INtegrated TOol chain for model-based design of CPSs

INTO-CPS Method Guidelines

Version: 1.5

Date: March 2018

The INTO-CPS Association

http://into-cps.org

Contributors:

John Fitzgerald, UNEW
Carl Gamble, UNEW
Martin Mansfield, UNEW
Richard Payne, UNEW
Ken Pierce, UNEW

Editors:

Ken Pierce, UNEW

Document History

Ver	Date	Author	Description
0.1	14-02-2017	Ken Pierce	Initial document structure
0.2	24-10-2017	Ken Pierce	Revised document structure
0.3	26-10-2017	Ken Pierce	First draft of materials
0.4	31-10-2017	Ken Pierce	Added SysML chapter
0.5	01-11-2017	Ken Pierce	Draft for comment
1.0	07-12-2017	Ken Pierce	First revisions based on comments
1.1	12-12-2017	Ken Pierce	Further updates based on comments
1.2	14-12-2017	Ken Pierce	Revised SysML chapter
1.3	18-12-2017	Carl Gamble	Traceability chapter completed
1.4	18-12-2017	Carl Gamble	DSE chapter completed
1.5	13-03-2018	Peter Gorm Larsen	Moved over to the INTO-CPS Association

Abstract

This document is the final methods guidance document for the INTO-CPS technologies. It is aimed at end users of the technologies, and complements the User Manual, Deliverable D4.3a [BLL$^+$17], by helping to describe the why to complement the how. This document presents: a concepts base, which describes the terminology used within INTO-CPS; information on getting started with the technologies, and the variety of workflows they support; a description of the traceability features of the tool chain, and why these must be considered at the beginning of development to realise them fully; guidelines on incorporating requirements engineering in a cyber-physical systems (CPS) context; a description of the INTO-SysML profile and its use; guidance on discrete-event first (DE-first) modelling as a way to begin multi-modelling; guidance on modelling networks in multi-models; and guidelines for the use of design space exploration (DSE) features of the INTO-CPS tool chain.

Contents

Part I

Introductory Material

Introduction

The INTO-CPS tool chain brings together a variety of technologies to allow engineers to undertake collaborative, model-based based design of Cyber-Physical Systems (CPSs). Each technology has its own culture, abstractions, and approaches to problem solving that inform how they are used. Many of these things are tacit and tend to be discovered only after trying to combine them. The guidance in this document aims to help the reader overcome these challenges, and to understand how best to use these technologies.

This document complements the tools User Manual (Deliverable D4.3a [BLL+17]) —which gives detail on how to use the features of the tool chain— by providing information on when and why you might use these features. The guidance in this document has been distilled from experience gained in a series of pilot studies and applications of INTO-CPS technologies to real industrial case studies. These pilot studies now appear as examples that can be opened directly from the INTO-CPS Application, supported by descriptions in the Examples Compendium (Deliverable D3.6 [MGP+17]). Industrial applications can be read about in the Case Studies report (Deliverable D1.3a [OLF+17]).

1 How to Use This Document

Since this document is aimed at both new and experienced users of the INTO-CPS technologies, it has been divided into two parts. Part I, Chapters I–8, covers introductory material including this introduction, the terminology used in INTO-CPS, and the various activities that INTO-CPS enables. Part II, Chapters II–29, covers more advanced topics that require a basic familiarity with the INTO-CPS technologies.

While the chapters in the Part II are ordered primarily based on a start-to-end "work flow" of system development with INTO-CPS, it is not necessary to read the advanced chapters in order. While experienced users may read any chapter on which they require further guidance, new users are recommended to:

- Read the introductory material in Part I.
- Follow the first tutorial to experience using the INTO-CPS Application.
- Import one or two examples from the Examples Compendium (Deliverable D3.6 [MGP+17]) into the INTO-CPS Application and interact with them.
- As you start your own multi-modelling, return to this document as and when you require guidance on a particular area.

2 Overview of Sections

Chapter 2: Concepts and Terminology This chapter is an introduction to the concepts and terminology used in INTO-CPS. It explains many terms from the various baseline technologies, as well as other model-based design terminology. In parts this involved reconciling terms used differently in different areas, and finding common, agreed-upon terms for similar concepts. These concepts are applicable for all documents produced by INTO-CPS (this document, user manuals, deliverables, and publications).

Chapter 8: Getting Started with INTO-CPS This chapter suggests how to get started with the INTO-CPS tool chain, trying out core features by following tutorials, which puts the other range of activities in context. It also describes the full range of activities that the

INTO-CPS tool chain enables.

Chapter II: Traceability and Provenance This chapter explains how to approach the INTO-CPS tool chain in order to make used of the machine-assisted traceability features included in the INTO-CPS Application and baseline tools. It also describes the set of included queries that can be run over traceability data sets, and how further queries can be written.

Chapter 14: Requirements Engineering This chapter focuses on a key initial activity for CPS design, specifically requirements engineering (RE) in a CPS context, and the specification and documentation of requirements placed upon a CPS. This section describes an approach called SoS-ACRE in the context of INTO-CPS, and includes descriptions of how this approach can be realised using tools identified as useful by the industrial partners (specifically SysML and Excel). By following these guidelines, engineers can bridge the gap between natural language requirements and multi-models.

Chapter 18: SysML and Multi-modelling This chapter describes the various roles of SysML in INTO-CPS. SysML can be used for architectural modelling of CPSs, while INTO-CPS provides additional SysML profiles that can be used to describe the architecture of multi-models and provide machine-assisted configuration of co-simulations and other analyses. This section provides a description of these profiles, how standard SysML can be used within INTO-CPS, and the relationship between these two uses.

Chapter 22: Initial Multi-modelling This chapter looks at producing an initial multi-model through the creation of abstract, discrete-event FMUs. These simplified FMUs can then be replaced by higher-fidelity versions in more appropriate tools such as 20-sim. This is referred to as a "DE-first" approach [FLPV13].

Chapter 25: Modelling Networks in Multi-models This chapter describes how to also model realistic communications between controllers in an FMI setting. This chapter describes one approach: introducing an FMU that represents an abstract communication mechanism, the *ether*. Guidance on the consequences of adopting such an approach is included, as well as extensions to cover quality-of-service modelling.

Chapter 29: Design Space Exploration This chapter gives guidance on DSE, including the types of search algorithms that can be used to explore a design space, and how the SysML profile extensions help in the design of experiments.

Differences from Previous Versions

This document builds on previous versions Deliverables D3.1a [FGPP15a] and D3.2a [FGPP16a]). Some material is retained and updated, while other material is entirely new. The following list gives an overview of new and updated material for each section:

Concepts and Terminology appeared in the previous version. The concepts base has been stable in the final year of the project.

Getting Started with INTO-CPS has been heavily revised from previous "workflows" section in response to end user interactions and feedback.

Traceability and Provenance is entirely new.

Requirements Engineering appeared in the previous version.

SysML and Multi-modelling has been updated significantly to present a comprehensive overview of SysML in the INTO-CPS context, using new and revised material.

Initial Multi-modelling appeared in the previous version.

Modelling Networks in Multi-models appeared previously.

Design Space Exploration has been revised to include description of how to select the algorithm to use and an outline of an iterative search approach.

Concepts and Terminology

This section introduces the basic concepts used in the INTO-CPS project. CPSs bring together domain experts from diverse backgrounds, from software engineering to control engineering. Each discipline has developed their own terminologies, principles and philosophy for years — in places they use similar terms for quite different meanings and different terms that have the same meaning. In addition, the INTO-CPS project aims to produce a tool chain for CPS engineering resulting in the need for common tool-based terminology. INTO-CPS requires experts from diverse fields to work collaboratively, so this section gives some core concepts of INTO-CPS that will be used throughout the project. We divide the concepts into several broad areas in the remainder of this section.

3   Systems

A *System* is defined as being "a combination of interacting elements organized to achieve one or more stated purposes" [INC15]. Any given system will have an *environment*, considered to be everything outside of the system. The behaviour exhibited by the environment is beyond the direct control of the developer [BFG$^+$12]. We also define a *system boundary* as being the common frontier between the system and its environment. The definition of the system boundary is application-specific [BFG$^+$12].

Cyber-Physical Systems (CPSs) refer to "ICT systems (sensing, actuating, computing, communication, etc.) embedded in physical objects, interconnected (including through the Internet) and providing citizens and businesses with a wide range of innovative applications and services" [Tho13, DAB$^+$15].

A *System of Systems (SoS)* is a "collection of constituent systems that pool their resources and capabilities together to create a new, more complex system which offers more functionality and performance than simply the sum of the constituent systems" [HIL$^+$14]. CPSs may exhibit the characteristics of SoSs.

4   Models

In the INTO-CPS project, we concentrate on "model-based design" of CPSs. A *model* is a potentially partial and abstract description of a system, limited to those components and properties of the system that are pertinent to the current goal [HIL$^+$14]. A model should be "just complex enough to describe or study the phenomena that are relevant for our problem context" [vA10]. Models should be abstract "in the sense that aspects of the product not relevant to the analysis in hand are not included" [FL98]. A model "may contain representations of the system, environment and stimuli" [FLV14][1].

In a CPS model, we model systems with cyber, physical and network elements. These components are often drawn from different domains, and are modelled in a variety of languages, with different notations, concepts, levels of abstraction, and semantics, which are not necessarily easily mapped one to another. This heterogeneity presents a significant challenge for simulation in CPSs [HIL$^+$14]. In INTO-CPS we use *continuous time (CT)* and *discrete event (DE)*

---

[1]Further discussion is required in the final year of INTO-CPS regarding the definition of aspects of models in particular; environment models, test models in RT-Tester and their correspondence in the INTO-CPS SysML profile.

models to represent physical and cyber elements as appropriate. A CT model has state that can be changed and observed *continuously* [vA10] and is described using either explicit continuous functions of time either implicitly as a solution of differential equations. A DE model has state that can be changed and observed only at fixed, *discrete*, time intervals [vA10]. The approach used in the DESTECS project was to use *co-models* – "a model comprising a DE model, a CT model and a contract" [BFG$^+$12]. In INTO-CPS we propose the use of **multi-models** – "comprising multiple **constituent** DE and CT models". Related to this is a **Hybrid Model**, which contains both DE and CT elements.

A *requirement* may impose restrictions, define system capabilities or identify qualities of a system and should indicate some value or use for the different stockholders of a CPS. **Requirements Engineering (RE)** is the process of the specification and documentation of requirements placed upon a CPS. Requirements may be considered in relation to different **contexts** – that is the point of view of some system component or domain, or interested stakeholder.

We cover the main features of the notations used in INTO-CPS in Section 7. Here we consider some general terms used in models. A **design parameter** is a property of a model that can be used to affect the model's behaviour, but remains constant during a given simulation [BFG$^+$12]. A **variable** is feature of a model that may change during a given simulation [BFG$^+$12]. **Non-functional properties (NFPs)** pertain to characteristics other than functional correctness. For example, reliability, availability, safety and performance of specific functions or services are NFPs that are quantifiable. Other NFPs may be more difficult to measure [PF10].

The activity of creating models may be referred to as **modelling** [FLV14] and related terms include **co-modelling** and **multi-modelling**. A **workflow** is a sequence of **activities** performed to aid in modelling. A workflow has a defined purpose, and may cover a subset of the CPS engineering development lifecycle.

The term **architecture** has many different definitions, and range in scope depending upon the scale of the product being 'architected'. In the INTO-CPS project, we use the simple definition from [PHP$^+$14]: "an architecture defines the major elements of a system, identifies the relationships and interactions between the elements and takes into account process. Those elements are referred to as **components**. An architecture involves both a definition of structure and behaviour. Importantly, architectures are not static but must evolve over time to reflect the change in a system as it evolves to meet changes to its requirements". In a CPS architecture, components may be either **cyber components** or **physical components** corresponding to some functional logic or an entity of the physical world respectively.

In INTO-CPS we consider both a **holistic architecture** and a **design architecture**. An example of their use is given in Chapter 18. The aim of a holistic architecture is to identify the main units of functionality of the system reflecting the *terminology and structure of the domain of application*. It describes a conceptual model that highlights the main units of the system architecture and the way these units are connected with each other, taking a holistic view of the overall system. The design architectural model of the system is effectively a multi-model. The INTO-CPS SysML profile [APCB15] is designed to enable the specification of CPS design architectures, which emphasises a decomposition of a system into **subsystems**, where each subsystem is an assembly of cyber and physical components and possibly other subsystems, and modelled separately in isolation using a special notation and tool designed for the domain of the subsystem. **Evolution** refers to the ability of a system to benefit from a varying number of alternative system components and relations, as well as its ability to gain from the adjustments

of the individual components' capabilities over time (Adjusted from SoS [NLF+13]).

Considering the interactions between components in a system architecture, an **interface** "defines the boundary across which two entities meet and communicate with each other" [HIL+14]. Interfaces may describe both digital and physical interactions: digital interfaces contain descriptions of operations and attributes that are *provided* and *required* by components. Physical interfaces describe the flow of physical matter (for example fluid and electrical power) between components.

There are many methods of describing an architecture. In the INTO-CPS project, an **architecture diagram** refers to the symbolic representation of architectural information contained in a model. An **architectural framework** is a "defined set of viewpoints and an ontology" and "is used to structure an architecture from the point of view of a specific industry, stakeholder role set, or organisation. [HIL+14]. In the application of an architecture framework, an **architectural view** is a "work product (for example an architecture diagram) expressing the architecture of a system from the perspective of specific system concerns" [PHP+14].

The INTO-CPS SysML profile comprises diagrams for architectural modelling and **design space exploration** specification. There are two architectural diagrams. The **Architecture Structure Diagram (ASD)** specialises SysML block definition diagrams to support the specification of a system architecture described in terms of a system's components. **Connections Diagrams (CDs)** specialise SysML internal block diagrams to convey the internal configuration of the system's components and the way they are connected. The system architecture defined in the profile should inform a co-simulation multi-model and therefore all components interact through connections between flow ports. The profile permits the specification of **cyber** and **physical** components and also components representing the **environment** and **visualisation** elements. The INTO-CPS SysML profile includes three design space exploration diagrams: a **parameters diagram**; an **objective diagram**; and a **ranking diagram**. See Section 6 for concepts relating to design space exploration.

5 Tools

The **INTO-CPS tool chain** is a collection of software tools, based centrally around FMI-compatible co-simulation, that supports the collaborative development of CPSs. The **INTO-CPS Application** is a front-end to the INTO-CPS tool chain. The application allows the specification of the co-simulation configuration, and the co-simulation execution itself. The application also provides access to features of the tool chain without an existing user interface (such as design space exploration and model checking). Central to the INTO-CPS tool chain is the use of the Functional Mockup Interface (FMI) standard.

The **Functional Mockup Interface (FMI)** is a tool-independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [Blo14]. Part of the FMI standard for model exchange is specification of a **model description** file. This is an XML file that supplies a description of all properties of a model (for example input/output variables). A **Functional Mockup Unit (FMU)** is a tool component that implements FMI. Data exchange between FMUs and the synchronisation of all simulation solvers [Blo14] is controlled by a **Master Algorithm**.

Co-simulation is the simultaneous, collaborative, execution of models and allowing information to be shared between them. The models may be CT-only, DE-only or a combination of

both. The ***Co-simulation Orchestration Engine (COE)*** combines existing co-simulation solutions (FMUs) and scales them to the CPS level, allowing CPS multi-models to be evaluated through co-simulation. This means that the COE implements a ***Master Algorithm***. The COE will also allow real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

In the INTO-CPS Application, a ***project*** comprises: a number of FMUs, optional source models (from which FMUs are exported); a collection of ***multi-models***; and an optional SysML architectural model. A multi-model includes a list of FMUs, defined instances of those FMUs, specified connections between the inputs/outputs of the FMU instances, and defined values for design parameters of the FMU instances. For each multi-model a ***co-simulation configuration*** defines the step size configuration, start and end time for the co-simulation of that multi-model. Several configurations can be defined for each multi-model.

Code generation is the transformation of a model into generated code suitable for compilation into one or more target languages (e.g. C or Java).

The INTO-CPS project considers two tool-supported methods for recording the rationale of design decisions in CPSs. ***Traceability*** is the association of one model element (e.g. requirements, design artefacts, activities, software code or hardware) to another. ***Requirements traceability*** "refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [GF94]. ***Provenance*** "is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [MG13]. In INTO-CPS traceability between model elements defined in the various modelling tools is achieved through the use of ***OSLC messages***, handled by a traceability ***daemon tool***. This supports the ***impact analysis*** and general ***traceability queries***.

Two broad groups of users are considered in the INTO-CPS project. A ***Tool Chain User*** is an individual who uses the INTO-CPS Tool Chain and its various analysis features. A ***Foundations Developer*** is someone who uses the developed foundations and associated tool support (see Section 8) to reason about the development of tools.

6 Analysis

Design-Space Exploration (DSE) is "an activity undertaken by one or more engineers in which they build and evaluate [multi]-models in order to reach a design from a set of requirements" [BFG$^+$12]. "The ***design space*** is the set of possible solutions for a given design problem" [BFG$^+$12]. Where two or more models represent different possible solutions to the same problem, these are considered to be ***design alternatives***. In INTO-CPS design alternatives are defined using either a range of parameter values or different multi-models. Each choice involves making a selection from alternatives on the basis of an ***objective*** – criteria or constraints that are important to the developer, such as cost or performance. The alternative selected at each point constrains the range of design alternatives that may be viable next steps forward from the current position. Given a collection of alternatives with corresponding objective results, a ***ranking*** may be applied to determine the 'best' design alternative.

Test Automation (TA) is defined as the machine assisted automation of system tests. In INTO-CPS, we concentrate on various forms of ***model-based testing*** – centering on testing system

models, against the requirements on the system. The ***System Under Test (SUT)*** is "the system currently being tested for correct behaviour. An alias for system of interest, from the point of view of the tester" [HIL$^+$14]. The SUT is tested against a collection of ***test cases*** – a finite structure of input and expected output [UPL06], alongside a ***test model***, which specifies the expected behaviour of a system under test [CMC$^+$13]. TA uses a ***test suite*** – a collection of ***test procedures***. These test procedures are detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases [WG-92].

INTO-CPS considers three main types of test automation: ***Hardware-in-the-Loop (HiL)***, ***Software-in-the-Loop (SiL)*** and ***Model-in-the-Loop (MiL)***. In ***HiL*** there is (target) hardware involved, thus the FMU is mainly a wrapper that interacts (timed) with this hardware; it is perceivable that realisation heavily depends on hardware interfaces and timing properties. In ***Software-in-the-Loop (SiL)*** testing the object of the test execution is an FMU that contains a software implementation of (parts of) the system. It can be compiled and run on the same machine that the COE runs on and has no (defined) interaction other than the FMU-interface. Finally, in ***Model-in-the-Loop (MiL)*** the test object of the test execution is a (design) model, represented by one or more FMUs. This is similar to the SiL (if e.g., the SUT is generated from the design model), but MiL can also imply that running the SUT-FMU has a representation on model level; e.g., a playback functionality in the modelling tool could some day be used to visualise a test run.

Model Checking (MC) exhaustively checks whether the model of the system meets its specification [CGP99], which is typically expressed in some temporal logic such as ***Linear Time Logic (LTL)*** [Pnu77] or ***Computation Tree Logic (CTL)*** [CE81]. As opposed to testing, model checking examines the entire state space of the system and is thus able to provide a correctness proof for the model with respect to its specification. In INTO-CPS, we can concentrate on ***Bounded Model Checking (BMC)*** [CBRZ01, CKOS04, CKOS05], which is based on encodings of the system in propositional logic, for a timed variant of LTL. The key idea of this approach is to represent the semantics of the model as a Boolean formula and then apply a ***Satisfiability Modulo Theory (SMT)*** [KS08] solver in order to check whether the model satisfies its specification. A powerful feature of model checking is that, if the specification is violated, it provides a counterexample trace that shows exactly how an undesired state of the system can be reached [CV03].

7   Existing Tools and Languages

The INTO-CPS tool chain uses several existing modelling tools. ***Overture***[2] supports modelling and analysis in the design of discrete, typically, computer-based systems using the ***VDM-RT*** notation. VDM-RT is based upon the ***object-oriented*** paradigm where a model is comprised of one or more ***objects***. An object is an instance of a ***class*** where a class gives a definition of zero or more ***instance variables*** and ***operations*** an object will contain. Instance variables define the identifiers and types of the data stored within an object, while operations define the behaviours of the object.

