An Information Model for High-Integrity Real Time Systems

Alek Radjenovic, Richard Paige, Philippa Conmy, Malcolm Wallace, and John McDermid

High-Integrity Systems Group, Department of Computer Science,
University of York
Heslington, York, YO10 5DD, United Kingdom
Telephone: +44 (1904) 433384

{alek,paige,philippa,malcolm,jam}@cs.york.ac.uk

ABSTRACT

We propose an information model that captures information suitable for architectural modelling in high integrity real time systems. We also suggest an overall context for architectural modelling and place the information model within this context. Furthermore, we use a simple three-phased approach to architectural modelling. With such approach, models are both tractable but also rich enough to be able to provide for various kinds of formal analyses. This work has been carried out by the DARP HIRTS (Defence and Aerospace Research Partnership for High Integrity Real Time Systems), part of the HISE (High Integrity Systems Engineering) research group at the University of York, and in collaboration with BAE SYSTEMS, Rolls-Royce plc and QinetiQ ltd.

Keywords: *information model, architectural modelling, high integrity real time systems*

INTRODUCTION

The purpose of this document is to propose an information model for architecture modelling in high-integrity real time systems (HIRTS). An **architectural model** (also known as a *design reference model*) is a high-level design for systems in a domain. By systems, in this context, we imply applications comprising software and hardware. The architectural model focuses on identifying domain-oriented subsystems and concurrent processes, and on allocating the features, functions, and data objects defined in the domain models to the processes, hardware items and subsystems. An architectural model is developed in the early phase of system design and the detailed decomposition and component construction can be done from this model.

Information analysis captures and defines the domain knowledge and data requirements that are essential for implementing systems in the domain. The domain knowledge is either contextual information, which gets lost after the development, or is deeply embedded in the software and is often difficult to trace. The purpose of the information analysis is to represent the domain knowledge explicitly in terms of domain entities and their relationships, and to make them available for the derivation of subsystem and component definitions during the process of architecture modelling.

The product of an information analysis is the **information model**. It is used by the requirements analysts and system architects to ensure that proper abstractions and decompositions are used in the development of the system. In addition, those who maintain, reuse and test the system (or parts of it) need this information in order to understand the problems the design addresses. Finally, the information model also defines data that is assumed to come from external sources or, in more general terms, the system environment.

The information model constructed in the course of work on the DARP HIRTS project [4] at the University of York was produced because we and our project partners (BAE SYSTEMS, Rolls Royce plc, and QinetiQ

ltd) had determined that no individual language provided complete coverage of necessary concepts. This conclusion, and the information model, were produced by reviewing a large number of industrial case studies of HIRTS supplied by the project partners and by review of standard architectural modelling languages for HIRTS, particularly AADL [1] and MetaH [3].

CONTEXT

The diagram in Figure 1 represents our perspective on the fundamental features in architectural modelling and places the information model in this context.

