coord_out to
                                        int_app_os.role1;
  }

  Bindings {
      request to Application_OS.req_out;
      response to Application_OS.resp_in;
  }
}

Component vis : Hardware_Sensor = new Hardware_Sensor
                                        extended with {
  Representation visual_sensor_Rep = {
    Component Sensor_OS  = {
      Port req_in;
      Port resp_out;
      Port coord_out;
      Port vis_in;
    }
    Component Sensor_Visual  = {
      Port Port2;
      Port Port3;
      Port coords_out;
      Port vis_in;
    }
    Connector Connector0  = {
      Roles role0, role1, role3, role2;
    }
    Attachment Sensor_Visual.Port3 to Connector0.role1;
    Attachment Sensor_Visual.Port2 to Connector0.role0;
    Attachment Sensor_OS.vis_in to Connector0.role3;
    Attachment Sensor_OS.coord_out to Connector0.role2;
  }

  Bindings {
    sense_in to Sensor_Visual.coords_out;
    cmd_out to Sensor_Visual.vis_in;
    req_in to Sensor_OS.req_in;
    resp_out to Sensor_OS.resp_out;
  }
}
Connector app-os : App_OS = new App_OS {}
```

```
      Connector os_hwd : OS_Hwd = new OS_Hwd {}

       Attachment app.response to app-os.caller_resp_in;
       Attachment os.resp_out to app-os.callee_resp_out;
       Attachment os.req_in to app-os.callee_req_in;
       Attachment app.request to app-os.caller_req_out;
       Attachment vis.resp_out to os_hwd.hwd_resp_out;
       Attachment vis.req_in to os_hwd.hwd_req_in;
       Attachment os.resp_in to os_hwd.os_resp_in;
       Attachment os.req_out to os_hwd.os_req_out;
      }

      Bindings {
        hardware_in to vis.cmd_out;
        hardware_out to vis.sense_in;
      }
    }
}
```

B Darwin Example

```
component Application_Visual_Usage{
    provide coord_out;
    require vis_in;
}

component Application_OS{
    provide vis_out;
            req_out;
    require coords_in;
            resp_in;
}

component Application{
    provide request;
    require response;
    inst a_v : Application_Visual_Usage;
         a_os: Application_OS;
    bind a_v.coord_out -- a_os.coords.in;
         a_os.vis_out -- a_v.vis_in;
         a_os.req_out -- request;
         response -- a_os.resp_in;
}


component Operating_System{
    provide resp_out;
            req_out;
    require resp_in;
            req_in;
}


component Sensor_OS{
    provide coord_out;
            resp_out;
    require req_in;
            vis_in;
}

component Sensor_Visual{
    provide coord_out;
            vis_out;
    require coord_in;
            vis_in;
```

```
}

component Hardware_Sensor{
    provide cmd_out;
            resp_out;
    require req_in;
            sense_in;
    inst s_os: Sensor_OS;
         s_v : Sensor_Visual;
    bind req_in -- s_os.req_in;
         s_os.coord_out -- s_v.coord_in;
         s_v.coord_out -- cmd_out;
         sense_in -- s_v.vis_in;
         s_v.vis_out -- s_os.vis_in;
         s_os.resp_out -- resp_out;
}