The ***20-sim***[3] tool can represent continuous time models in a number of ways. The core concept is that of connected ***blocks***. ***Bond graphs*** may implement blocks. Bond graphs offer a domain-

---

[2]http://overturetool.org/
[3]http://www.20sim.com/

independent description of a physical system's dynamics, realised as a directed graph. The vertices of these graphs are idealised descriptions of physical phenomena, with their edges (**bonds**) describing energy exchange between vertices. Blocks may have input and output **ports** that allow data to be passed between them. The energy exchanged in 20-sim is the product of **effort** and **flow**, which map to different concepts in different domains, for example voltage and current in the electrical domain.

OpenModelica[4] is an open-source **Modelica**-based modelling and simulation environment. Modelica is an "object-oriented language for modelling of large, complex, and heterogeneous physical systems" [FE98]. Modelica models are described by **schematics**, also called **object diagrams**, which consist of connected components. Components are connected by ports and are defined by sub components or a textual description in the Modelica language.

Modelio[5] is an open-source modelling environment supporting industry standards like UML and SysML. INTO-CPS will make use of Modelio for high-level system architecture modelling using the **SysML** language and proposed extensions for CPS modelling. The systems modelling language (SysML) [Sys12] extends a subset of the UML to support modelling of heterogeneous systems.

8 Formalisms

The **semantics** of a language describes the meaning of a (grammatically correct) program [NN92] (or model). There are different methods of defining a language semantics: **structural operational semantics**; **denotational semantics**; and **axiomatic semantics**.

A structural operational semantics (SOS) describes how the individual steps of a program are executed on an abstract machine [Plo81]. An SOS definition is akin to an interpreter in that it provides the meaning of the language in terms of relations between beginning and end states. The relations are defined on a per-construct basis. Accompanying the relations are a collection of semantic rules which describe how the end states are achieved. Where an operational semantics defines how a program is executed, a denotational approach defines a language in terms of denotations, in the form of abstract mathematical objects, which represent the semantic function that maps over the inputs and outputs of a program [SS71].

The Unifying Theories of Programming (UTP) [HJ98] is a technique to for describing language semantics in a unified framework. A theory of a language is composed of an **alphabet**, a **signature** and a collection of **healthiness conditions**.

The Communicating Sequential Processes **CSP** notation [Hoa85] is a formal process algebra for describing communication and interaction. **INTO-CSP** is a version of CSP, which will be used to provide a model for the SysML-FMI profile, FMI, VDM-RT and Modelica semantics. It is a front end for a UTP theory of reactive concurrent continuous systems customised for the needs of INTO-CPS. **Hybrid-CSP** is a continuous version of CSP defined originally by He Jifeng [Jif94]. It will be used as a basis to inform the design of INTO-CSP.

Several forms of verification are enabled through the use of formally defined languages. **Refinement** is a verification and formal development technique pioneered by [BW98] and [Mor90]. It is based on a behaviour preserving relation that allows the transformation of an abstract speci-

---

[4]https://www.openmodelica.org/
[5]http://www.modelio.org/

fication into more and more concrete models, potentially leading to an implementation. ***Proof*** is the process of showing how the validity of one statement is derived from others by applying justified rules of inference [BFL$^+$94].

For the purposes of verification in INTO-CPS, and in particular the work of WP2, we make use of the Isabelle/HOL theorem prover and the FDR3 refinement checker. These are not considered part of the INTO-CPS tool chain, and are used in the INTO-CPS project primarily to support the development of foundation work.

Getting Started with INTO-CPS

This chapter should help you become familiar with the possibilities for collaborative, model-based design offered by the INTO-CPS tool chain. It does this by explaining the types of activities that can be undertaken with support of one or more of the INTO-CPS technologies, and hopefully putting some of the concepts from the previous chapter in context.

Performing one or more of these activities in order, possibly with iterations, forms a "workflow" for using the INTO-CPS technologies. There are many potential workflows, which depend on the users background and intended use for the tools. A key aspect of most workflows is to produce a multi-model, therefore this chapter includes some guidance.

9   Activities Enabled by INTO-CPS

The following activities are all enabled by one or more of the INTO-CPS technologies. They are grouped into broad categories and include both existing, embedded systems activities and activities enabled by INTO-CPS, since INTO-CPS extends traditional embedded systems design capabilities towards CPS design. The choice of granularity for defining these activities naturally affects the size of such a list. The level chosen is instructive for describing workflows, but one that does not make the described workflows overly long.

In the following descriptions (and corresponding summary in Table 1), we identify the tools that support the activities, where applicable, using the following icons:

The INTO-CPS Application, COE and its extensions.
Modelio.
The Overture tool.
RT-Tester.
OM OpenModelica.
20-sim.

Descriptions of these tools can be found in the concepts base at the beginning of this document in Section 7. Those activities in *italics* can be recorded by the traceability features of INTO-CPS, which is described in Chapter II.

Requirements and Traceability   Writing *Design Notes* ( ) includes documentation about what has been done during a design, why a decision was made and so on. *Requirements* ( ) includes requirements gathering and analysis. *Validation* ( ) is any form of validation of a design or implementation against its required behaviour.

Architectural Modelling   INTO-CPS primarily supports architectural modelling in SysML. *Holistic Architectural Modelling* ( ) and *Design Architectural Modelling* ( ) are described in Section 18. The former focuses on a domain-specific view, whereas the latter targets multi-modelling using a special SysML profile. The *Export Model Descriptions* ( ) activity indicated passing component descriptions from the Design Architectural Model to other modelling tools.

Modelling   The *Import Model Description* (OM) activity means taking a component interface description from the Design Architectural Model into another modelling tool. *Cyber Modelling* ( ) means capturing a "cyber" component of the system, e.g. using a formalism/tool such as VDM/Overture. *Physical Modelling* (OM) means capturing the "physical" component of the system, e.g. in 20-sim or OpenModelica. Collectively, these can be referred to as *Simulation Modelling* (OM) to distinguish from other forms, such as *Architectural Modelling* ( ). *Co-modelling* ( ) means producing a system model with one DE and one CT part, e.g. in Crescendo. *Multi-modelling* ( ) means producing a system model with multiple DE or CT parts with several tools.

Design   *Supervisory Control Design* means designing some control logic that deals with high-level such as modal behaviour or error detection and recovery. *Low Level Control Design* means designing control loops that control physical processes, e.g. PID control. *Software Design* is the activity of designing any form of software (whether or not modelling is used). *Hardware Design* means designing physical components (whether or not modelling is used).

Analysis   In INTO-CPS, the RT-Tester tool enables the activities of *Model Checking* ( ), *Creating Tests* ( ) and creating a *Test Oracle* ( ) FMU. The *Create a Configuration* ( ) activity means preparing a multi-model for co-simulation. The *Define Design Space Exploration Configurations* ( ) activity means preparing a multi-model for multiple simulations. *Export FMU* (OM) means to generate an FMU from a model of a component. *Co-simulation* ( ) means simulating a co-model, e.g. using Crescendo baseline technology or the COE.

Prototyping   *Manual Code Writing* means creating code for some cyber component by hand. *Generate Code* (OM) means to automatically create code from a model of a cyber component. *Hardware-in-the-Loop (HiL) Simulation* ( ) and *Software-in-the-Loop (HiL) Simulation* ( ) mean simulating a multi-model with one or more of the models replaced by real code or hardware.

Table 1: Activities in existing embedded systems design workflows or enhanced INTO-CPS workflows.

Requirements Engineering	
Stakeholder Documents	
Requirement Definition	
Validation	
Architectural Modelling	
Holistic Architectural Modelling	
Design Architectural Modelling	
Export Model Descriptions	
Modelling	
Import a Model Description	OM
Physical Modelling (Simulation Modelling)	OM
Cyber Modelling (Simulation Modelling)	
Co-modelling	
Multi-modelling	
Design	
Supervisory Controller Design	
Low Level Controller Design	
Software Design	
Hardware Design	
Analysis	
Create Tests	
Model Checking	
Create Test Oracle	
Create a Configuration	
Define Design Space Exploration Configurations	
Export FMU	OM
Co-simulation	
Prototyping	
Generate Code	OM
Hardware-in-the-Loop (HiL) Simulation	
Software-in-the-Loop (SiL) Simulation	
Manual Code Writing	

10 Configuring Multi-Models

As discussed in Chapter 2, a multi-model is a collection of FMUs with a configuration file that: defines instances of those FMUs, specifies connections between the inputs/outputs of the FMU instances, defines values for design parameters of the FMU instances, and defines other simulation settings such as a start, end time, and Master algorithm settings. As seen above, creating a multi-model is a key part of using the INTO-CPS tool chain as it is a pre-requisite for many of the analysis techniques that INTO-CPS can perform.

The INTO-CPS Application supports a project, a view of a folder containing source models, generated FMUs, and configuration files for co-simulation (multi-models) as well as configuration files for other analyses (design space exploration, model checking, test automation). Multi-model configurations can be created in three ways:

1. Created manually using the GUI of the INTO-CPS Application; or
2. Generated from a SysML model created in Modelio; or
3. Created manually by editing JSON configuration files

All three approaches produce the same configuration file, so the choice of which to use depends on the engineer's background. Those comfortable with SysML may find it best to follow the SysML route, but this is not required. So those unfamiliar with SysML can use the Application directly. These two approaches are covered in the second and third tutorials in Part **??**. Manually editing the JSON configuration is an advanced topic that is not covered in the tutorials, but since JSON is human-readable, not complicated with some experimentation.

11 First Steps for Users

In this final section of this chapter, and of Part I, we consider a how different types of users might approach the INTO-CPS technologies. As described in Section I, all new users are recommended to:

- Follow the first tutorial (see Part **??**) to experience the INTO-CPS Application.
- Import one or two examples from the Examples Compendium (Deliverable D3.6 [MGP$^+$17]) into the INTO-CPS Application and interact with them.
- Return to Part II document as and when you require guidance on a particular area.

After initial familiarisation, the following list provides hints on next steps for different types of users, and where to find further information. As a reminder, tutorials are found in Part **??**[6].

Students Bachelor and Masters students wishing to build multi-models should follow the first few tutorials on adding exporting and adding FMUs. The SysML tutorial can be skipped if desired. Further guidance on exporting FMUs from different tools can be found in the User Manual, Deliverable D4.3a [BLL$^+$17].

Individual Engineers Engineers should follow the first few tutorials on adding and exporting FMUs. The SysML tutorial is also recommended. Further guidance on exporting FMUs from different tools can be found in the User Manual, Deliverable D4.3a [BLL$^+$17]

Engineering Teams Teams requiring traceability must read Chapter II first (and Chapter 14 is also recommended), as traceability must be considered from the outset. The SysML

---

[6]Updated tutorials supporting newer versions of the tool can be found at `https://github.com/INTO-CPS-Association/training/releases`

tutorial is mandatory, because traceability links begin with requirements and architectural models in Modelio.

Those with Legacy Models A primary goal is to generate an FMU from the tool for your existing models. These can be incorporated into multi-models as described in the second tutorial.

Those wishing to run Design Space Exploration It is necessary to build a multi-model first in order to run a DSE, so the first tutorials should be followed. The SysML tutorial is optional, though useful as the SysML profile includes extensions to help configure DSE analyses. The later tutorials cover DSE, with further guidance in the user manual, Deliverable D4.3a [BLL$^+$17], and Deliverable D5.3e [Gam17] (for more technical details).

Those interested in model checking The User Manual, Deliverable D4.3a [BLL$^+$17], provides useful insight, with in-depth information found in Deliverable D5.3c [BH17].

Those interested in formal semantics and analysis The collection of D2.3deliverables [ZCWO17, CFT$^+$17, ZFC$^+$17, CFWZ17] provides in-depth information on these aspects of the tool chain, including mechanisation efforts in Isabelle.

Part II

Advanced Topics

Traceability and Provenance

The technologies in the INTO-CPS tool chain are able to automatically capture traceability information as activities are performed using the various parts in the tool chain. This includes information about who created or modified an artefact (model, simulation result etc.) and which requirements it is linked to. The traceability features of the INTO-CPS tool are powerful, but require a specific workflow to be followed in order to make best use of them. This chapter explains the steps in this workflow.

This chapter appears first in this advanced material as the following chapters, in particularly Chapters 14 and 18, provide key guidance on the first part of the workflow that must be followed in order for traceability to be realised. Those not wishing to use the traceability features can read chapters in any order, driven by their needs or interest. This chapter should be used in conjunction with the User Manual (Deliverable D4.3a [BLL+17]), which covers details of how to enable traceability recording in the INTO-CPS Application and baseline tools[7]. Readers interested in detailed specifications of the traceability and provenance features are directed to Methods Progress Report (Deliverable D3.3b [FGP17]), while the tool implementation is described in Deliverables D4.2d [LNH+16] and D4.3d [KLN+17].

12 Traceability Workflow

The INTO-CPS tool chain builds a graph of traceability relations, as there can be multiple relationships between different artefacts. The graph is however tree-like in the sense that there must be some root node(s) to trace from or back too. These root nodes are *requirements*. To use fully the machine-assisted traceability features, it is necessary to initialise the traceability graph by using Modelio from the beginning of the development process. This means that it is necessary to follow these steps:

1. Define requirements through some requirements process (see guidance in Chapter 14);
2. Create a Requirements Diagram (RD) in Modelio representing these requirements;
3. Create an Architecture Structure Diagram (ASD) and Connections Diagram (CD) describing the multi-model;
4. Link each requirement to one «EComponent» (FMU);
5. Export model descriptions for each «EComponent»;
6. Import model descriptions into baseline tools; and
7. Generate a multi-model configuration from the CD.

After these steps, the traceability graph will then be updated by the baseline tools as models are created from the model descriptions, FMUs are exported and so on, and co-simulation runs and results will be recorded by the INTO-CPS Application. Therefore, by following this workflow it is possible to take advantage of the machine-assisted traceability within INTO-CPS. By performing the required manual input of requirements and links to SysML elements, it is then possible to automatically trace forward to models, FMUs and simulation results, and to trace backwards from these artefacts to individual requirements.

---

[7]Traceability is turned off by default as it can be intrusive if the right workflow is not followed.

13 What Artefacts are Traced?

Traceability in the INTO-CPS tool chain is based upon a study of the actions performed when using the INTO-CPS tool chain, the artefacts that are used and produced and a combination of two existing standards, the W3C's Prov [8] and the OMG's OSLC [9]. The combination of these resulted in the INTO-CPS traceability ontology that captures in detail all elements in the INTO-CPS workflow and describes the relationships between them. The complete ontology is presented in deliverable D3.3b [FGP17] and a summary is presented here.

Traceability data is inherently a graph based structure based upon nodes and the connections between them, and Prov provides basic types for those nodes along with list of relationships that may exist between them. The three types of nodes are: Entities, things that may be produced or used during a development process; Activities, are things that act upon and make use of entities; and Agents, objects that have responsibility for entities and activities. The Prov relations then allow then connection of nodes such as an activity may use an entity, and an entity may be generated by an activity.

The combination of the Prov nodes and relations supports the representation of the processes that lead to the generation of a particular entity, but it does not support connection of those entities to requirements. OSLC contains a set of specifications, each of which defines a list of relations that it supports between entities. In the case of the INTO-CPS traceability, parts of the OSLC architecture management and requirements management specifications are employed, these allow the connection of entities to requirements via a 'satisfies' relation indicating the entity attempts to address the needs of the requirement, additionally it allows the connection of simulation results to requirements via a 'verifies' relation indicating that the requirement has been met.

The INTO-CPS traceability ontology breaks the INTO-CPS workflow down into activities that, while not atomic if we consider a user's interaction with a particular tool, could be considered atomic when viewing the process of developing a CPS. Figure 1 shows the traceability links recorded during one step in the development of a line following robot. In this example, the requirements, R1 & R2, already exist in the architecture models and the user has created an ASD to decompose the proposed robot into components. The user has, at the same time, associated the blocks within the ASD with the the requirements that each block aims to satisfy. When the user saves the updates architecture model, the Modelio tool records the user's 'Architecture Modelling' activity, along with references to the ASD, the blocks it contains and the newly created links between the blocks and the requirements. Here the *used*, *wgb* (short for 'was generated by'), *assoc* (short for 'associated with') and *attrib* (short for 'attributed to') are links that come from the Prov standard. The *OSLC_Sat* (short for 'satisifies') comes from the OSLC requirements management specification.

A development project will likely consist of many instances of the activities identified in the ontology being performed and together they form a traceability graph. Figure 2 shows a simplified view of a traceability graph with some steps removed for brevity. At the top of the graph we see the architecture modelling step described previously, that produces an architecture model. From the architecture, model description files are exported to start the production of the simulation models. In turn the simulation models are exported as FMUs and the FMUs are used to produce simulation results. Key to the traceability graph then are the 'used' and 'wgb'

---

[8]https://www.w3.org/TR/prov-overview/
[9]http://open-services.net/
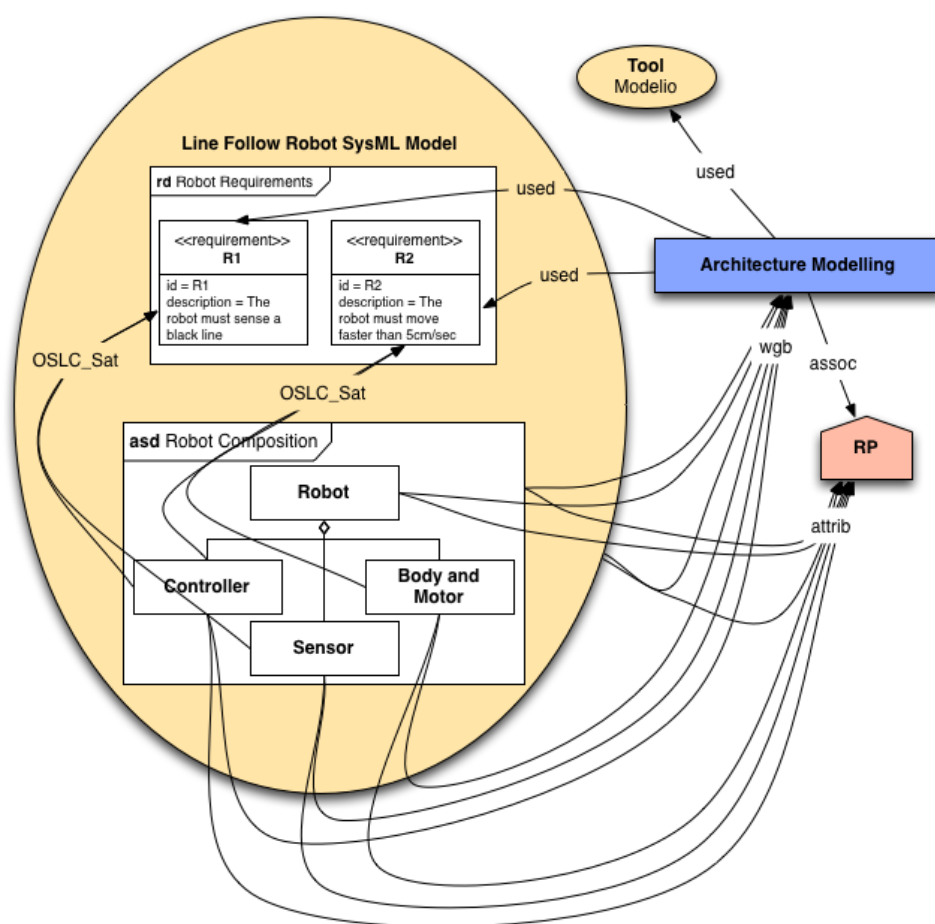
Figure 1: Traceability links captured during the production of an ASD for a line following robot.

connections that can be used by a query to determine from where each entity was generated. By following these links back from any entity to the individual blocks within the architecture model, it is possible to determine which requirement(s) each should satisfy. Finally when simulation results are output, these may be linked back to the relevant requirements, stating whether a requirement was verified or violated by that result.
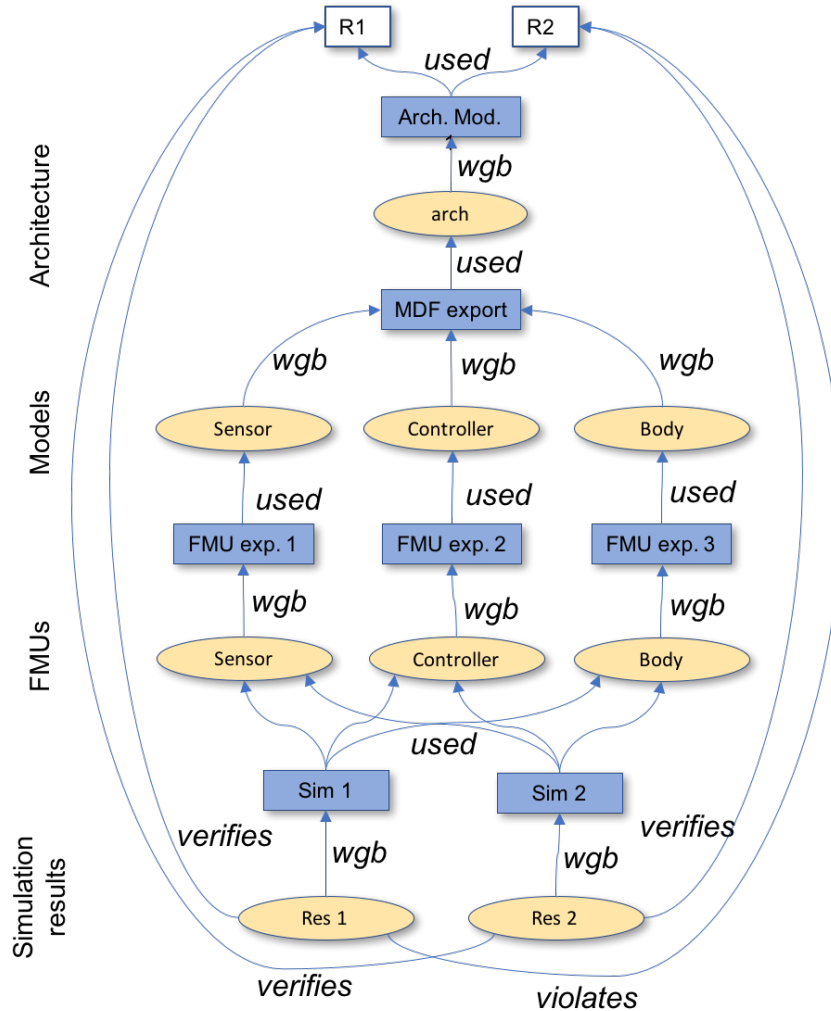
Figure 2: Traceability links captured during the production of an ASD for a line following robot.