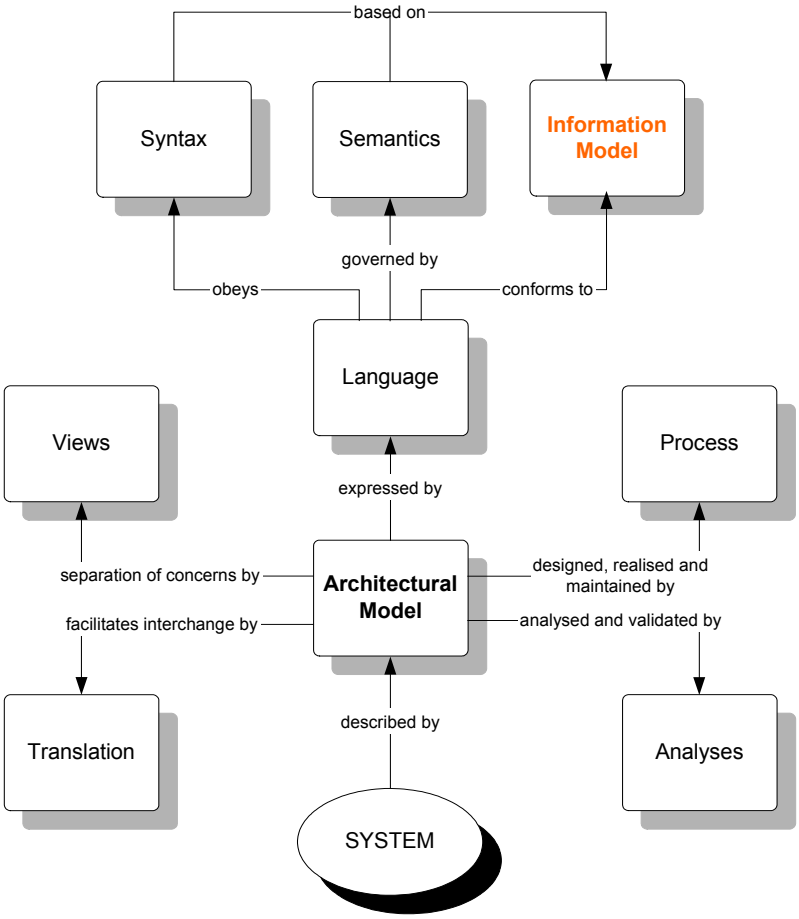

Figure 1. Architectural Modelling Context

The *Information Model, Syntax* and *Semantics* form the *Language* required to represent *any* HIRTS model. A textual notation, such as AADL, is essential, but a graphical equivalent, such as UML 2.0 [2] or SysML [5], may also be regarded as necessary. The model design, and the system realisation (implementation) and its maintenance are controlled by the *Process*. The *Architectural Model* is analysed and validated by various forms of *Analyses*. The analyses performed will depend on the model's level of abstraction. These include constraint, assertion and contract validations, as well as change management. The *Views* represent the separation of concerns. Finally, it is vital to provide a mechanism for information interchange - a two way model mapping into existing technologies, languages and notations, such as UML or an ADL (Architectural

Description Language). This is achieved by means of *Translation*, facilitating the use of external tools and processes for: analyses, graphical representations, formal proofs, code generation and testing.

PRINCIPLES

It is important to emphasise the key philosophy behind the solution presented here, and that is:

"Documenting the architecture in an abstract but precise way."

Abstraction is inherent in any modelling. Nevertheless, in order to prepare a solid foundation for mathematical analyses and formal proofs, it is crucial that the model is also expressed in a very precise way.

To keep matters simple we have sought to capture a *minimal* set of necessary information that will allow easy translation into mathematical models. In order to achieve this, we propose a three- phased approach, as shown in Figure 2.

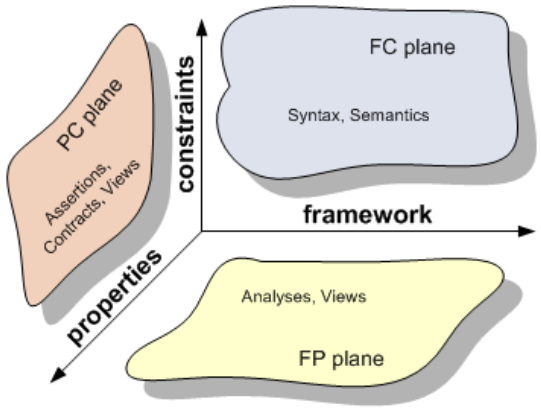

Figure 2. A three-phased approach to information modelling

The sections that follow describe in more detail the rationale, the meaning and the content of each phase in the diagram: *framework*, *properties* and *rules*.

FRAMEWORK

At the heart of the information model is the *Component* element. Component-based approaches enforce modularisation, hierarchical design, reuse, good composition and a well-defined interface to the environment. Each component can declare and/or define its *category*, *interface* and *implementation*.