component System{
    provide hardware_out;
    require hardware_in;
    inst app: Application;
         os : Operating_System;
         vis : Hardware_Sensor;
    bind app.request -- os.req_in;
         os.resp_in -- app.response;
         os.req_out -- vis.req_in;
         vis.resp_out -- os.resp_in ;
         hardware_out -- vis.cmd_out;
         hardware_in -- vis.sense_in;
}
```

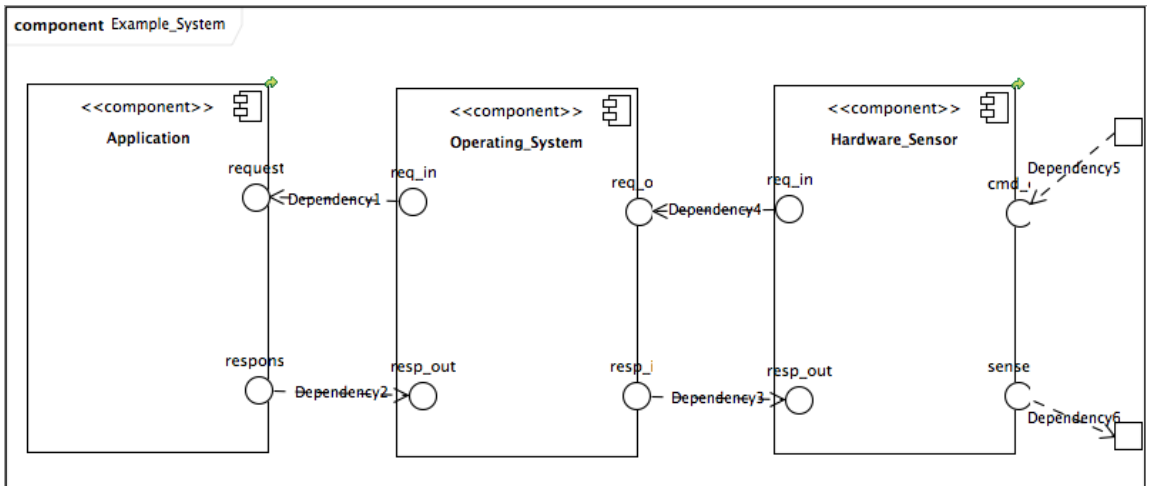
C Wright Example

```
System Example
  Component Application
    Port request = appReq!x -> request
    Port response = appResp?x -> response
    Computation request.appReq -> response.appResp -> Computation
  Component Operating_System
    Port req_in = osReqIn!x -> req_in
    Port req_out = osReqOut!x -> req_out
    Port resp_in = osRespIn!x -> resp_in
    Port resp_out = osRespOut!x -> resp_out
    Computation req_in.osReqIn  ->  req_out.osReqOut ->
                    resp_in.osRespIn -> resp_out.osRespOut ->
                    Computation
  Component Hardware_Sensor
    Port req_in = hwdReqIn!x -> req_in
    Port cmd_out = hwdCmdout!x -> cmd_out
    Port sense_in = hdwSenseIn!x -> sense_in
    Port resp_out = hwdRespout!x -> resp_out
    Computation req_in.hwdReqIn  ->  cmd_out.hwdCmdOut ->
                    sense_in.hwdSenseIn -> resp_out.hwdRespOut ->
                    Computation

  Connector A-OS_Conn
    Role AppMakeRequest = appReq?x -> AppMakeRequest
    Role OSRecRequest = osReqIn!x -> OSRecRequest
    Glue AppMakeRequest.appReq?x -> OSRecRequest.osReqIn!x -> Glue
          [] SKIP
  Connector OS-A_Conn
    Role OSReturnResp = osRespOut?x -> OSReturnResp
    Role AppRecResp = appResp!x -> AppRecResp
    Glue OSReturnResp.osRespOut?x -> AppRecResp appResp!x -> Glue
          [] SKIP
  Connector OS-HWD_Conn
    Role OSSendReq = osReqOut?x -> OSSendReq
    Role HwdRecReq = hwdReqIn!x -> HwdRecReq
    Glue OSSendReq.osReqOut?x -> HwdRecReq.hwdReqIn!x -> Glue
          [] SKIP
  Connector HWD-OS_Conn
    Role HwdSendResp = hwdRespOut?x -> HwdSendResp
    Role OSRecResp = osRespIn!x -> OSRecResp
    Glue HwdSendResp.hwdRespOut?x -> OSRecResp.osRespIn!x -> Glue
          [] SKIP

Instances
```

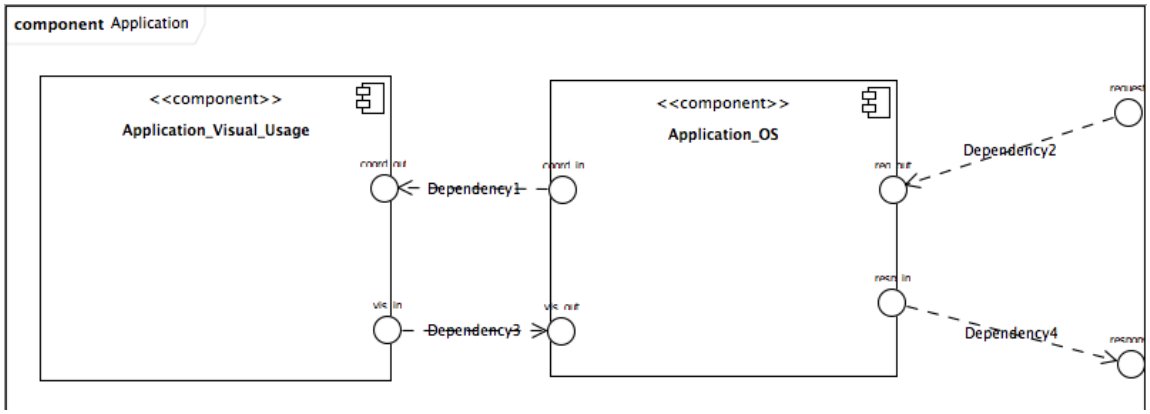
```
   app : Application
   os : Operating_System
   vis : Hardware_Sensor
   a-os : A-OS_Conn
   os-a : OS-A_Conn
   os-hwd : OS-HWD_Conn
   hwd-os : HWD-OS_Conn