The traceability ontology captures the significant activities and entities that form the INTO-CPS workflow. For example a development project might see the following activities recorded in the traceability graph: *Requirements Management*, *Architecture Modelling*, *Architecture Configuration Creation Model Description Export*, *Simulation Modelling*, *FMU Export*, *Configuration Creation*, *Simulation Configuration Creation* and then *Simulation*. These activities are connected in the workflow by the entities they create and use, so the example would see the traceability graph containing records of: *Archtecture Structure Diagram*, *Architecture SubSystem*, *Architecture Connection Diagram*, *Model Description File*, *Simulation Model*, *FMU*, *Multimodel Configuration*, *Simulation Configuration* and *Simulation Result*. Alongside these will be records of the agent(s), who are both associated with activities and have entities attributed to them.

14   Traceability Queries

The traceability graph created by the INTO-CPS tool chain uses a graph database tool called *Neo4J*. Once a graph has been built, queries can be executed over the graph to perform both forwards and backwards traceability. Below are some types of queries that can be executed over the graphs. The INTO-CPS Application supports some of these queries with the GUI, and the rest through inline access to the Neo4J console.

1. Impact analysis
   - Forward traceability (from requirements to entities)
   - Backwards traceability (from FMU to requirements)
   - Backwards traceability (from components to requirements)
2. Simulation sources
   - Find all simulations
   - Find sources and sinks for a simulation
3. Coverage
   - Requirements without architecture elements
   - Requirements without simulation models
   - Requirements without FMUs
   - Requirements without positive simulation results
   - Requirements without any simulation results
4. Code sources
   - Find all generated source code entities
   - Find the models for a given source code entity
5. User impact
   - Find all users in the database
   - Find all artefacts influenced by a user
   - Find all activities performed by a user

Queries are written in Cypher, a query language built into Neo4J. Advanced users or those developing extensions to INTO-CPS can build their own queries in Cypher[10] and execute them using Neo4J directly as described in the User Manual (Deliverable D4.3a [BLL+17]).

---

[10]https://neo4j.com/developer/cypher-query-language/

Requirements Engineering

In this chapter, we consider the requirements engineering (RE) activities for the design of CPSs. Specifically, we consider the specification and documentation of requirements placed upon a CPS. These requirements may, for example, impose restrictions, define system capabilities or identify qualities of a system. The requirements should indicate some value or use for the different stockholders of a CPS.

As described in the previous chapter, traceability needs requirements to be defined as early as possible in a development process, and these must be recorded in Modelio for the machine-assisted traceability information to be recorded accurately. It is therefore appropriate to consider requirements processes for such developments at this stage.

In this remainder of this chapter, we discuss the needs for requirements engineering in CPS development, in particular based on the experience of the industrial partners for INTO-CPS. We describe one possible approach to RE for CPS, specifically adapting the SoS-ACRE approach for systems-of-systems (SoS) to CPS. Note however that this approach is not mandatory, and in general RE processes and tools vary widely across organisations and domains. For this reason, tool support for traceability in INTO-CPS begins once requirements have been defined and can be added to Modelio. The diagrams described in the example are not part of INTO-CPS SysML specification. Therefore, this chapter should truly be treated as guidance, primarily serving to highlight the nature of RE for CPS, which may be of use for both new and more experienced CPS teams.

15 Requirements Engineering and Cyber-Physical Systems

The main issue of concern for RE in CPSs is that of differing domain contexts [WGS$^+$14]. In addition, it has been noted that there are overlaps in challenges in CPSs and SoSs [PE12]— especially independence, evolution and, increasingly, distribution. As described by Lewis et al. [LMP$^+$09], as system architectures become more complex, there is often a need to consider requirements and structural architectures during the RE process. The authors suggest that an engineer should identify the system needs, component interactions and stakeholders, and map those needs onto those interested parties.

As research in RE in CPS is a nascent field, we suggest one approach is to adopt RE processes from the SoS world, rather than defining an approach specifically for CPSs. In chapter, we consider SoS-ACRE (System of Systems Approach to Context-based Requirements Engineering) [HPP$^+$15], as an example. This approach was adapted from standard systems engineering, and tailored for SoSs— enabling the identification and reasoning about requirements across constituent systems of an SoS and understanding multi-stakeholder contexts. We suggest it might be useful to organisations trying to approach RE for CPS.

INTO-CPS industry partners and RE

At the beginning of the INTO-CPS project, the four industrial partners were surveyed about their use of various technologies and methods, including requirements engineering [FGPP15b]. Microsoft Excel was quoted as being used by three partners (UTRC, TWT and CLE), IBM Rational Doors used by one partner (UTRC), and Microsoft Word by one partner (AI).

Issues raised by industrial partners include:

- Language/terminology of the requirements not consistent;

- Different people involved in the workflow do not have common understandings of requirements;

- Requirements traceability is considered to be highly inefficient and time consuming;

- Different people have to meet together and generate proofs among each other to validate dependable requirements; and

- Stakeholders do not have a clear vision about the product and tend to disagree on the objectives.

As can be seen, the above issues may be due to not having a rigorous RE approach, but also due to the challenges in CPSs— that of different domains. In this section, we consider how a context-based approach to RE (SoS-ACRE) may be incorporated into the INTO-CPS tool chain, in particular using both the INTO-CPS technologies and the industrial partners' baseline technologies.

16   The SoS-ACRE View of Requirements

We first consider the collection of views defined in SoS-ACRE, and their applicability to CPS engineering and the INTO-CPS tool chain. These views could be represented as diagrams in SysML[11], or as we describe, could equally be represented in other tools where these are already used (e.g. Excel). Examples of each view are shown in Figures 5, 6, 3 and 4.

Source Element View (SEV) The SEV defines a collection of source materials from which requirements are derived. In SoS-ACRE, a SysML block definition diagram is considered. In INTO-CPS, this view could also be represented using an Excel table or Word document (with each source having a unique identifier), or by simply referring to source documents using OSLC traces.

Requirement Description View (RDV) The RDV is used to define the requirements of a system and forms the core of the requirement definition. SoS-ACRE suggests the use of SysML requirements diagram or in tabulated form, such as through the use of Excel. In addition, specifying requirements in Doors would support this view.

Context Definition View (CDV) The CDV is a useful view for CPS engineering in order to explicitly identify interested stakeholders and points of context in the system development, including customers, suppliers and system engineers themselves. In SoS-ACRE, they are defined using SysML block definition diagrams, and could also be represented using an Excel table or Word document (with each context having a unique identifier). This diagram type could be useful when identifying the divide in CT/DE and cyber-physical elements of a system.

Requirement Context View (RCV) In SoS-ACRE, a RCV is defined for each constituent system context identified in CDVs. This is appropriate when there is a set of diverse system owners, which is typical for SoSs and increasingly CPSs. A **Context Interaction View (CIV)** is then defined to understand the overlap of contexts and any common/conflicted views on requirements. In a CPS, however, there may not be such a clear delineation

---

[11]Note that SoS-ACRE is not specifically supported as a Modelio plug-in, but other equivalent diagrams could be used.

between the owners of constituent system components. However, if we consider the different domains (e.g. CT/DE or cyber/physical divides) as different contexts, then this approach would be useful. In SoS-ACRE, RCVs and CIVs are both defined with SysML use case diagrams. Excel could be used if unique identifiers are defined for contexts and requirements as described earlier.

Validation View (VV) VVs, defined as SysML sequence diagrams in SoS-ACRE, describe validation scenarios for a SoS to ensure each constituent system context understands the correct role of the requirements in the full SoS. This is not an obvious fit in CPS engineering, and therefore not necessarily required.

17   The SoS-ACRE RE Process

The SoS-ACRE requirement engineering process may be useful for organisations wishing to better understand requirements for CPSm, particularly across multiple domains. It is a lightweight process, and therefore suitable for small- to medium-sized enterprises. Organisations with established may not feel the need to radically alter their existing practice, but may find it instructive to consider how their current processes might be updated or revised to consider better CPS requirements.

A SoS-ACRE process for CPS should include the following steps:

1. Identify and record source elements. This would be using a SEV, or simply recording paths to relevant files or documents.

2. Record system-level functional and non-functional requirements. Requirements may be derived using RDVs, and we could consider domain-specific requirements (e.g. cyber or physical), or analysis-specific requirement types (e.g. DSE or testing requirements).

3. Model initial System structure using INTO-CPS ASD. This will identify the cyber and physical elements and the domain/phenomena of the CPS. This may also give initial idea of component functionalities, which may lead to a repeat of Step 2 above[12].

4. Define the various contexts in CDVs – both external stakeholders, and if appropriate, contexts for the different components. If only a single system context is defined, then a single RCV is defined. However, if multiple contexts are defined for a CPS, then several RCVs are to be defined, along with a CIV to explore requirements from multiple contexts.

5. Trace the requirements through INTO-CPS tool chain models and results. This was covered in the previous chapter, however we revisit it below in the context of requirements.

18   Using technologies with SoS-ACRE

As INTO-CPS does not specifically support SoS-ACRE. Indeed INTO-CPS does not mandate and specific approach to RE, because of the wide variety of approaches in industry. We conclude this chapter with an example of how a SoS-ACRE (or other RE process) could be integrated into an INTO-CPS development. We describe a range of permutations of the use of models and documents for recording the requirements engineering process described above. In

---

[12]In the process of architectural modelling, it may also be necessary to redefine contexts depending on whether different simulation tools, or indeed different components of a model, are better able to provide the requirements of the CPS.

addition, we include discussions on the links between requirements and architectural models—identified above as a key method for requirements engineering in CPSs.

URI, Excel and SysML We first consider an approach using URIs for the source elements, an Excel document (or a collection of Excel tables) for the RDV, CDV, RCV and CIV of SoS-ACRE. A SysML model in Modelio can be used to define the architecture of the multi-model. Internal tracing in Excel can be achieved using identifiers referenced between sheets. Excel requirements can be replicated in Modelio then traced to elements in the INTO-CPS tool chain automatically. Figure 3 presents an example with URI, Excel and SysML models and OSLC links between the artefacts.

Excel and SysML The next approach uses Excel to define the SEV and RDV of SoS-ACRE, a SysML model to define the context-oriented views (CDV, RCV and CIV) and a separate architectural model to define the CPS architecture. The Excel requirements can then be mirrored in a Modelio model, and linked to the architectural model. The INTO-CPS traceability features can trace the requirement artefacts to the architectural model. Additional OSLC links could be added manually to link elements of the Excel requirements and context views in a SysML. Figure 4 presents an example with URI, Excel and two SysML models with OSLC links between the artefacts.

Single SysML model The next permutation is to use a single SysML model for both requirements engineering and architectural modelling. Such a model will contain all SoS-ACRE views[13] (SEV, RDV, CDV, RCV and CIV), in addition to diagrams defined using the INTO-CPS profile for the CPS composition and connections. Modelling in this way enables trace links to be defined inside a single SysML model. Figure 5 presents an example SysML model with trace relationships.

SysML requirements and SysML architectural models The final permutation is to use SysML for both requirements engineering and architectural modelling, however to use two separate models for the two activities (one containing the RE views (SEV, RDV, CDV, RCV and CIV) and another for architectural diagrams (ASD and CD)). We consider this permutation with two SysML models in addition to the single SysML model, because the requirements engineering and architectural modelling activities are often considered separately, with different engineering teams comprised of engineers with specialist skills. As such we can assume there are cases where these teams have ownership of different models. Trace links may be used within each individual model (for example, tracing from source elements to requirements in a RE model), and OSLC links defined to trace between requirements elements and architectural elements. Figure 6 presents an example with two SysML models with trace relationships and OSLC links between the models.

---

[13]Note that Modelio does not currently provide an extension for SoS-ACRE, but these views can be realised using existing SysML stereotypes.
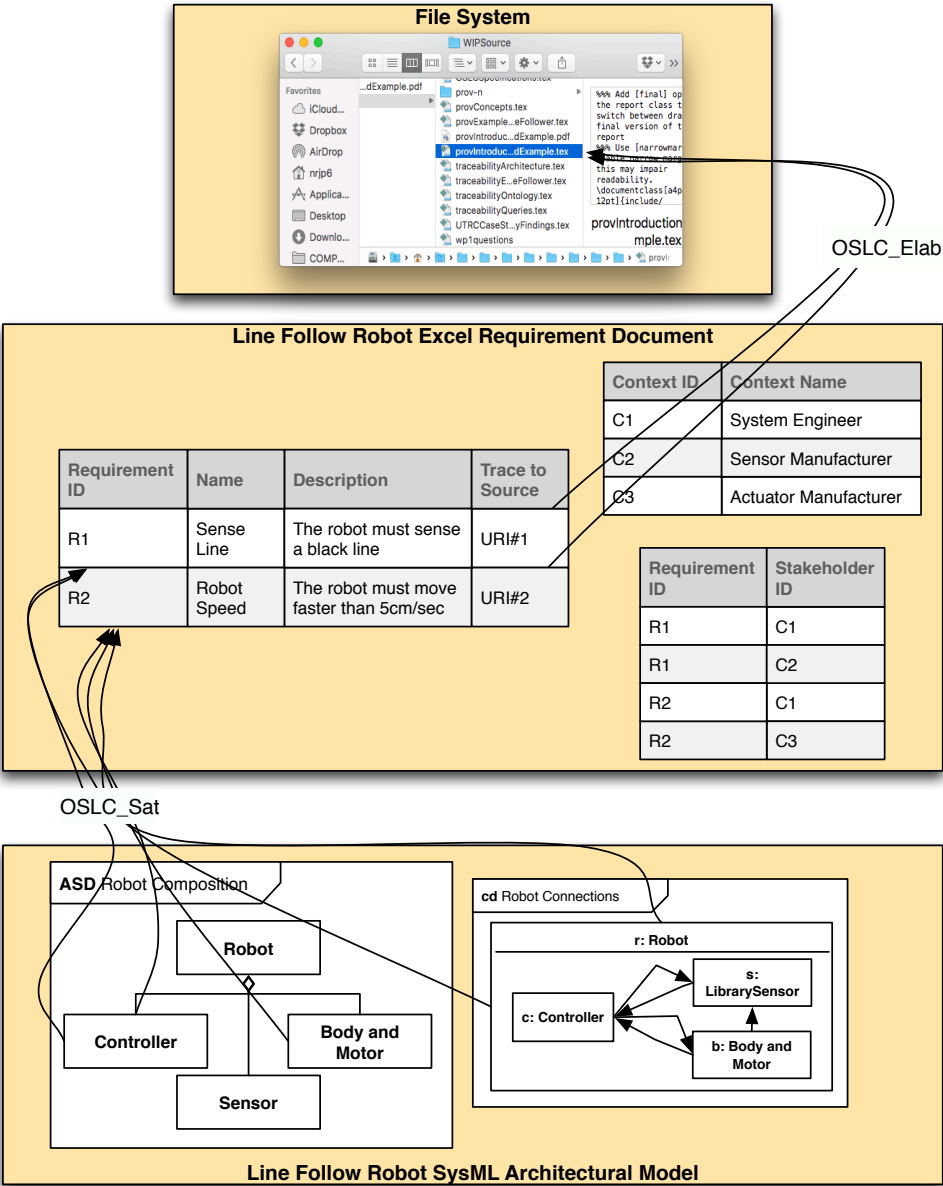
Figure 3: URI, Excel and SysML – model overview
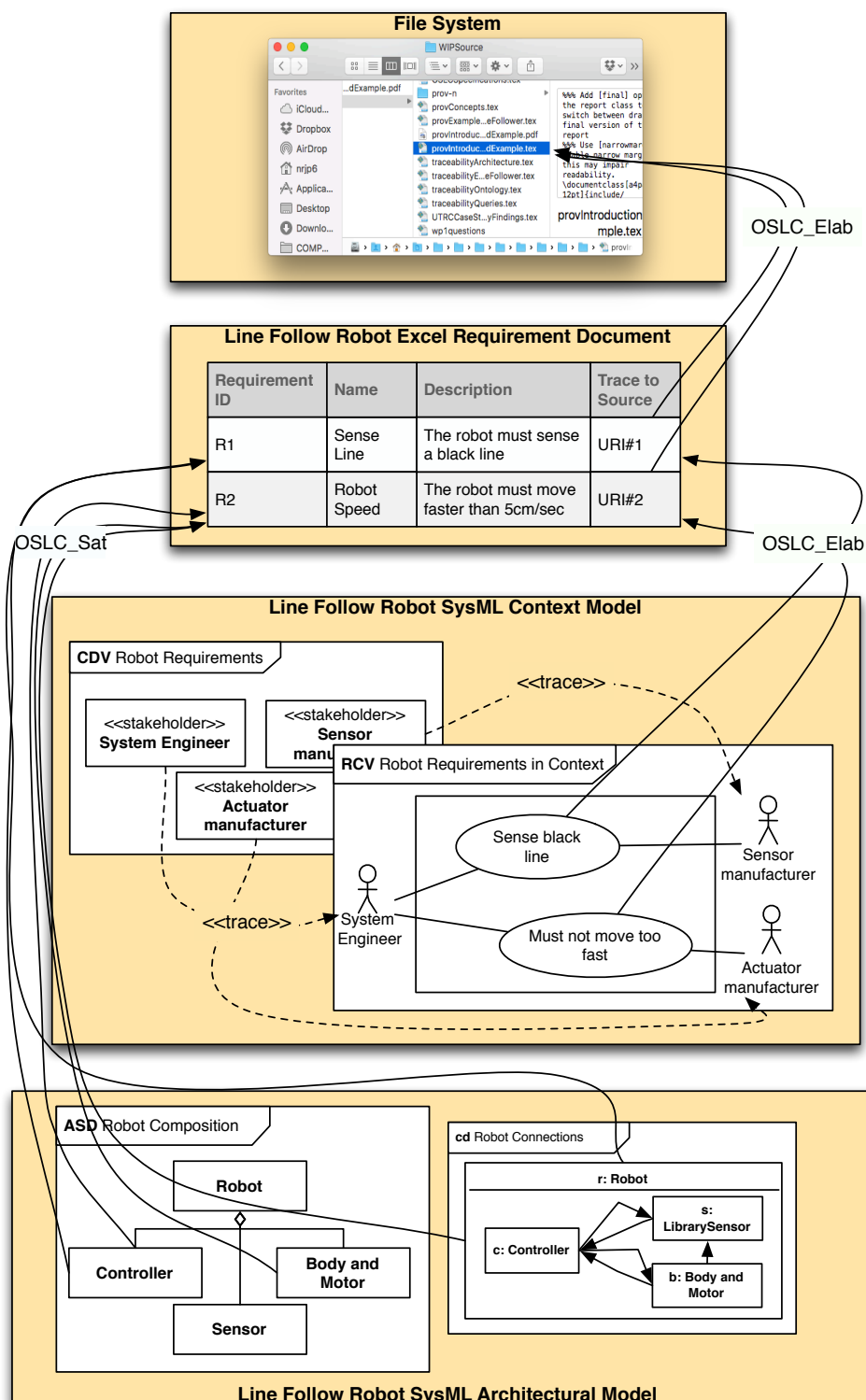
Figure 4: Excel and SysML – model overview