In this first phase, we describe those information elements that reflect physical constructs of real time embedded systems, such as software and hardware components, and their combinations. Thus we recognise *software*, *hardware* and *composite* component categories.

We define a minimal set of three kinds of software components: *process*, *thread* and *data*.

A **process** represents a protected virtual address space (partition unit) that contains an executable binary image. A **thread** is a sequential flow of control through a binary image, i.e. a schedulable unit executing on a processor under a scheduling protocol. A **data** component corresponds to a data type in source text that defines a representation and interpretation for: instances static or dynamic data or parameter values transferred between subprograms.

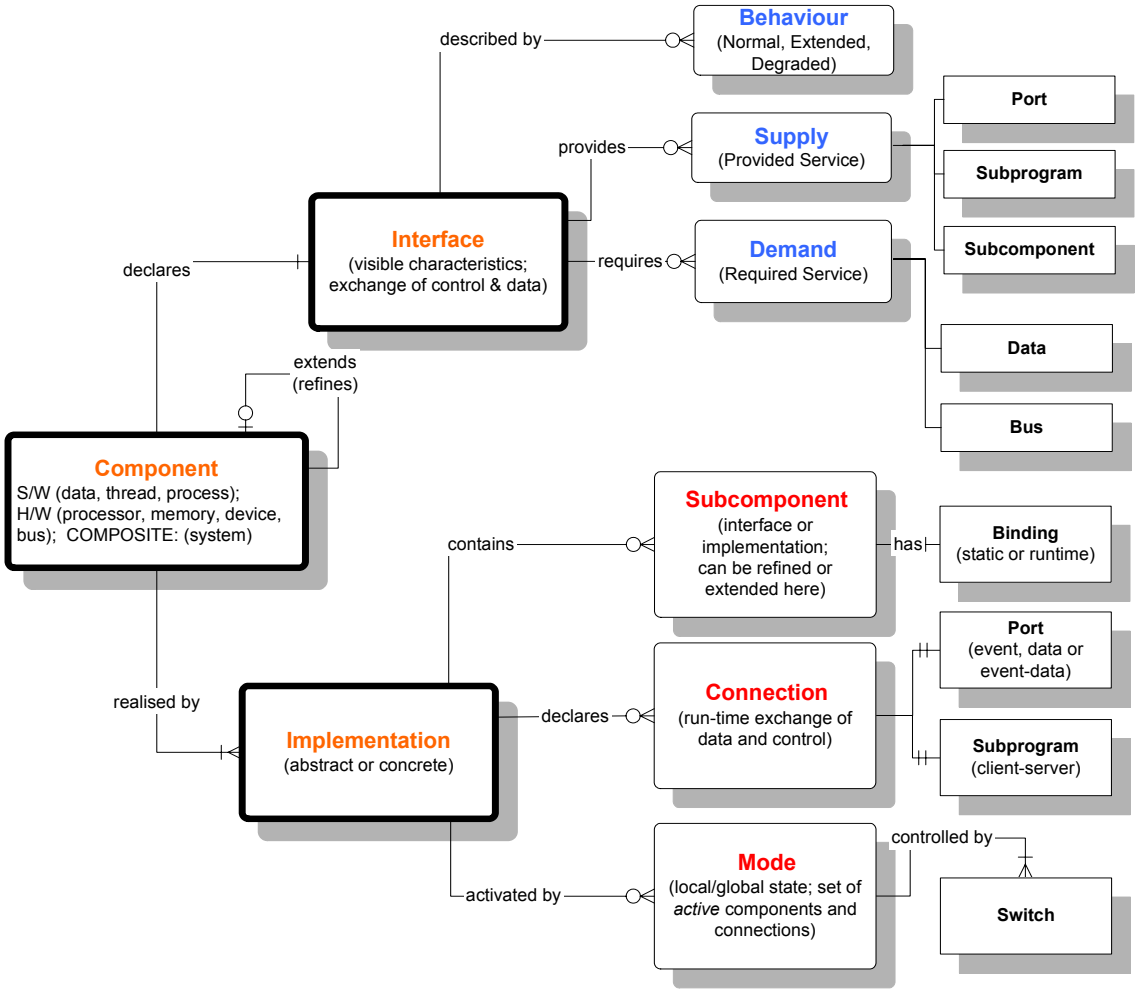

Figure 3. Framework Model

We recognise four kinds of hardware components: *processor*, *memory*, *bus* and *device*.