Attachments
app.request as a-os.AppMakeRequest;
app.response as os-a.AppRecResp;
os.req_in as a-os.OSRecRequest;
os.req_out as os-hwd.OSSendReq;
os.resp_in as hwd-os.OSRecResp;
os.resp_out as os-a.OSReturnResp;
vis.req_in as os-hwd.HwdRecReq;
vis.cmd_out as -- System ports not in lang --
vis.sense_in as -- System ports not in lang --
vis.resp_out as hdw-os.HwdSendResp;
end Example
```

D UML2.0 Example

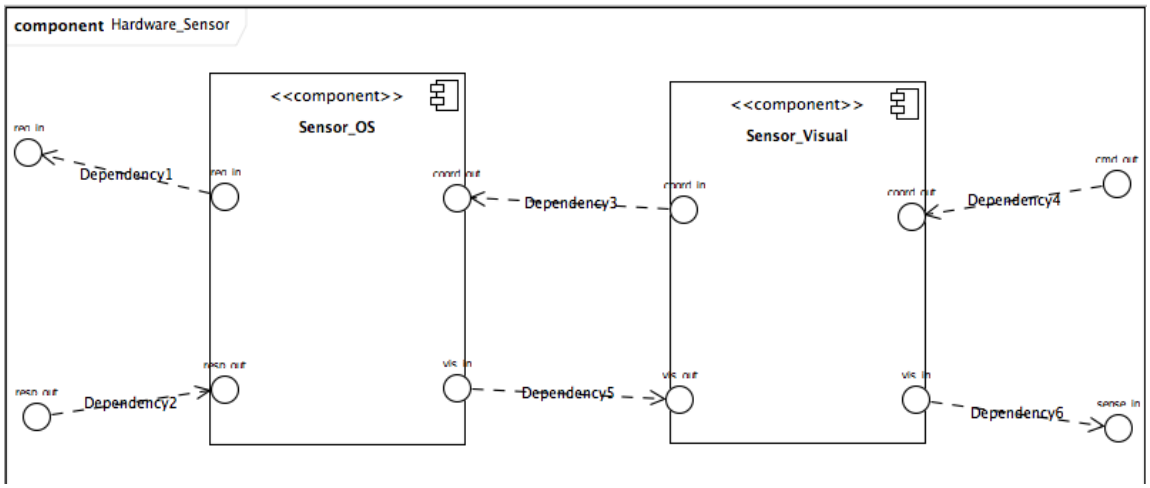
D.1 Example System Component Diagram



D.2 Application Component Diagram

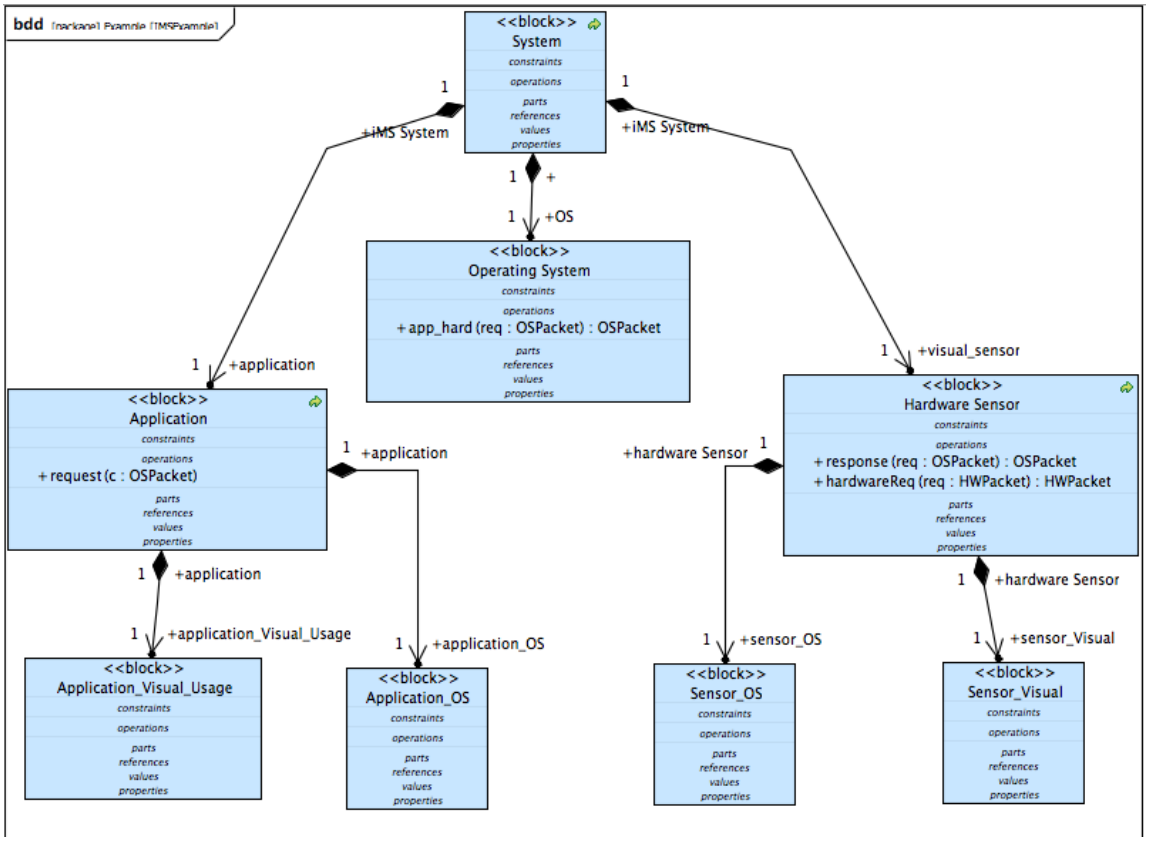


D.3 Hardware Sensor Component Diagram

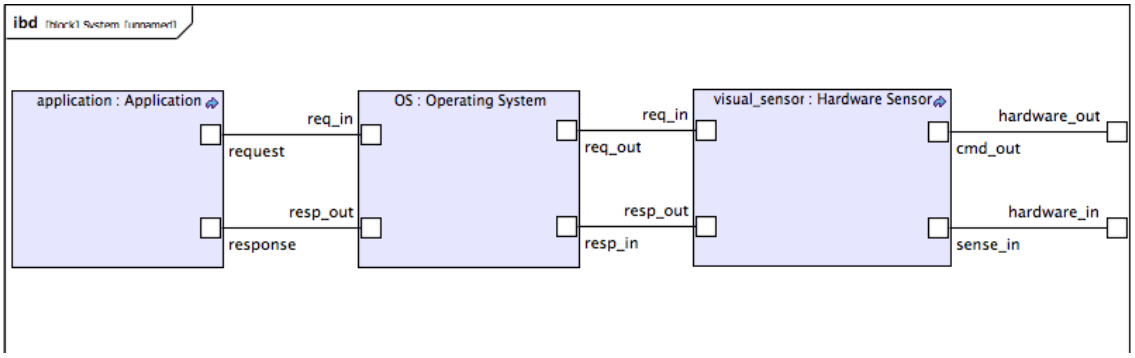


E SysML Example

E.1 Block Definition Diagram