Line Follow Robot SysML Requirement and Architectural Model
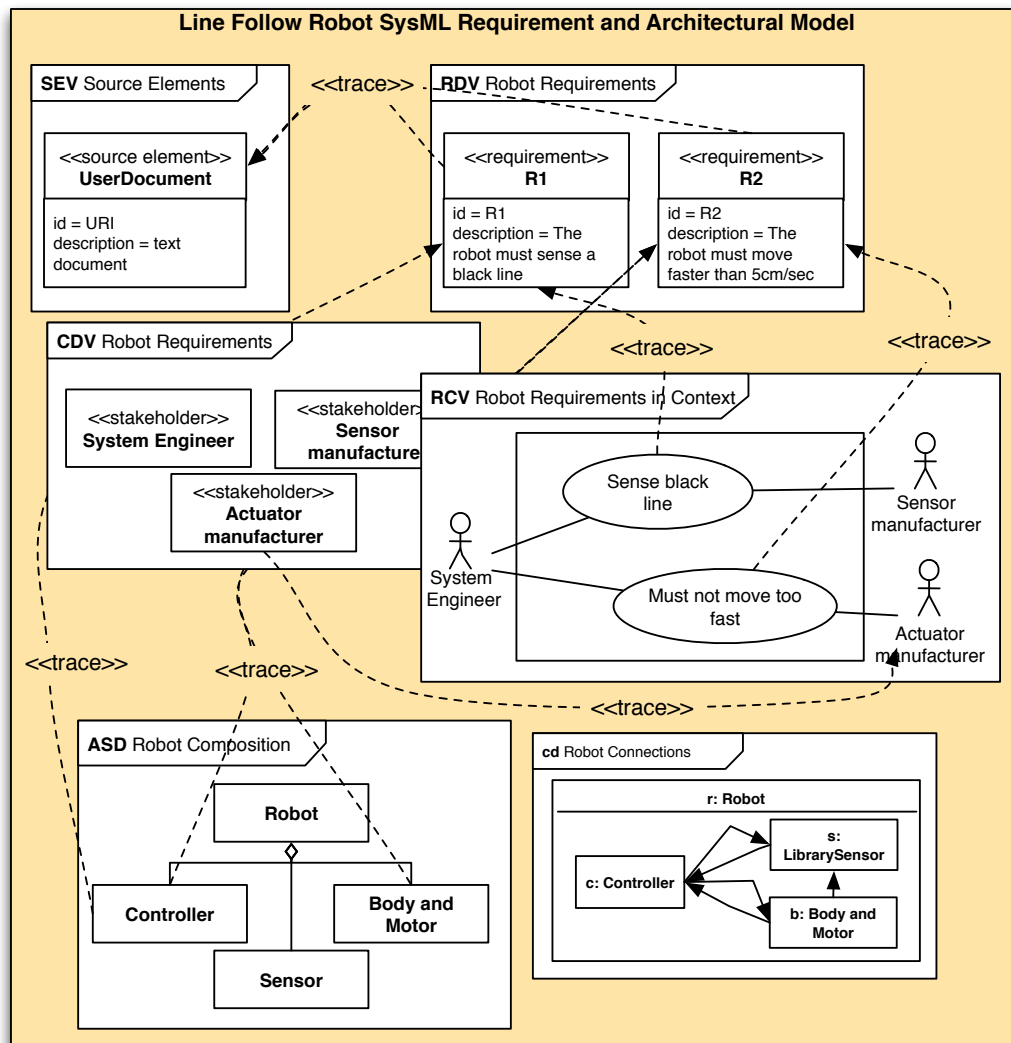
Figure 5: Single SysML model – model overview
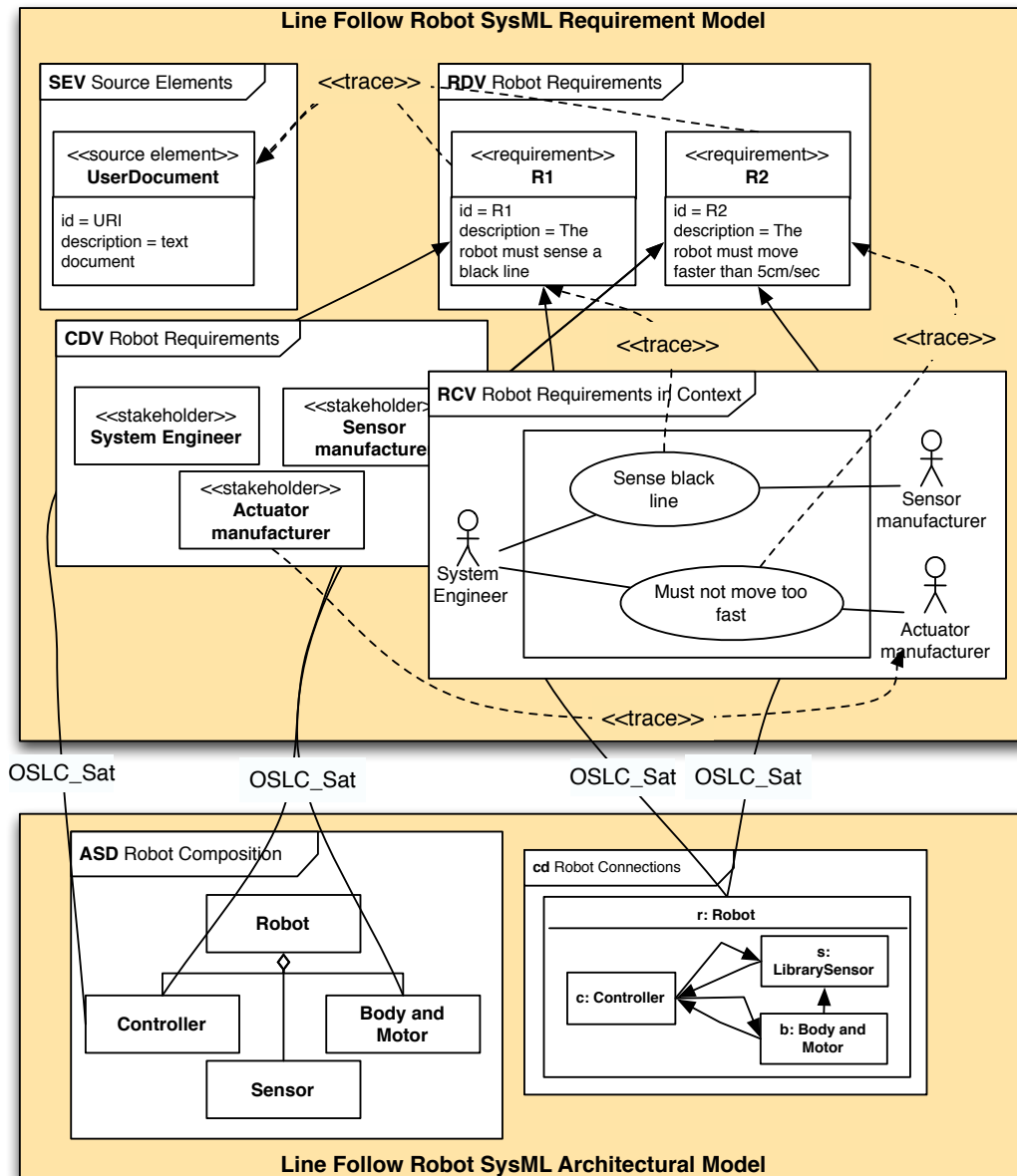
Figure 6: SysML requirements and SysML architectural models – model overview

SysML and Multi-modelling

This chapter describes the use of SysML with the INTO-CPS tool chain. As described previously in Chapter 14, standard SysML can be used as part of a development process to build a model of a system and link elements to requirements. The INTO-CPS tool chain also provides an extended SysML profile that help users to *configure multi-models for co-simulation* and *configure design space exploration (DSE) analysis* [APCB15, ACM+16, ZCWO17, BQ15, BQ16, Bro17]. For ease explanation, we describe these separately below, however all the diagrams described are part of a single extended SysML profile.

This chapter summarises the diagrams provided in the two profiles and describe their use in Sections 19 and 20. The diagrams presented are illustrative, showing the main elements of a diagram; they are not full definitions of the meta-model, which can be found in the documents cited above. All diagrams are supported by the Modelio tool, and we refer readers to the user manual, Deliverable D4.3a [BLL+17], for further information on how to use Modelio to draw these diagrams and generate configurations for use in the INTO-CPS Application.

The chapter concludes with an example of the relationship between a *holistic* model created using standard SysML and a *design* model using the INTO-CPS profile, and concludes with a discussion on how to represent non-design elements (such as FMUs that only perform visualisation) in the INTO-CPS profile in Section 22.

19 SysML Diagrams Describing Multi-models

The multi-modelling SysML profile defines two diagrams for configuring a co-simulation. The INTO-CPS Application can run a co-simulation based on a configuration file, using the JSON format to describe the FMUs, their parameters and connections between them. These can be created manually in a text editor, or from the INTO-CPS Application itself. Alternatively, a configuration can be generated by Modelio from the diagrams defined in this profile. There are two types diagram, the *Architectural Structure Diagram* describing the static structure of FMUs, and the *Connections Diagram* describing their instantiation and connections. These are shown in Figure 7.
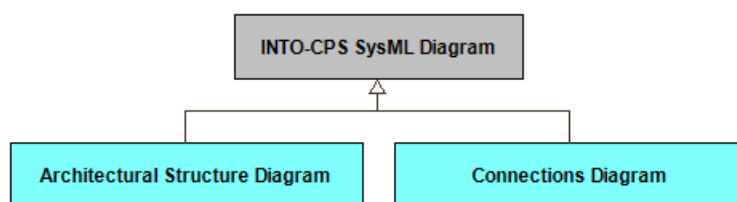
Figure 7: Diagrams in the multi-modelling SysML profile

19.1 Architectural Structure Diagram

The *Architecture Structure Diagram* (ASD) specialises SysML block definition diagrams (BDDs) to support the specification of a multi-model architecture described in terms of a systems components, which will be represented by FMUs. As shown in Figure 8 this diagram must include a «System» which is then broken down into zero or more «Component» blocks.
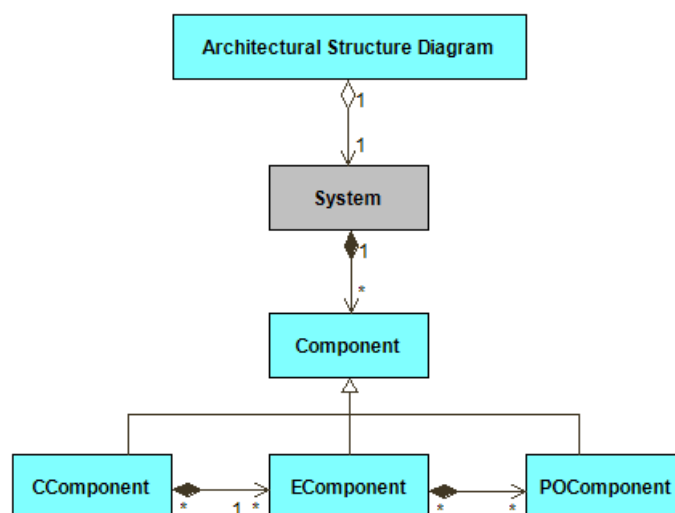
Figure 8: *Architectural Structure Diagram* describing FMUs (EComponents) and their hierarchies

There are three types of component block. The «EComponent» (encapsulating component) represents a part of a system that will be represented by a single FMU. These blocks have properties indicating which modelling language and tool will be used: modelType (*discrete* or *continuous*) and platform (*VDMRT*, *TwentySim*, *OM*, and *other*).

An «EComponent» can be broken down logically into «PComponent» (part-of component) representing an internal element of an «EComponent». Both «EComponent» and «PComponent» blocks can define *variables* and *FlowPorts* that an FMU will have.

The third type of component is a «CComponent» (collection component) that allows other components to be grouped logically (it has no ports or behaviours). These can be used to separate design elements within a diagram, as described in Section 22. All component blocks have a *kind* that marks their purpose in the model (*cyber*, *physical*, *environment*, *visualisation*).

FMUs are connected by *ports*, and may also present internal state through externally visible *variables*, which can be monitored on a live graph, for example. Both «EComponent» and «PComponent» blocks can define *FlowPort* and *Variable* attributes, as shown in Figure 9, which will form the interface of the FMU and are added to the "model description" exported by Modelio.

19.2 Connections Diagram

The *Connections Diagram* (CD) specialises SysML internal block diagrams to convey the internal configuration of the systems components. Specifically, it describes which FMUs are
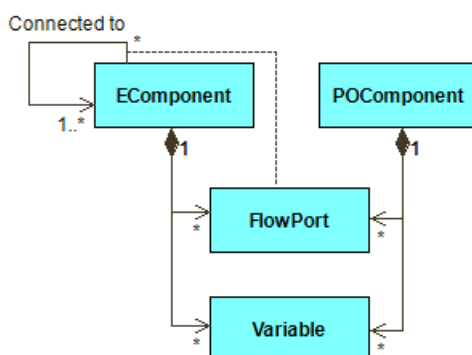
Figure 9: Component blocks may define variables and ports

instantiated (i.e. which «EComponent»s form the ASD), and how the ports are connected. This diagram is used by Modelio to generate multi-model configurations.
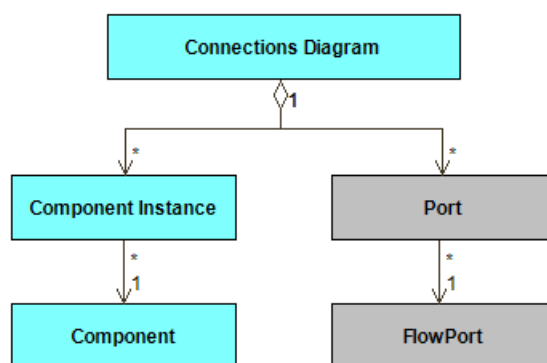
Figure 10: *Connections Diagram* describing the static structure of FMUs

20 SysML Diagrams Describing Design Space Exploration

The design space exploration (DSE) SysML profile is an addition to the multi-modelling SysML profile described above. As with single co-simulation, the INTO-CPS Application can run a DSE based on a JSON configuration file. These can be created manually in a text editor or edited in the INTO-CPS Application. Alternatively, a configuration can be generated by Modelio, from a set of diagrams defined in the profile. There are five diagram types, which are described below. Further guidance on DSE can be found in Chapter 29.

20.1 Objective Definition Diagram

The *Objective Definition Diagram* is used to define the objectives for use during a DSE. Objectives are characterising measures of performance that may be used to determine the relative benefits of competing designs. They are defined as metrics over the results of a co-simulation of a specific design and are used to judge its quality for use in later processing e.g. ranking.

Objectives are described in terms of a name, a script file that will be used to compute them, and the ports that will provide the data they require. As with the *Architectural Structure Diagram*
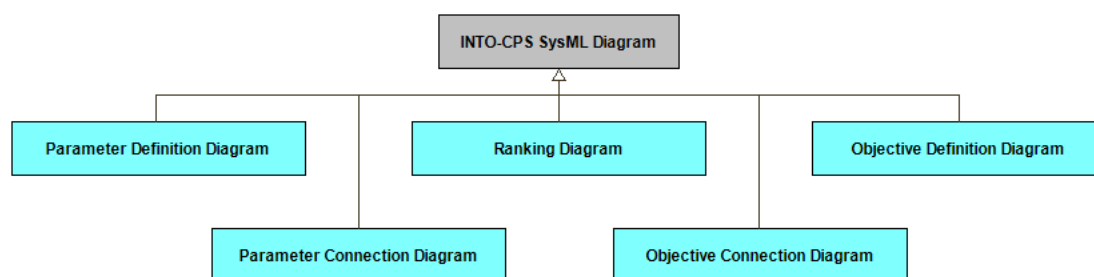
Figure 11: Diagrams in the DSE SysML profile

above (Section 19.1), this diagram gives the static structure of the objectives; instances of these definitions are created using the *Objective Connection Diagram* below (Section 20.2).
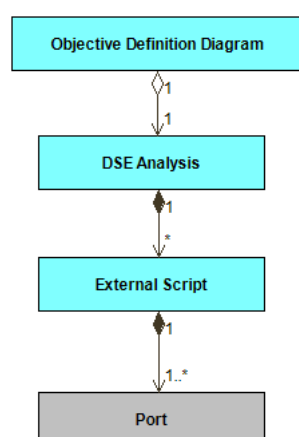
Figure 12: *Objective Definition Diagram* describing objectives in a DSE

20.2   Objective Connection Diagram

The *Objective Connection Diagram* is used to instantiate objectives defined in the *Objective Definition Diagram* above (Section 20.1). The diagrams allow the ports of each instance of the objective to be linked to a data source: either a static value, or a value from data exchanged in the multi-model.
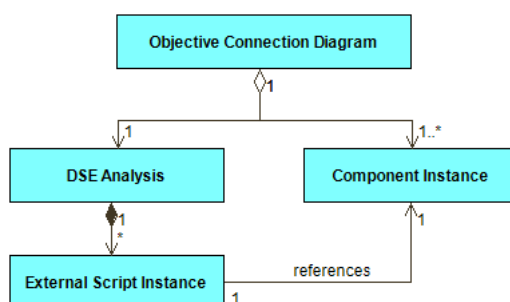
Figure 13: *Objective Connections Diagram* linking objectives to data sources

20.3　Parameter Definition Diagram

The *Parameter Definition Diagram* is used to define the parameters that will changed for each co-simulation in a DSE. Parameters are described in terms of a name, and a set of values that we wish to test. The product of the cardinalities of the set of values for each parameter gives the size of the design space— the total number of simulation required for an exhaustive search. As with the *Architectural Structure Diagram* above (Section 19.1), this diagram gives the static structure of the parameters; instances of these definitions are created using the *Parameter Connection Diagram* below (Section 20.4).
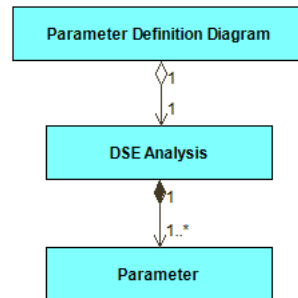
Figure 14: *Parameter Definition Diagram* defining parameters and their values

20.4　Parameter Connection Diagram

The *Parameter Connection Diagram* is used to instantiate parameters defined in the *Parameter Definition Diagram* above (Section 20.3). The diagram allows the parameters to be linked to those provided by the FMUs in the multi-model.
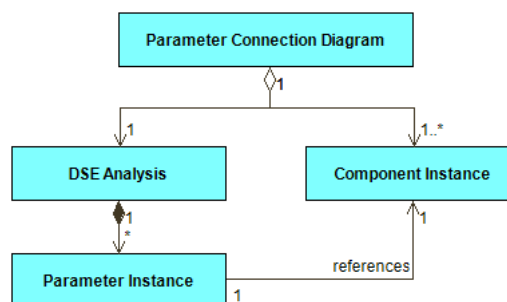
Figure 15: *Parameter Connections Diagram* linking parameters to FMUs

20.5　Ranking Diagram

The *Ranking Diagram* is used to declare which of the objectives should be used to compare competing designs, and whether lower or higher values for each the objectives is better (i.e. whether to maximise or minimise a value).
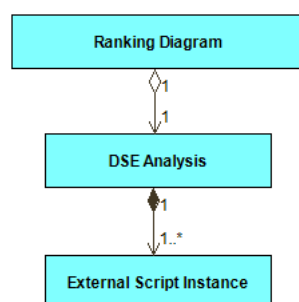
Figure 16: *Ranking Diagram* defining how to rank designs based on objectives

21   Holistic and Design Architectural Modelling

A system architecture defines the major components of a system, and identifies their relationships, behaviour and interactions. A model of the architecture is potentially partial (representing some or all of the system) and abstract, limited to those elements pertinent to the modelling goal. In CPS engineering, this goal may include understanding the system in terms of the application domain (a *holistic* model), or capturing the system components in a way that targets multi-modelling (a *design* model).