A **processor** represents a hardware unit capable of executing binary images that schedules and executes threads, providing a periodic clock interrupt to drive periodic dispatching. A **memory** component represents randomly accessible physical storage (e.g. RAM, ROM) or a complex permanent storage (e.g. disks, logical storage). A **bus** is a hardware component with communication protocols for exchange of control and data. It represents any communication channel between processors, memory, and devices. A **device** symbolises a physical component interfacing with the environment.

The only composite defined is **system**. It specifies the hierarchical composition of software and hardware components.

Component interface represents externally visible characteristics of a component. It identifies the interaction points with other components. These can be expressed in terms of *provided* (offered) or *required* (supplier) services. The interface also defines the component behaviour in the presence (or absence) of some, or all, of the required services. **Component implementation** is an embodiment of a component interface. It represents contained components (subcomponents) and their interconnection, properties, and modes of a component. Every implementation must be associated with a single component interface, and there may be zero or more implementations declared for each component interface. Figure 3 summarises the framework information elements.

PROPERTIES

In the second phase, we attach properties to the framework information elements in order to describe non-functional characteristics of the system.

A **property** constrains or adds information to the declaration of a component or its constituent parts. A property has a name and a property type. A **property set** contains declarations of property types and property names that are allowed to appear in the specification (the architectural model). A *standard* property set pre-declares a set of *standard* properties for all specifications. Additional property sets can be declared and they represent an extension mechanism for the information model.

Properties can have associated **expressions** that are statically typed, and evaluate to a specific value. The time at which a property expression is evaluated may depend on the property and on how a specification is processed. For example, some expressions may be evaluated immediately, some after binding decisions have been made, and some reflect run-time state information, e.g., the current mode. During analysis all property expressions can be evaluated to a known value, if necessary by considering all possible runtime states. Property expressions may also be used as part of a simulation or runtime execution, e.g., to make run-time binding or mode switching decisions. A given property name may have a default expression.

A **property association** associates a property value, or list of property values, with a property name resulting from evaluation of property expressions.

Properties defined in the standard set include: *temporal properties* (various deadlines, execution times, propagation delays, load times); *integer properties* (queue sizes, word counts, priorities, message size, code, data, stack and heap sizes); *protocols* (connection, dispatch, concurrency control, overflow handling, scheduling, memory); *bindings* (actual/allowed/available processor/memory/connection); *access* (provided, required); and others such as source language and source mapping.

CONSTRAINTS

The constraints can be applied to framework elements or properties. We distinguish between two types of constraints: *language rules* and *system constraints*.

The **language rules** deal with the syntax, semantics and legality rules that need to be adhered to during the formulation of a system specification (architectural model). They apply to the declaration of each of the framework elements and their composition, the declaration of new, extended or refined property sets, property associations, system constraint definitions, assertions and contracts.

System constraints capture the behaviour essential to meet the requirements, the emergent properties from system analyses, as well as the syntactic information for integration purposes (such as units, accuracy etc). A set of one or more constraints may form a *contract* between components when two or more components collaborate. A **contract** is formed of pre-conditions (which a client – a user of the service – must meet) and post-conditions (which the service supplier guarantees). In addition a constraint may describe a dependency between two or more different components (for example, if they should be using the same scientific units

for a variable, or if they should be physically independent). Figure 4 describes in more detail the concept of constraint modelling.