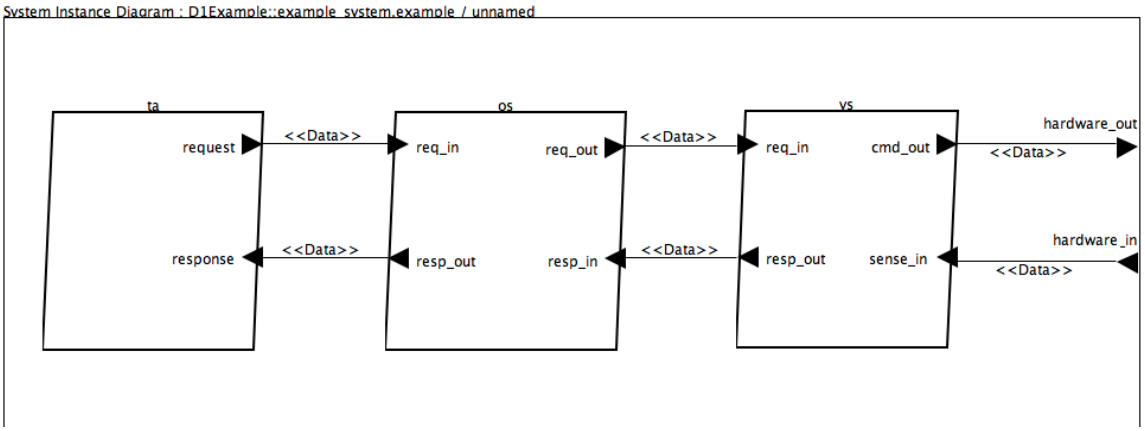
E.2 Internal Block Diagram



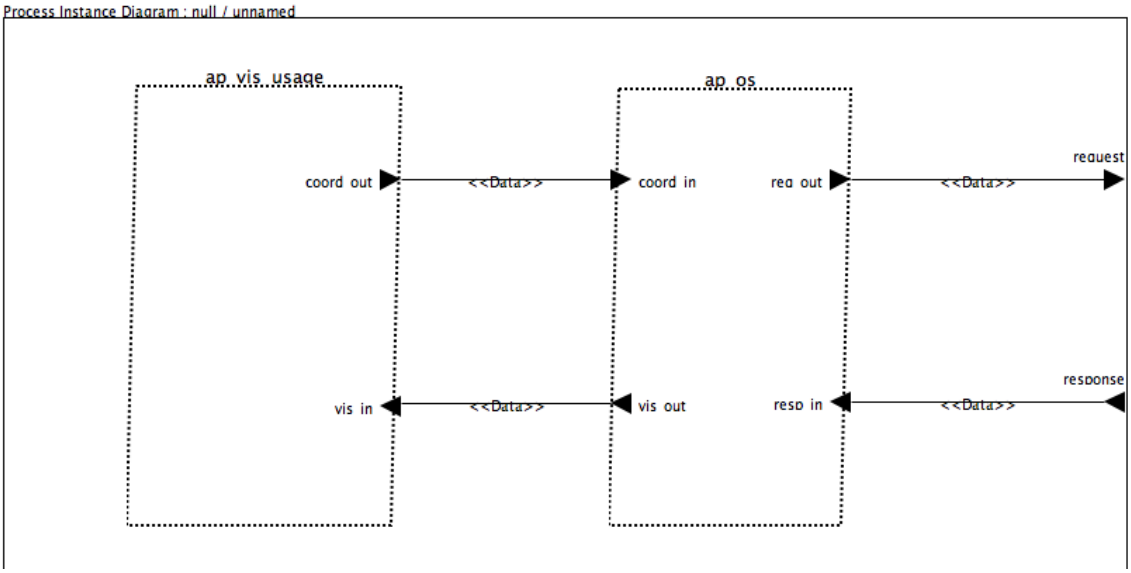
F AADL Example

F.1 AADL Graphical Representation

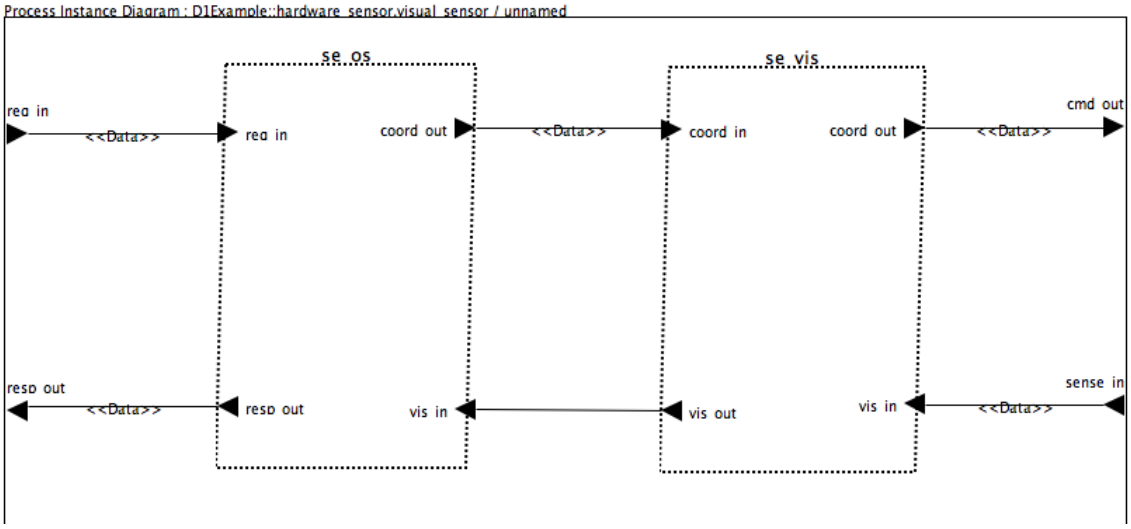
F.1.1 Example System Component



F.1.2 Application Process Component



F.1.3 Hardware Sensor Process Component



F.2 AADL Textual Representation