The diagrams in the two profiles described above divide architectural models into subsystems composed of cyber or physical components. Defining an architecture this way may not be the best approach when designing a system ab initio, with systems comprising entities across different domains requiring diverse domain expertise. Following on from Chapter 14, this section uses a smart grid example to show both holistic and design architectural modelling approaches, and provide some commentary and guidance on how to model in a way which is natural for domain experts, and how to move from holistic to design models when multi-modelling.

Example Introduction

A smart grid is an electricity power grid where integrated ICT systems play a role in the control and management of the electricity power supply. Such ICT elements include distributed control in households, control of renewable energies and networked communications. In this section we outline a Smart Grid model to explore different design decisions in the cyber control of an electricity power grid. The model presented here is a small illustrative example, which omits complexities of a real Smart Grid. For example, the change from three-phase AC power to one-phase DC power allowing us to use simpler physical models. A second simplification is in the number of houses present in the grid model. We model only 5 houses, assumed to be in a small local area supplied by a single substation. We do not consider the remainder of the grid. To ensure that any effect due to changes in the power consumption by those properties are observed by the other houses, we skew the resistance of the transmission lines between the power generation and substation, and substation to houses.

Holistic Architectural Model

A Block Definition Diagram (BDD) of the Smart Grid is given in Figure 17. The figure shows that the *Smart Grid* system comprises two top-level physical elements: *Power Gen-*
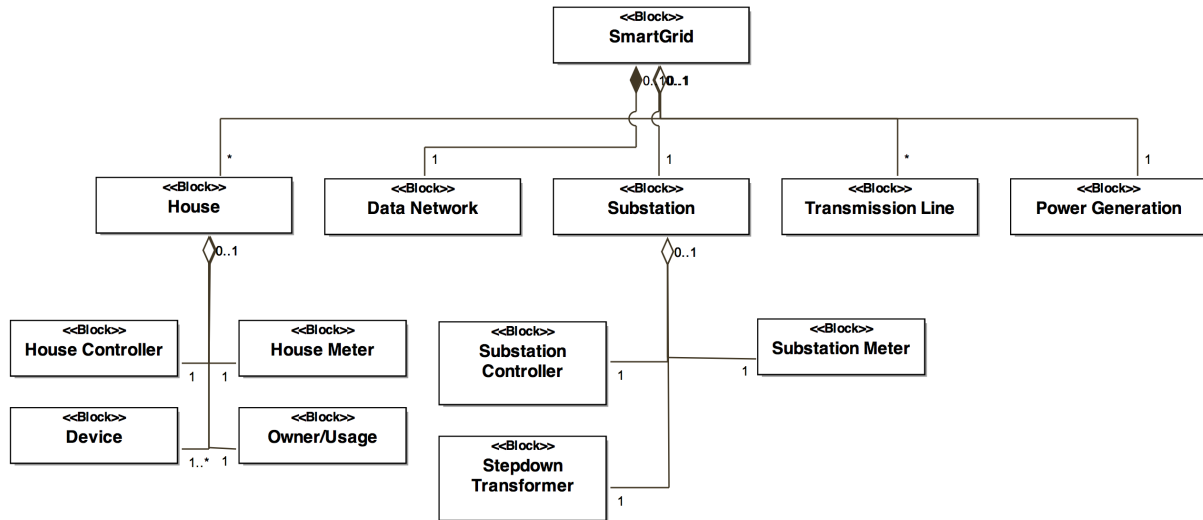
Figure 17: Block Definition Diagram of Smart Grid

eration and *Transmission Lines*; a single top-level cyber elements: the *Data Network*; and two cyber-physical systems: a *Substation* and several *Houses*. The two elements may be further decomposed. The *Substation* elements is composed of a cyber *Substation Controller* and physical *Substation Meter* and *Step-down Transformer*. The *House* element comprises: a cyber *House Controller*, physical *House Meter* and *Devices*, and an *Owner/Usage Profile*.

An Internal Block Diagram (IBD) of the Smart Grid is given in Figure 18. The diagram shows there are two main connection types in the model, corresponding to the physical power connections and the cyber data connections. The model also shows the connections between the cyber and physical parts of the models – currently modelled using data-type connections.

The first type of connection —the physical power connections— show a flow of *Power* from the Power Generation, through the Transmission Lines to the Houses, via the Substation. In the Substation, the Stepdown Transformer is connected to the Substation Meter. Similarly, in each House (only one is shown in the figure), the Power flows through the House Meter to each Device (again only one is shown for readability). The data connections exist between the Substation Controller and House Controllers. The Data Network is explicitly modelled and links the various controllers. Finally, there are links between the cyber controllers and the physical systems. In this model, the Substation Controller is connected to the Substation Meter, and the House Controller is linked to the House Meter and Devices.

Design Architectural Model

Looking at the holistic architecture defined in Figures 17 and 18 and moving towards a multi-model, we use the INTO-CPS SysML profile to define the architecture of the Smart Grid from the perspective of multi-model. This yields the ASD in Figure 19. This structure removes all subsystem structures such that each component is to be realised in a single FMU. Each element is defined as either a *physical* or *cyber* component, with the model type and platform identified.

The connections between the components are defined in the Connections Diagram (CD) in Figure 20. The interface between subsystems is defined as the interaction points between cyber and physical components (FMUs).
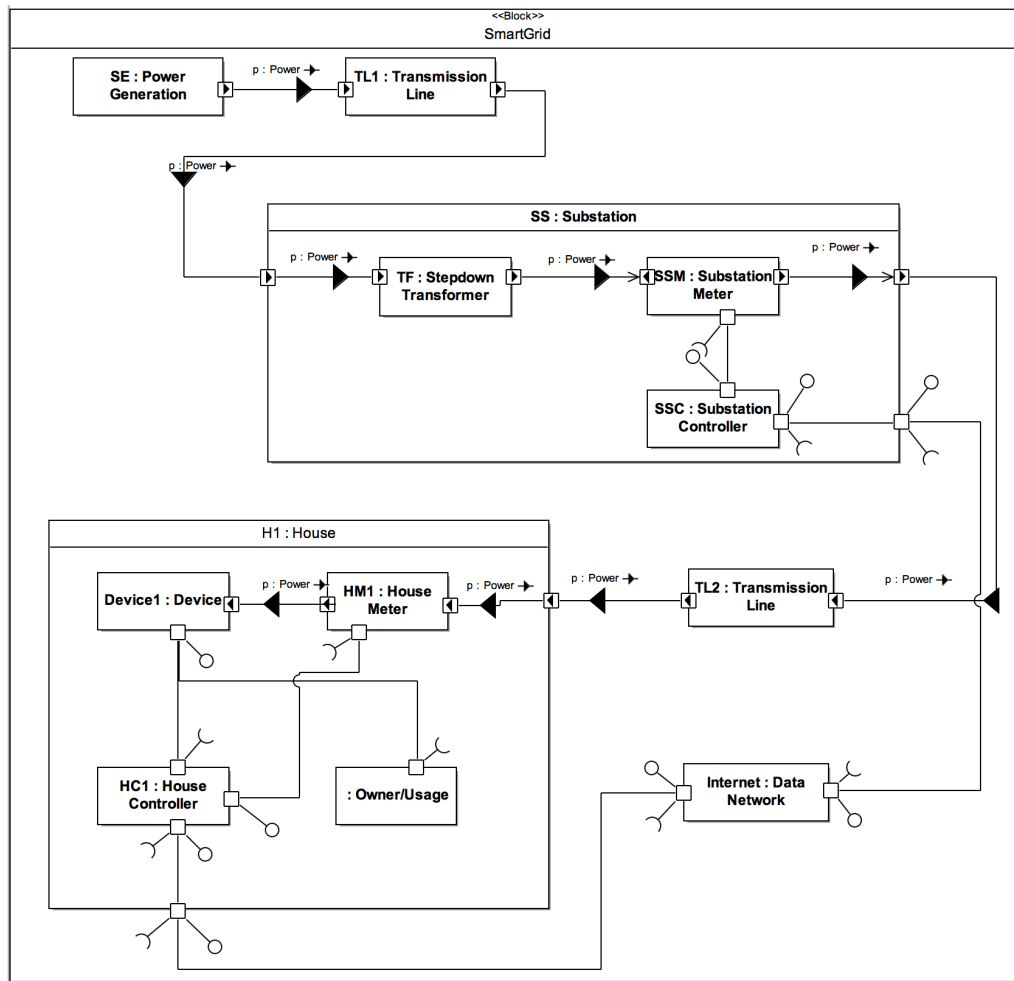
Figure 18: Internal Block Diagram of Smart Grid
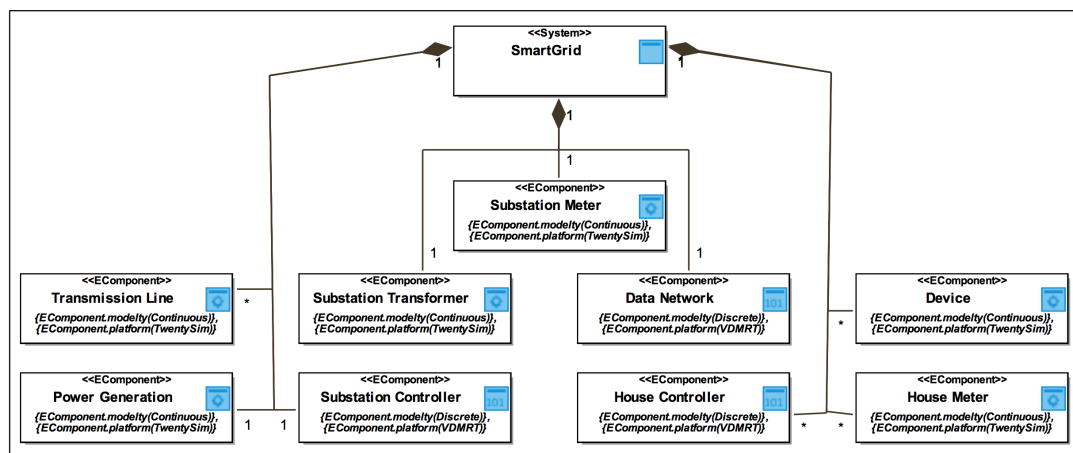
Figure 19: Architecture Structure Diagram for multi-model of Smart Grid
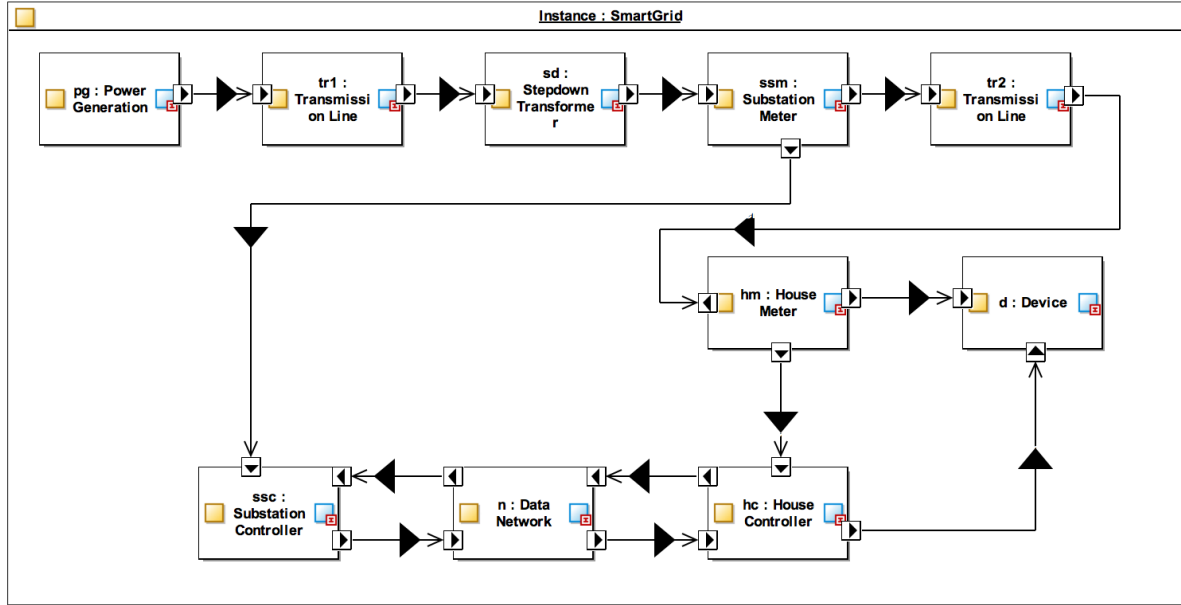
Figure 20: Connections Diagram for multi-model of Smart Grid

Discussion

Contrasting the architectures shown in the initial model (Figures 17 and 18) to that in the multi-model (Figures 19 and 20), whilst the same base components are present in both, some of the intuitive domain-specific structures are lost when moving to a multi-model. For example, it is now not clear where the *substation* or *house* elements are in the multi-model.

An important issue here is in the reason behind producing different architectural models. Using SysML diagrams in a *holistic* approach, a CPS engineer describes the model using a structure natural to the application domain. As such, the *reason* for modelling is not in the ultimate analysis to perform, but to define and understand the structure and behaviour of a system. In contrast, the *design* approach is necessary to configure INTO-CPS multi-models from SysML.

Figure 21 presents an overview of the relationships between the different types of models. The figure shows that the 'real' system may be modelled in different forms: the holistic and design architectures and the multi-model.

As illustrated in the figure, one approach can inform another. In some cases this may be a natural process; for example in the Smart Grid example, isolating each of the lowest level components in Figure 17 to be individual FMUs in a multi-model is an evolution which will likely result in a feasible model. By creating a domain-specific holistic architecture first, then transforming these models into a design architecture for multi-modelling, design teams will likely gain the most benefit.

22   Representing Non-Design Elements in SysML

Using the INTO-CPS tool chain, we generate co-simulation configurations using an architectural model defined with the INTO-SysML profile. This model defines the structure of a system in terms of the composition of its components and their connections. There are however circumstances where elements in the multi-model are not part of the design of the final system,
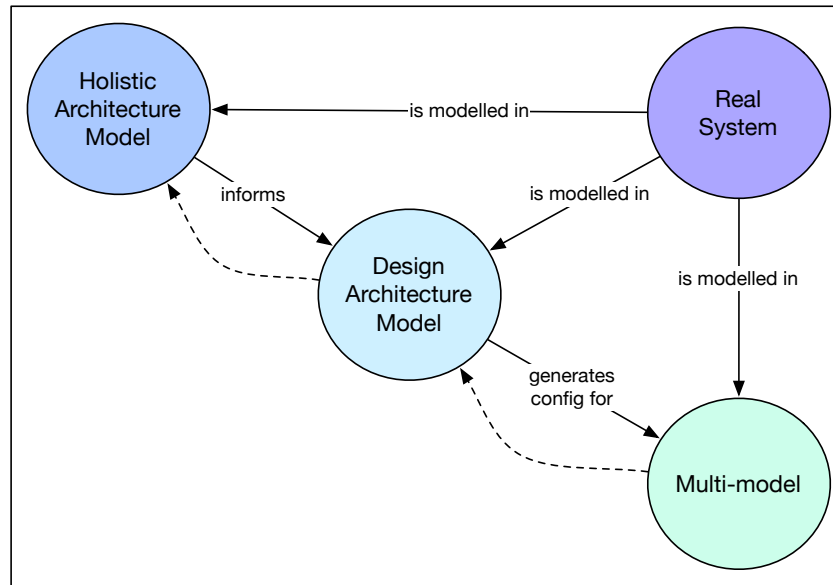
Figure 21: Relating holistic and design architectures
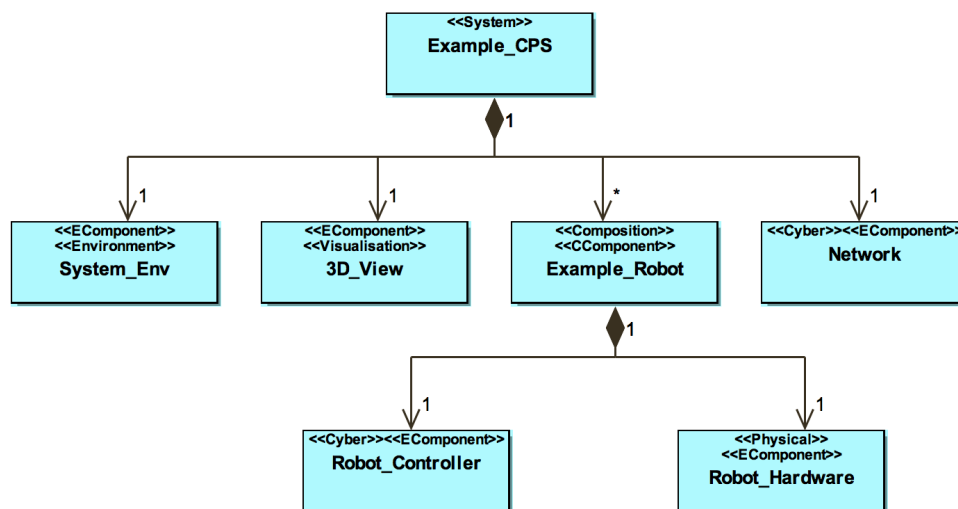
Figure 22: Example Architecture Structure Diagram of robot system

for example where an FMU is used purely for visualisation. This FMU must be connected to the system components, however is not itself a system component. This is also true when considering the environment of the system.

Here we present a small example of the use of these extensions, using a simple robot example (based on the line-following robot pilot study, see Deliverable D3.6 [MGP+17]) to illustrate the use of «CComponent»s and the *kind* of components (*cyber*, *physical*, *environment*, *visualisation*) described in Section 19.1 above.

The architecture structure diagram in Figure 22 shows: a *System_Env* block, an «EComponent» defined as an Environment FMU; a *3D_View* block an «EComponent», defined as an Visualisation FMU; and an *Example_Robot* block, an «EComponent» defined as an composition of two FMUs.