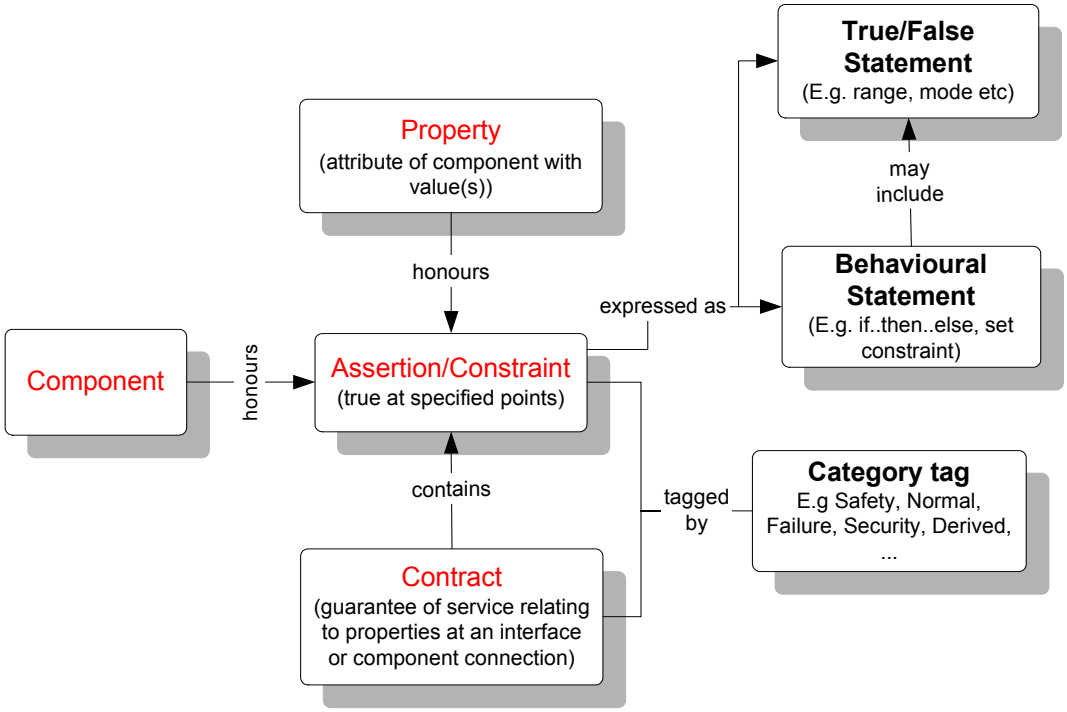

Figure 4. Constraint Modelling.

SUMMARY

The information model presented here provides minimal but sufficient information necessary to describe any high-integrity real time embedded system. This approach allows for various degrees of abstraction in describing the system architecture. As a consequence, the models are both tractable but also rich enough to be able to provide for various kinds of analyses, such as schedulability, performance, reliability, safety, fault tolerance or dynamic configurability analysis. Some early work carried out on small case studies already shows promising results using different languages (such as AADL, MetaH, etc). Furthermore, XML can be used to describe the information model, and by doing so it would be possible to leverage this approach by employing (through translations) existing technologies, such as UML and MetaH, and thus gaining access to graphical representation and formal proofs.

REFERENCES

1. SAE, *Avionics Architecture Description Language (AADL)*. 2003, SAE, AS-2C Architecture Description Language Subcommittee.
2. OMG, *UML 2.0 Infrastructure*, 2004, www.omg.org.
3. Honeywell, *MetaH Language and Tools*, available at http://www.htc.honeywell.com/projects/dssa/dssa_tools/dssa_tools_mh.html, last accessed 1 April 2004.
4. DARP HIRTS public web site: http://www.cs.york.ac.uk/hise/darp/index.php?link=main.php, last accessed 15 April 2004.
5. SysML Partners, *Systems Modelling Language (SysML) version 0.3*, January 2004, www.sysml.org