```
-- System type definition --
system example_system
  features
    hardware_out: out data port hard_cmd;
    hardware_in: in data port hard_rsp;
end example_system;

-- Process type definitions --
process application
  features
    response: in data port os_packet;
    request: out data port os_packet;
end application;

process operating_system
  features
    req_in: in data port os_packet;
    resp_in: in data port os_packet;
    req_out: out data port os_packet;
    resp_out: out data port os_packet;
end operating_system;

process hardware_sensor
  features
    req_in: in data port os_packet;
```

```
    sense_in: in data port hard_rsp;
    resp_out: out data port os_packet;
    cmd_out: out data port hard_cmd;
end hardware_sensor;

-- Thread type definitions --
thread application_os
  features
    coord_in: in data port coord;
    vis_out: out data port vis;
    resp_in: in data port os_packet;
    req_out: out data port os_packet;
end application_os;

thread application_visual_usage
  features
    coord_out: out data port coord;
    vis_in: in data port vis;
end application_visual_usage;

thread sensor_os
  features
    vis_in: in data port vis;
    coord_out: out data port coord;
    req_in: in data port os_packet;
    resp_out: out data port os_packet;
end sensor_os;

thread sensor_visual_creation
  features
    coord_out: out data port hard_cmd;
    vis_in: in data port hard_rsp;
    coord_in: in data port coord;
    vis_out: out data port vis;
end sensor_visual_creation;

-- System implementation definition --
system implementation example_system.example
  subcomponents
    ta: process application.targeting_application;
    os: process operating_system.os;
    as: process hardware_sensor.sensor_sensor;
  connections
    data port ta.request -> os.req_in;
    data port os.resp_out -> ta.response;
    data port os.req_out -> as.req_in;
```

```
    data port as.resp_out -> os.resp_in;
    data port as.cmd_out -> hardware_out;
    data port hardware_in -> as.sense_in;
  properties
    contract_props::resp_time => 30;
end example_system.example;

-- Process implementation definitions --
process implementation application.targeting_application
  subcomponents
    ap_os: thread application_os;
    ap_vis_usage: thread application_visual_usage;
  connections
    data port ap_os.req_out -> request;
    data port response -> ap_os.resp_in;
    data port ap_vis_usage.coord_out -> ap_os.coord_in;
    data port ap_os.vis_out -> ap_vis_usage.vis_in;
end application.targeting_application;

process implementation operating_system.os
end operating_system.os;

process implementation hardware_sensor.sensor_sensor
  subcomponents
    se_os: thread sensor_os;
    se_vis: thread sensor_visual_creation;
  connections
    data port req_in -> se_os.req_in;
    data port se_os.coord_out -> se_vis.coord_in;
    data port se_vis.coord_out -> cmd_out;
    data port sense_in -> se_vis.vis_in;
    data port se_vis.vis_out -> se_os.vis_in;
    data port se_os.resp_out -> resp_out;
end hardware_sensor.sensor_sensor;

-- Property set defintion example --
property set contract_props is
  resp_time: aadlinteger  applies to (system);
end contract_props;

-- Data type definitions --
data os_packet
end os_packet;

data hard_rsp
end hard_rsp;
```

```
data hard_cmd
end hard_cmd;

data coord
end coord;

data vis
end vis;
```