The example has two connection diagrams. The first is shown in Figure 23, it contains only
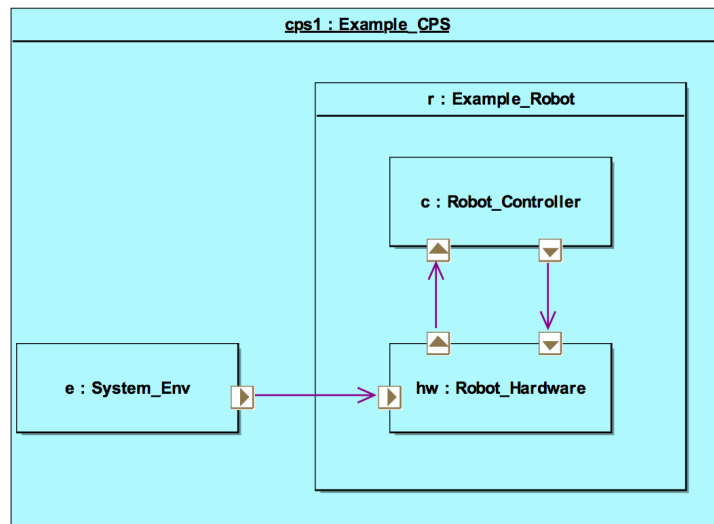
Figure 23: Connections Diagram for robot showing only system and environment connectors
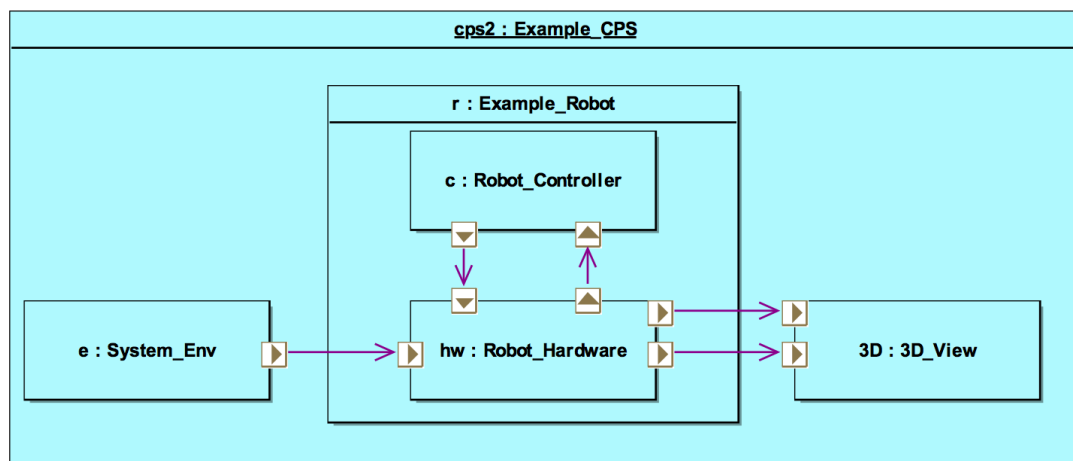
Figure 24: Connections Diagram for the robot system showing the system *and* visualisation components

those connections with respect to the system and its constituent components . This diagram shows a block instance *cps1* containing the environment (*e*) and the example robot (*r*) which contains two the controller and hardware components.

The second is shown in Figure 24, it depicts the use of the block instance *3D* of type *3D_View*. In this diagram, we show additional ports of the original block instances to output internal model details and connect these to the *3D* instance. The diagram includes the system connectors as shown in Figure 23.

Initial Multi-Modelling using VDM

In this section we provide guidance on producing initial multi-models from architectural descriptions produced using the INTO-CPS SysML profile. We focus on using discrete-event (DE) models to produce initial, abstract FMUs that allow integration testing through co-simulation before detailed modelling work is complete. This is called a "DE-first" approach [FLPV13, FLV13]. We describe the use of VDM and the Overture tool, with FMI export plug-in installed, for this approach. The principles outlined in this section can be applied in other modelling tools. This approach can work with or without the SysML profile.

23 The DE-first Approach

After carrying out requirements engineering (RE), as described in Chapter 14, and design architectural modelling in SysML, as described in Chapter 18, the engineering team should have the following artifacts available:

- One or more Architecture Structure Diagrams (ASDs) defining the composition of «EComponent»s (to be realised as «Cyber» or «Physical» FMUs) that will form the multi-model.
- Model descriptions exported for each «EComponent».
- One or more Connections Diagrams (CDs) that will be used to configure a multi-model.

The next step is to generate a multi-model configuration in the INTO-CPS Application and populate it with FMUs, then run a first co-simulation. This however requires the source models for each FMU to be ready. If they already exist this is easy, however they may not exist if this is a new design. In order to generate these models, the model descriptions for each «EComponent» can be passed to relevant engineering teams to build the models, then FMUs can be passed back to be integrated.

It can be useful however to create and test simple, abstract FMUs first (or in parallel), then replace these with higher-fidelity FMUs as the models become available. This allows the composition of the multi-model to be checked early, and these simple FMUs can be reused for regression testing. This approach also mitigates the problem of modelling teams working at different rates.

Where these simple FMUs are built within the DE formalism (such as VDM), this is called a *DE-first* approach. This approach is particularly appropriate where complex DE control behaviours —such as supervisory control or modal behaviours— are identified as a priority or where the experience of the modelling team is primarily in the DE domain [FLV14].

Guidance on how to produce DE approximations for use in multi-modelling, and in particular approximations of CT behaviour, can be found in material describing the Crescendo baseline technology [FLV13], which is also available via the Crescendo website[14].

24 DE-first within INTO-CPS

Given an architectural structure diagram, connections diagram and model descriptions for each «EComponent», the suggested approach is to begin by building a single VDM-RT project in Overture with the following elements:

---

[14]See http://crescendotool.org/documentation/
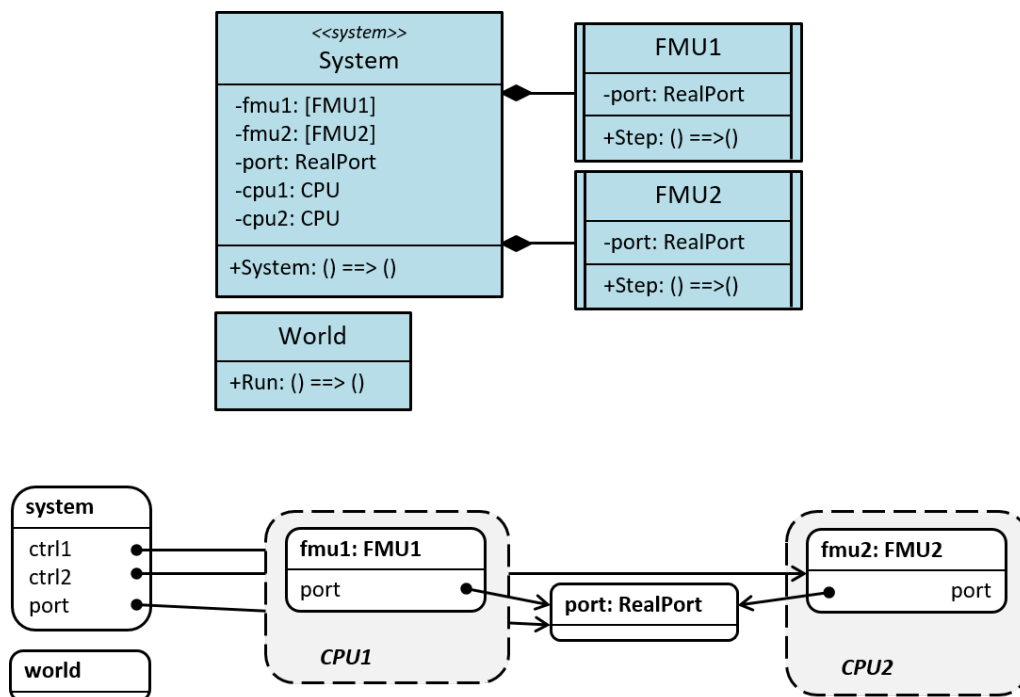
Figure 25: Class diagram showing two simplified FMU classes created within a single VDM-RT project, and an object diagram showing them being instantiated as a test.
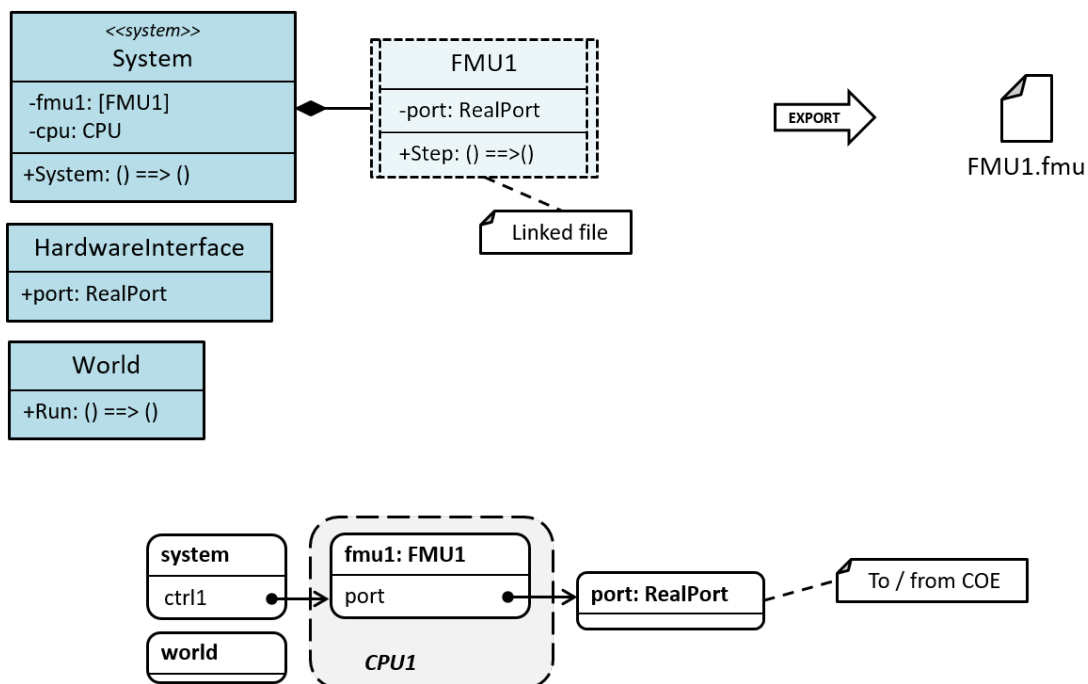
Figure 26: Class and object diagrams showing a linked class within its own project for FMU creation.

- A class for each «EComponent» representing an FMU. Each class should define port-type instance variables (i.e. of type IntPort, RealPort, BoolPort, or StringPort) corresponding to the model description and a constructor to take these ports as parameters. Each FMU class should also define a thread that calls a Step operation, which should implement some basic, abstract behaviour for the FMU.
- A system class that instantiates port and FMU objects based on the connections diagram. Ports should be passed to constructor of each FMU object. Each FMU object should be deployed on its own CPU.
- A World class that starts the thread of each FMU objects.

Class and object diagrams giving an example of the above is shown in Figure 25. In this example, there are two «EComponent»s (called *FMU1* and *FMU2*) joined by a single connection of type real. Such a model can be simulated within Overture to test the behaviour of the FMUs. This approach can be combined with the guidance in Chapter 25 to analyse more complicated networked behaviour. Once the behaviour of the FMU classes has been tested, actual FMUs can be produced and integrated into a first multi-model by following the guidance below.

25   FMU Creation

The steps outlined below assume a knowledge of FMU export in Overture, which can be found in the User Manual, Deliverable D4.3a [BLL$^+$17], in Section 5.1. To generate FMUs, a project must be created for each «EComponent» with:

- One of the FMU classes from the main project.
- A HardwareInterface class that defines the ports and annotations required by the Overture FMU export plug-in, reflecting those defined in the model description.
- A system class that instantiates the FMU class and passes the port objects from the HardwareInterface class to its constructor.
- A World class that starts the thread of the FMU class.

The above structure is shown in Figure 26. A skeleton project with a correctly annotated HardwareInterface class can be generated using the model description import feature of the Overture FMU plug-in. The FMU classes can be linked into the projects (rather than hard copies being made) from the main project, so that any changes made are reflected in both the main project and the individual FMU projects. These links can be created by using the *Advanced* section of the *New > Empty VDM-RT File* dialogue, using the PROJECT-1-PARENT_LOC variable to refer to the workspace directory on the file system (as shown in Figure 27). Note that if the FMU classes need to share type definitions, these can be created in a class called Types in the main project, then this class can be linked into each of the FMU projects in the same way.

From these individual project, FMUs can be exported and co-simulated within the INTO-CPS tool. These FMUs can then be replaced as higher-fidelity versions become available, however they can be retained and used for regression and integration testing by using different multi-model configurations for each combination.
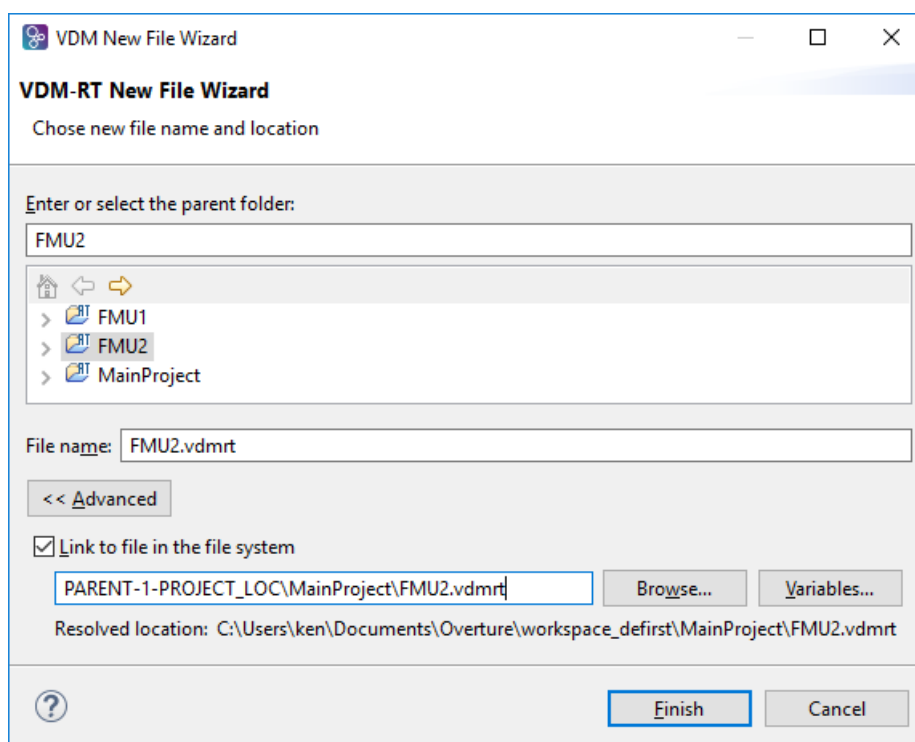
Figure 27: Linking files in the *New > Empty VDM-RT File* dialogue.
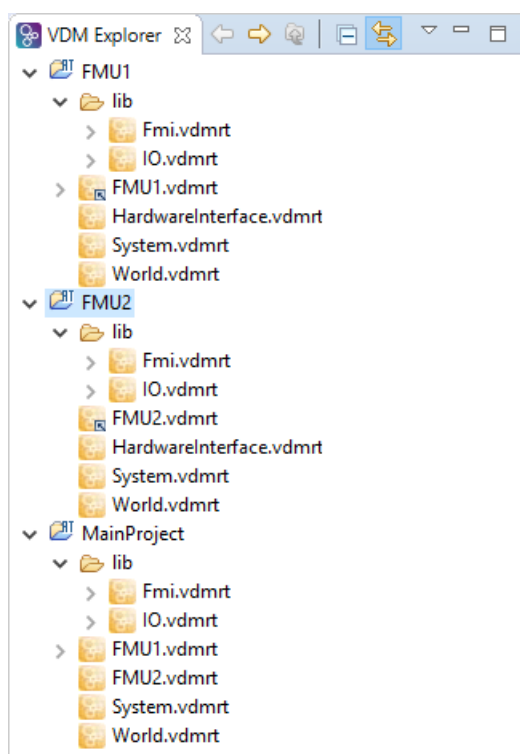
Figure 28: Project structure of an Overture workspace showing a main project and two projects used for generating FMUs from linked class files.

Modelling Networks with VDM in Multi-models

In this section, we address the problem of modelling networked controllers in multi-models, presenting a solution using VDM. When modelling and designing distributed controllers, it is necessary to model communications between controllers as well. While controller FMUs can be connected directly to each other through for co-simulation, this quickly becomes unwieldy due to the number of connections increasing exponentially. For example, consider the case of five controllers depicted in Figure 29. In order to connect each controller together, 20 connections are needed (i.e. for a complete bidirected graph). Even with automatic generation of multi-model configurations, this is in general not a feasible solution.
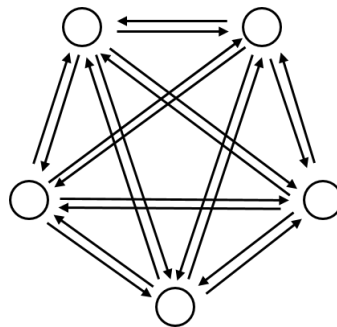
Figure 29: Topology of five controllers connected to each other

We suggest employing a pattern described initially as part of the Crescendo technology [FLV14], in which a representation of an abstract communications medium called the 'ether' is introduced. In the INTO-CPS setting, the ether is an FMU that is connected to each controller that handles message-passing between them. This reduces the number of connections needed, particularly for large numbers of controllers such as swarms. For five controllers, only 10 connections are needed, as shown in Figure 30.
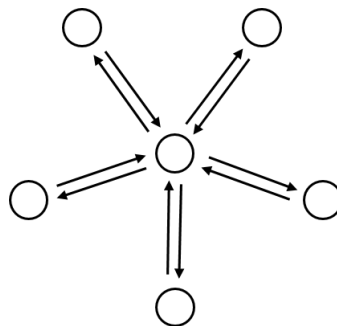
Figure 30: Topology of five controllers connected via a shared medium

In the remainder of this section, we describe how to pass messages between VDM FMUs using string types, how the ether class works, some of the consequences of using the ether pattern, and finally some extensions for providing quality of service (QoS) guarantees. An example multi-model, called *Case Study: Ether*, is available from the INTO-CPS Application. It is also described in the Examples Compendium, Deliverable D3.6 [MGP+17].

26　Representing VDM Values as Strings

Connections between FMUs are typically numerical or Boolean types. This works well for modelling of discrete-time (DT) controllers and continuous-time (CT) physical models, however one of the advantages of VDM is the ability to represent more complex data types that better fit the abstractions of supervisory control. Therefore, in a multi-modelling setting, it is advantageous if VDM controllers can communicate with each other using data types that are not part of the FMI specification.

This can be achieved by passing strings between VDM FMUs (which are now supported by the Overture FMU export plug-in) and the `VDMUtil` standard library included in Overture, which can convert VDM types to their string representations and back again.

The `VDMUtil` library provides a (polymorphic) function called `val2seq_of_char`, that converts a VDM type to a string. It is necessary to tell the function what type to expect as a parameter in square brackets. For example, in the following listing, a 2-tuple is passed to the function, which will produce the output `"mk_(2.4, 1.5)"`:

```
VDMUtil`val2seq_of_char[real*real](mk_(2.4, 1.5))
```

The above can be used when sending messages as strings. In the model receiving message, the inverse function `seq_of_char2val` can be used. This function returns two values, a Boolean value indicating if the conversion was successful, and the value that was received:

```
let mk_(b,v) = VDMUtil`val2seq_of_char[real*real](msg) in
  if b then ...
```

In the first few steps of co-simulation, empty or invalid strings are often passed as values, so it is necessary to check if the conversion was successful (as in the above listing) before using the value.

Note that currently (as of Overture 2.4.0), the `VDMUtil` library is called in the default scope, meaning that it does not know about custom types defined in the model. Therefore, it is recommended to pack values in a tuple (as in the above example) for message passing, then convert to and from any custom types in the sending and receiving models.

27　Using the Ether FMU

By encoding VDM values as strings, it is possible to define a simple broadcast ether that receives message strings on its input channel(s) and sends them to its output channel(s). As a concrete example, we consider the *Case Study: Ether* (see Deliverable D3.5 [PGP+16]), which contains a `Sender`, a `Receiver` and an `Ether`, as depicted in Figure 31. In this example, the three FMUs have the following roles:

Sender Generates random 3-tuple messages of type `real * real * real`, encodes them as strings using the `VDMUtil` library and puts them on its output port.

Receiver Receives strings on its input port and tries to convert them to single messages of type `real * real * real` or to a sequence of messages of type of type `seq of (real * real * real)`.

Ether Has an input port and output port, each assigned a unique identifier, i.e. as a `map Id to StringPort`. It also has a mapping of input to output ports as a set of pairs: `set`
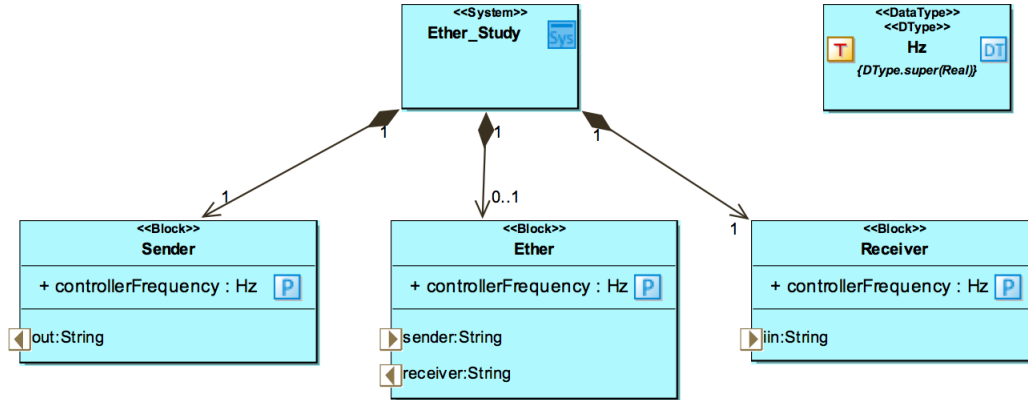
Figure 31: *Case Study: Ether* example

of (Id * Id). It has a list that holds messages for each output destination, because multiple messages might arrive for one destination. It gathers messages from each input and passes them to the outputs defined in the above mapping.

In this simple example, the sender and receiver are asymmetrical, but in more complicated examples controllers can be both senders and receivers by implementing both of the behaviours described above.

The *Case Study: Ether* example contains two multi-models that allow the sender and receiver to be connected directly (connection diagram shown in Figure 32(a)), or to be connected via the ether (connection diagram shown in Figure 32(b)). The description in the Examples Compendium, Deliverable D3.5 [PGP+16], explains how to run the two different multi-models. This approach shows that the use of string ports and the VDMUtil library can be useful even without the ether for message passing between controllers in simple topologies.

For the sender, this connection is transparent, it does not care whether it is connected to the ether or not. For the receiver, in the direct connection it will receive single messages, whereas when receiving from the ether it will receive a list of messages (even for a single value). So the receiver is able to deduce when it is directly connected or connected via the ether.

The ether defined in this example is intended to be generic enough that it can be used in other case studies that need a simple broadcast ether without guarantees of delivery. To use it, you can:

1. Import the *Ether* model from the *case-study_ether/Models* directory into Overture;
2. Update the HardwareInterface[15] class to provide input and/or output ports for all controllers that will be connected to the ether.
3. Update the System class to assign identifiers to all input and output ports; and
4. Update the set of identifier pairs that define connections.

28   Consequences of Using the Ether

The ether as presented above is fairly basic. In each update cycle, it passes values from its input variables to their respective output variables. This essentially replicates the shared variable

---

[15]A class that provides annotated definitions of the ports for a VDM FMU.
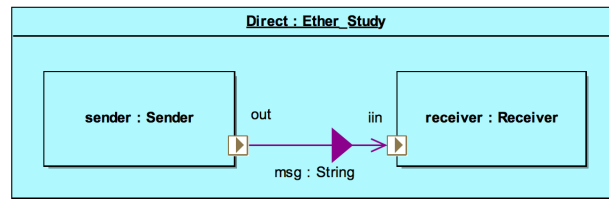
(a) Connection diagram of the *Direct* multi-model in the *Case Study: Ether* example
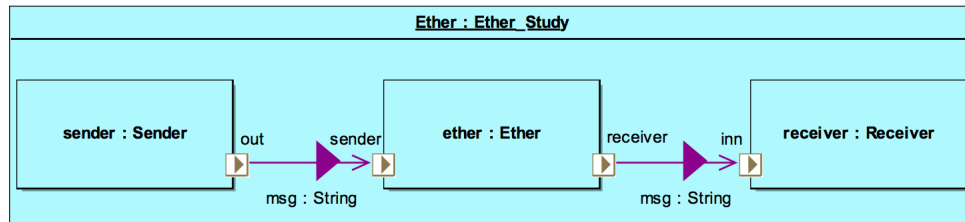
(b) Connection diagram of the *Ether* multi-model in the *Case Study: Ether* example

Figure 32: Alternative multi-models in the *Case Study: Ether* example

style of direct FMU-FMU connections, which means that the relative update speeds of the FMUs may lead to the following:

Values may be delayed The introduction of an intermediate FMU means that an extra update cycle is required to pass values from sender to ether and ether to receiver. This may delay messages unless the ether updates at least twice as fast as the receiver.

Values may not be read If a value is not read by the receiver before it is changed, then that value is lost.

Values may be read more than once If a value is not changed by the sender before the receiver updates, then the value is read twice. In the simple ether, the receiver cannot distinguish an old message from a new message with the same values.

In the Examples Compendium, Deliverable D3.5 [PGP+16], the *Case Study: Ether* example is described along with some suggested experiments to see the effects of the above examples by changing the controller frequency parameters of the sender, ether and receiver. In the final part of this section we outline ways to overcome such problems if it is necessary to guarantee that messages arrive and are read during a co-simulation.

29 Modelling True Message Passing and Quality of Service

The key to achieving a true message-passing is to overcome the problem of distinguishing old messages from new messages with the same values. This can be done by attaching a unique identifier to each message, which could be, for example, an identifier of the sender plus a message number:
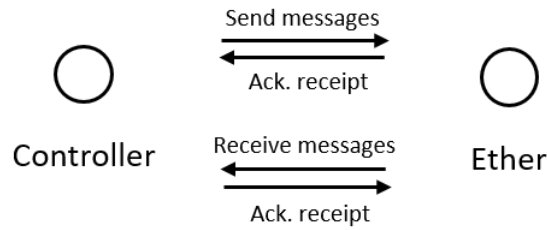
Figure 33: Topology of controller to *Ether* connection with dedicated channels for messages and acknowledgements

```
instance variables

id: seq of char := "a";
seqn: nat1 := 1;

...

VDMUtil`val2seq_of_char[seq of char*real*real](
  mk_(id ^ [seqn], 2.4, 1.5));
seqn := seqn + 1
```

The advantage of assigning an identifier to each controller is that messages could also contain destination addresses, instead of the broadcast model presented above. In order to achieve these, some changes are needed to allow for acknowledging receipt of messages. Controllers should:

1. Send a queue of messages on their output channel along with message identifiers of (recently received) messages;
2. Expect to receive a queue of messages along with message identifiers of successfully sent messages; and
3. Senders should remove messages from their output queue once their receipt has been acknowledged.

The `Ether` class must be extended to:

1. Inspect the message identifier (and destination if required) using `VDMUtil`;
2. Pass message identifiers back to senders to acknowledge receipts; and
3. Listen for message identifiers from receivers to know when to remove messages from the queue.

A dedicated channel for acknowledging messages could also be introduced, which would simplify the above. Therefore, each controller would have four connections to the ether: send and acknowledge, receive and acknowledge, as depicted in Figure 33.

The advantages of guaranteed message delivery as described here are that realistic and faulty behaviour of the communication medium can be studied. An ether can be produced that provides poorer quality of service (delay, loss, repetition, reordering). These behaviours could be parameterised and explored using DSE (see Chapter 29). By controlling for problems introduced by the nature of co-simulation, any reduction in performance of the multi-model can be attributed to the realistic behaviour introduced intentionally into the model of communications.

Design Space Exploration

In this section, we outline guidelines for DSE over co-models of CPSs that: (a) support decision management by helping engineers to articulate clearly the parameters, objectives and metrics of a DSE analysis (Section 30); and (b) enable the tuning of DSE methods for given domains and systems of interest (Section 31).

30   Guidelines for Designing DSE in SysML

30.1   Rationale

Designing DSE experiments can be complex and tied closely to the multi-model being analysed. The definitions guiding the DSE scripts should not just appear with no meaningful links to the any other artefacts in the INTO-CPS Tool chain. There are two main reasons for this, firstly there is no traceability back to the requirements from which we might understand why the various objectives (measures) were being evaluated or why they were included in the ranking definition. Secondly, if DSE configurations are created manually for each new DSE experiment it is easy to imagine that the DSE analysis and ranking might not be consistent among the experiments.

Engineers need, therefore, to be able to model at an early stage of design how the experiments relate to the model architecture, and where possible trace from requirements to the analysis experiments. Here we describe the first step towards this vision: a SysML profile for modelling DSE experiments. The profile comprises five diagrams for defining *parameters*, *objectives* and *rankings*.

We take the same approach to defining the SysML profile for DSE as that used to define the INTO-SysML profile. A metamodel is defined (see Deliverable D3.2b [FGPP16b]) and the collection of profile diagrams that implement this metamodel are defined in Deliverable D4.2c [BQ16].

In this section, we present an illustrative example of the use of the DSE-SysML profile – from requirements engineering through defining parameters and objectives in the DSE-SysML profile to the final DSE JSON configuration files. We present result of the execution of DSE for the defined configuration.

As an example, we use the line follower robot pilot study. More details can be found in Deliverable D3.5 [PGP$^+$16].

30.2   Requirements

We propose the use of a subset of the SoS-ACRE method detailed in Chapter 14 (as this section concentrates on the application of the DSE-SysML profile, we don't consider the full SoS-ACRE process). In the Requirements Definition View in Figure 34, the following five requirements are defined:

1. The robot shall have a minimal cross track error

2. The cross track error shall never exceed **X** mm

3. The robot shall maximise its average speed

4. The robot shall have a minimum average speed of $\mathbf{X}$ ms^{-1}

5. The robot sensor positions may be altered to achieve global goals

RDV Robot Requirements

<<requirement>>
Line Follower DSE Requirements

<<requirement>>
Minimise Cross Track Error

id = "R1"
description = "The robot shall have a minimal cross track error"

<<requirement>>
Maximise Average Speed

id = "R3"
description = "The robot shall maximise its average speed"

<<requirement>>
Alterable Sensors

id = "R5"
description = "The robot sensor positions may be altered to achieve global goals"

<<requirement>>
Cross track not exceeded

id = "R2"
description = "The cross track error shall never exceed X mm"

<<requirement>>
Minimum Average speed

id = "R4"
description = "The robot shall have a minimum average speed of X ms^{-2} "

Figure 34: Subset of the Requirements Definition View for requirements of the Line Following Robot

30.3   Objectives from Requirements

Based upon the requirements above, we define two objectives: the calculation of deviation from a desired path, and the speed of the robot.

Deviation   The deviation from a desired path, referred to as the cross track error, is the distance the robot moves from the line of the map, as shown in Figure 35.

Desired path   Cross Track Error
Actual Path
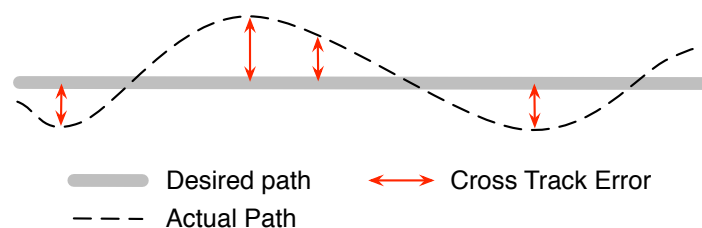
Figure 35: Cross track error at various points for a robot trying to follow a desired line

To compute cross track error we need some model of the desired path to be followed and the actual path taken by the robot. Each point on the actual path is compared with the model of the desired path to find its distance from the closest point, this becomes the cross track error. If the desired path is modelled as a series of points, then it may be necessary to find shortest distance to the line between the two closest points.

Speed   The speed may be measured in several ways depending on what data is logged by the COE and what we really mean by speed, indicated in Figure 36.
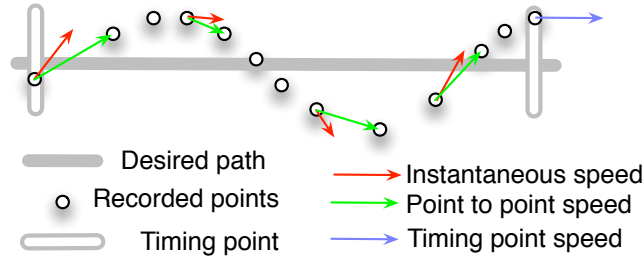
Figure 36: Cross track error at various points for a robot trying to follow a desired line

Inside the CT model there is a bond graph flow variable that represents the forwards motion of the robot. This variable is not currently logged by the COE but it could be and this would result in snapshots of the robot speed being taken when simulation models synchronise. In this example, we take the view that speed is referring to the time taken to complete a lap.

30.4   SysML Representation of Parameters, Objectives and Ranking

We next consider the use of the upcoming DSE profile to define the DSE parameters, objectives and desired ranking function. In the following SysML diagrams, we explicitly refer to model elements as defined in the architectural model of the line follower study, presented in Deliverable D3.5 [PGP+16].

Parameters   In the requirements defined above, we see that the position of the line follower sensors may be varied. In real requirements, we may elicit the possible variables allowed. Figure 37 is a DSE Parameter Definition Diagram and defines four parameters required: *S1_X*, *S1_Y*, *S2_X* and *S2_Y*, each a set of real numbers. The DSE experiment in this example is called *DSE_Example*.

Figure 38 identifies the architectural model elements themselves (the `lf_position_x` and `lf_position_y` parameters of *sensor1* and *sensor2*) and the possible values each may have (for example the `lf_position_x` parameter of *sensor1* may be either 0.01 or 0.03). The diagram (or collection of diagrams if there is a large number of design parameters) should record all parameters for the experiment.

Objectives   The objectives follow from the requirements as mentioned above. Figure 39 shows the DSE Objectives Definition Diagram with four objectives: *meanSpeed*, *lapTime*, *maxCrossTrackError* and *meanCrossTrackError*. Each have a collection of inputs – defined either as constants (e.g. parameter *p1* of *meanSpeed*), or to be obtained for the multi-model.

The objective definitions are realised in Figure 40. The *meanSpeed* requires the step-size of the simulation (this is obtained from the co-simulation results, rather than defined here) and the `robot_x` and `robot_y` position of the robot body. The *lapTime* objective requires the time at each simulation step (again, obtained directly from the co-simulation output), the `robot_x` and `robot_y` position of the robot body and the name of the map. Both the *maxCrossTrackError* and *meanCrossTrackError* objectives require only the `robot_x` and `robot_y` position of the robot body.
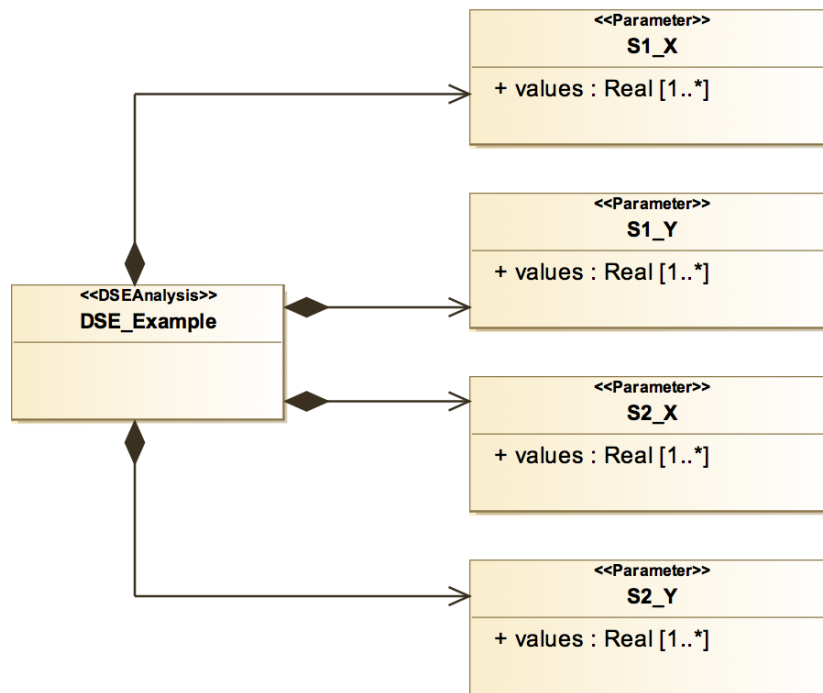
Figure 37: DSE-SysML Parameter Definition Diagram of Line Following Robot example
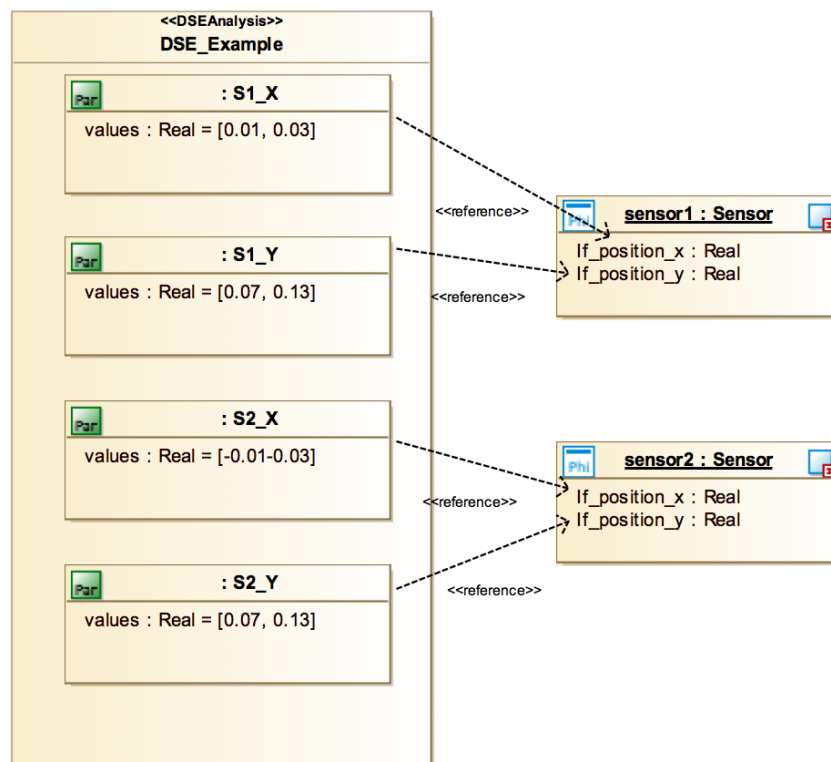
Figure 38: DSE-SysML Parameter Connection Diagram of Line Following Robot example
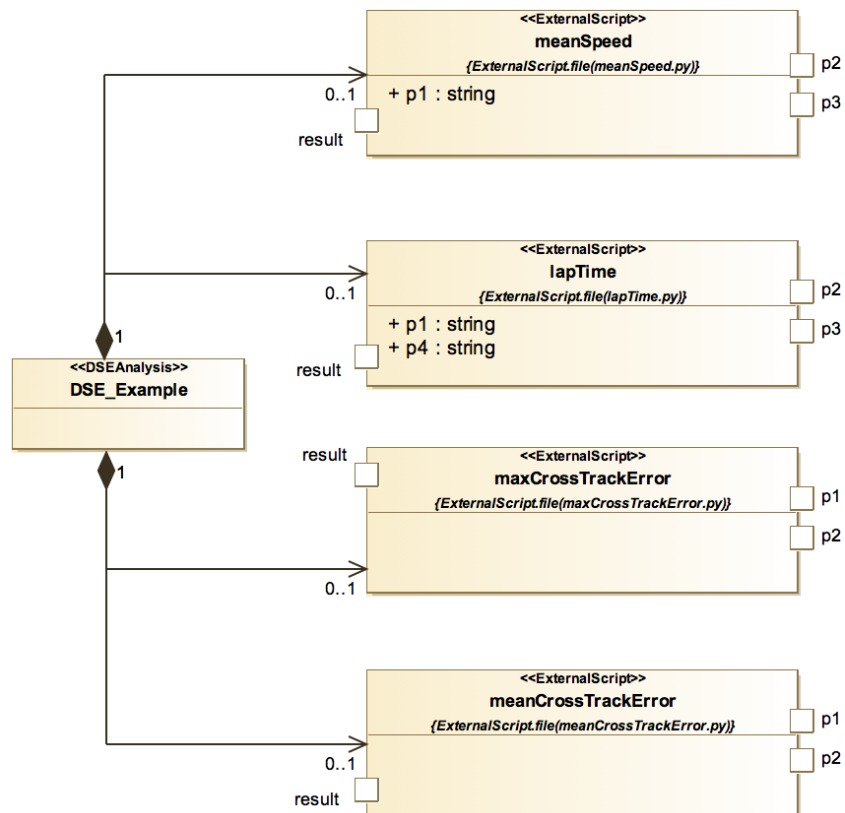
Figure 39: DSE-SysML Objective Definition Diagram of Line Following Robot example
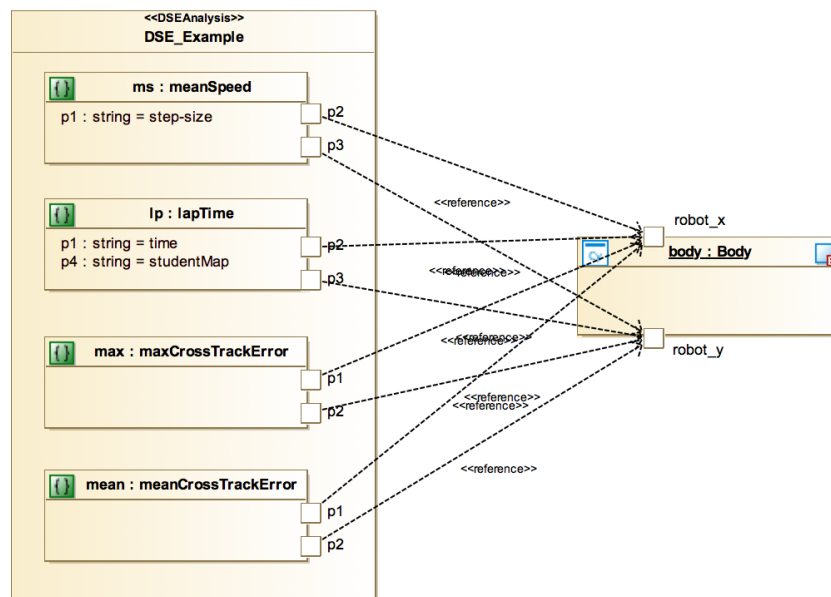
Figure 40: DSE-SysML Connection Objective Diagram of Line Following Robot example

Ranking   Finally, the DSE Ranking Diagram in Figure 41 defines the ranking to be used in the experiment. This diagram states that the experiment uses the Pareto method, and is a 2-value Pareto referring to the *lapTime* and *meanCrossTrackError* objectives.
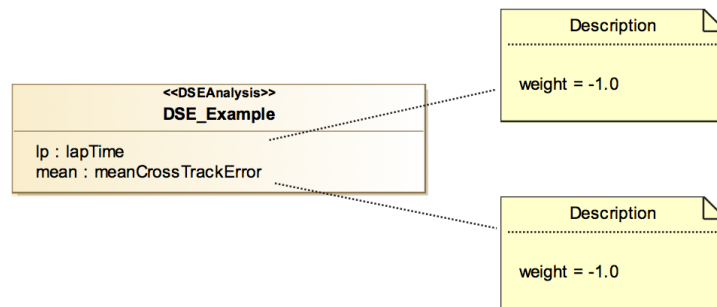
Figure 41: Example DSE-SysML Ranking Diagram of Line Following Robot example

30.5   DSE script

These diagrams may then be translated to the JSON config format required by the DSE tool. The export of the configuration is performed in the Modelio tool and the subsequent movement of the resulting configuration file is performed in the INTO-CPS application (see the INTO-CPS User Manual, Deliverable D4.3a [BLL+17] for more details). Figure 42 shows the corresponding DSE configuration file for the line follower experiments. Note that where we refer to model elements of the architecture (such as model parameters), we now use the same conventions used in the co-simulation orchestration engine configuration.

30.6   DSE results

DSE is performed in the DSE tool (again, see the INTO-CPS User Manual, Deliverable D4.3a [BLL+17] for more detail) by processing the DSE configuration using scripts that contain the required algorithms. The main scripts contain the search algorithm that determines which parameters to use in each simulation, the simplest of these is the exhaustive algorithm that methodically runs through all combinations of parameters and runs a simulation of each. The log files produced by each simulation are then processed by other scripts to obtain the objective values defined in the previous section. Finally, the objective values are used by a ranking script to place all the simulation results into a partial order according to the defined ranking. The ranking information is used to produce tabular and graphical results that may be used to support decisions regarding design choices and directions.

Figure 43 shows an example of the DSE results from the line follower robot where the lap time and mean cross track error were the objectives to optimise. These results contain two representations of the data, a graph plotting the objective values for each design, with the Pareto front of optimal trade-offs between the key objectives highlighted, here in blue. The second part of the results presents the data is tables, indexed by the ranking position of each result. This permits the user to determine the precise values for both the measured objectives and also the design parameters used to obtain that result.

```
{
    "algorithm": {},
    "objectiveConstraints": {},
    "objectiveDefinitions": {
        "externalScripts": {
            "meanSpeed": {
                "scriptFile": "meanSpeed.py",
                "scriptParameters": {
                    "1": "step-size",
                    "2": "{bodyFMU}.body.robot_x",
                    "3": "{bodyFMU}.body.robot_y"
                }
            },
            "lapTime": {
                "scriptFile": "lapTime.py",
                "scriptParameters": {
                    "1": "time",
                    "2": "{bodyFMU}.body.robot_x",
                    "3": "{bodyFMU}.body.robot_y",
                    "4": "studentMap"
                }
            },
            "maxCrossTrackError": {
                "scriptFile": "maxCrosstrackError.py",
                "scriptParameters": {
                    "1": "{bodyFMU}.body.robot_x",
                    "2": "{bodyFMU}.body.robot_y"
                }
            },
            "meanCrossTrackError": {
                "scriptFile": "meanCrosstrackError.py",
                "scriptParameters": {
                    "1": "{bodyFMU}.body.robot_x",
                    "2": "{bodyFMU}.body.robot_y"
                }
            }
        },
        "internalFunctions": {}
    },
    "parameters": {
        "{sensor1FMU}.sensor1.lf_position_x": [
            0.01,
            0.03
        ],
        "{sensor1FMU}.sensor1.lf_position_y": [
            0.07,
            0.13
        ],
        "{sensor2FMU}.sensor2.lf_position_x": [
            -0.01,
            -0.03
        ],
        "{sensor2FMU}.sensor2.lf_position_y": [
            0.07,
            0.13
        ]
    },
    "ranking": {
        "pareto": {
            "lapTime": "-",
            "meanCrossTrackError": "-"
        }
    },
    "scenarios": [
        "studentMap"
    ]
}
```

Figure 42: A complete DSE configuration JSON file for the line follower robot example

31 An Approach to Effective DSE

Given a "designed" design space using the method detailed above, we use the INTO-CPS Tool Chain to simulate each design alternative. The initial approach we took was to implement an algorithm to exhaustively search the design space, and evaluate and rank each design. Whilst this approach is acceptable on small-scale studies, this quickly becomes infeasible as the design space grows. For example, varying n parameters with m alternative values produces a design space of m^n alternatives. In the remainder of this paper, we present an alternative approach to exploring the design space in order to provide guidance for CPS engineers on how to design the exploration of designs for different classes of problems.
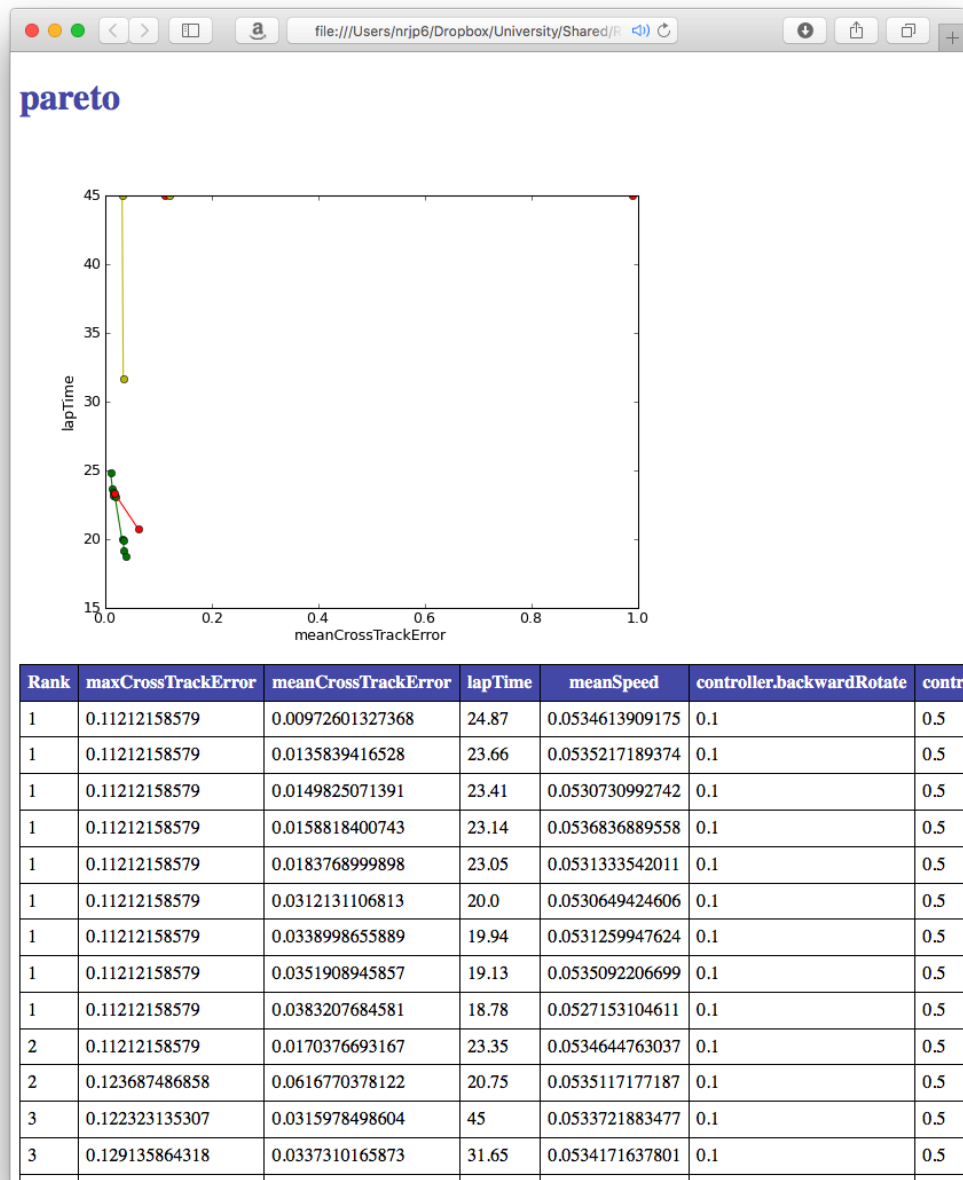
Figure 43: DSE results

31.1 A Genetic Algorithm for DSE

Inspired by processes found in nature, genetic algorithms "breed" new generations of optimal CPS designs from the previous generation's best candidates. This mimics the concept of survival of the fittest in Darwinian evolution. Figure 44 represents the structure of a genetic algorithm used for DSE. Several activities are reused from exhaustive DSE: simulation; evaluation of objectives; rank simulated designs; and generate results. The remaining activities are specific to the genetic approach and are detailed in this section.

Generating initial population: Two methods for generating an initial population of designs are supported: randomly, or uniformly across the design space. Generating an initial
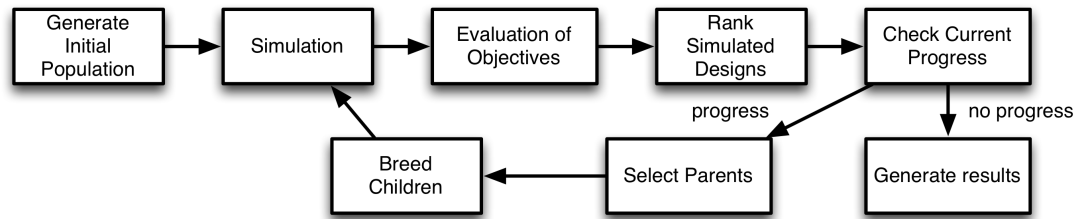
Figure 44: High-level process for DSE Genetic Algorithm

design set which is distributed uniformly could allow parts of the design space to be explored that would otherwise not be explored with a random initial set. This could give us greater confidence that the optimal designs found by the genetic algorithm are consistent with the optimal designs of the total design space.

Selecting parents: Two options for parent selection are supported: random and distributed. Random selection means that two parents are chosen randomly from the non-dominated set (NDS). There is also a chance for parents to be selected which are not in the NDS, potentially allowing different parts of the design space to be explored due to a greater variety of children being produced.

An intelligent approach involves calculating the distribution of each design's objectives from other designs in the NDS. One of the parents chosen is the design with the greatest distribution, enabling us to explore another part of the design space which may contain other optimal designs. Picking a parent that has the least distribution suggests that this parent is close to other optimal designs, meaning that perhaps it is likelier to produce optimal designs.

Figure 45(a) shows the fitness roulette by which how much a design solution in Figure 45(b) satisfies the requirements. It can be seen that there exists a relationship where the greater the fitness value a design has, the more likely it is to be selected as a parent. The probability P of design d being selected as a parent can be calculated by:

Solution	Fitness
1	25
2	5
3	40
4	10
5	20

(a) Example fitness roulette   (b) Fitness of designs
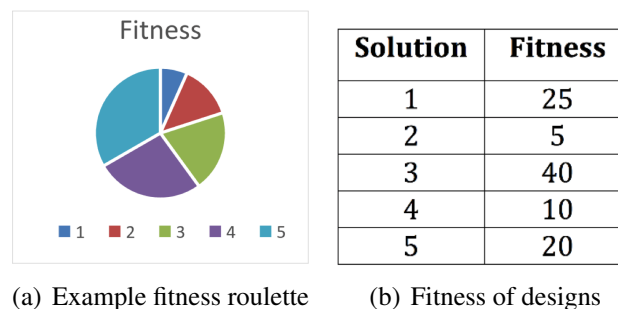
Figure 45: Genetic Algorithm fitness selection

Breeding children: After the parents are selected, the algorithm creates two new children using a process of crossover. Figure 46 shows this process. Mutation could also occur, where a randomly chosen parameter's value is replaced by another value defined in the initial DSE configuration, producing new designs to explore other parts of the design space.

Checking current progress: Progression is determined by the change in the NDS on each
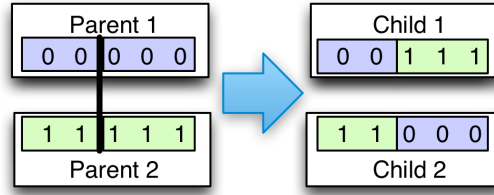
Figure 46: Depiction of genetic crossover

iteration. It is possible to tune the number of iterations without progress before termination.

31.2 Measuring Effectiveness

To provide guidance on selection and tuning of a specific algorithm to a DSE situation it is necessary that there is a means for experimenting with the algorithm parameters and also means for evaluating the resulting performance. To this end an experiment was devised that supports exploration of these parameters using a range of design spaces as the subject. The experiment is based upon generating a ground truth for a set of design spaces such that the composition of each Pareto front is known and we may assess the cost and accuracy of the genetic algorithm's attempt to reach it. A limiting factor for these design spaces is that they must be exhaustively searched and so there are current four of these all based upon the line follow robot: an 81-point and a 625-point design space where the sensor positions are varied and a 216-point and 891-point design spaces where the controller parameters are varied. There are three measures applied to each result that target the tension between trading off the cost of running a DSE against the accuracy of the result

Cost: The simplest of the measures is the cost of the performing the search and here it is measured by the number of simulations performed to reach a result. For the purposes of comparison across the different design spaces, this cost is represented as a proportion of the total number of designs

$$cost = \frac{|Simulations\ Run|}{|Design\ Space|}$$

Accuracy: The ground truth exhaustive experiments provide us with the Pareto Front for that design space and each DSE experiment returns a non-dominated set of best designs found. Here the accuracy measure considers how many of the designs in the genetic non-dominated set are actually the best designs possible. It is measured by finding the proportion of points in the genetic NDS that are also found in the ground truth Pareto front.

$$accuracy = \frac{|GeneticNDS \cap ExhaustiveNDS|}{|GeneticNDS|}$$

Generational Distance: The accuracy measure tells us something about the points in the genetically found NDS that are also found in the exhaustive NDS (Van Veldhuizen & Lamont, 2000). The generational distance gives us a figure indicating the total distance between the genetic NDS and the exhaustive NDS. It is calculated by computing the sum of the distance between each point in the genetic NDS and its closest point in the exhaustive NDS and dividing this by the total number of points.

$$generational\ distance = \frac{\sqrt{(\sum_{i=1}^{n} d_i^2)}}{n}$$

31.3 Genetic DSE Experiments and Results

The DSE experiments involved varying three parameters of the genetic algorithm and repeating each set of parameters with each design space five times. The parameters of the genetic algorithm varied were:

Initial population size: The initial population size took one of three values. All design spaces were tested using an initial population of 10 designs, they were also tested with initial populations equal to 10% of the design space and 25% of the design space. These are represented on the left hand graphs by the 10, 10% and 25% lines.

Progress check conditions: The number of rounds the genetic algorithm would continue if there was no progress observed was tested with three values, 1, 5 and 10. These are represented on the right hand graphs with the 1, 5 and 10 lines.

Algorithm options: There are two variants of the genetic algorithm, phase 1 with random initial population and random parent selection, and phase 3 which give an initial population distributed over the design space and where parent selection is weighted to favour diverse parents. The phase one experiments are on the left hand side of the graphs, with points labelled '<design space size>-p1' while the phase three experiments are on the right labelled '<design space size>-p3'.

The results of the simulations are shown in graph form below. Each point graphed is the averaged result of the five runs of each set of parameters. Figure 47, shows the graphs of cost of running the DSEs. Encouragingly there is a slight trend of the cost of DSE reducing as a proportion of the design space as the design space size increases. As expected the cost was greater with larger initial populations but the cost did not vary when changing the progress check condition as much as expected.

Figure 48 shows the graphs of DSE accuracy. There is again a slight downward trend as design space size increases, meaning that there is a slight increase in the number of points in the genetic NDS that are not truly optimal. As expected the larger initial population generally resulted in more accurate NDS, this was also true of using the largest value for the progress check condition.

Figure 49 presents the generational distance results. Here we find that the results are generally low, with the exception of the 891-point design space which is significantly worse, the reason for this is still to be determined. The largest initial design space resulted in the lowest (best) values as did using a progress check condition value of five.

31.4 Selecting Approaches based on Design Space

The choice of which search algorithm to use when performing DSE is dependant on one factor, and that is a comparison of the 'simulations required' compared to the 'simulation budget', before describing what to do with the comparison, it is first necessary to explain those terms.

The number of 'simulations required' is dependant on the number of different design alternatives that the DSE is supposed explore, but there also other factors, specifically repetitions and scenarios. The 'design' part of the number of simulations required is determined by multiplying together the number of values each parameter may adopt, since this gives the number of unique designs. If the DSE configuration includes parameter constraints then the number of
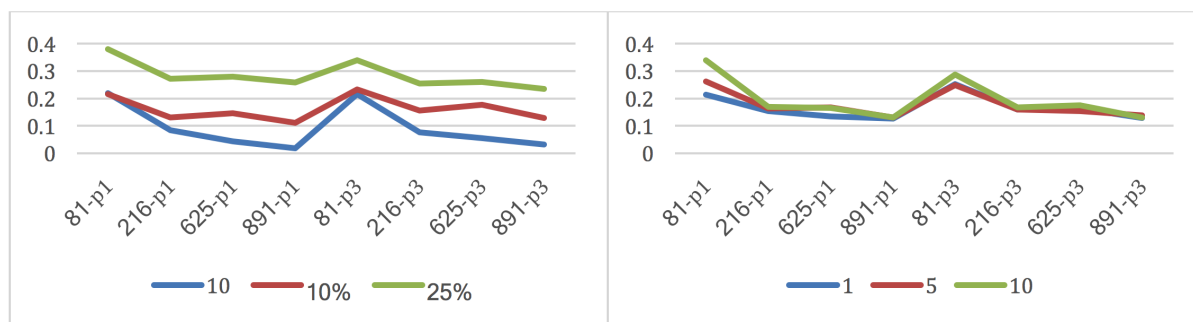
Figure 47: Cost of DSE, number of simulations run as proportion of the total design space.
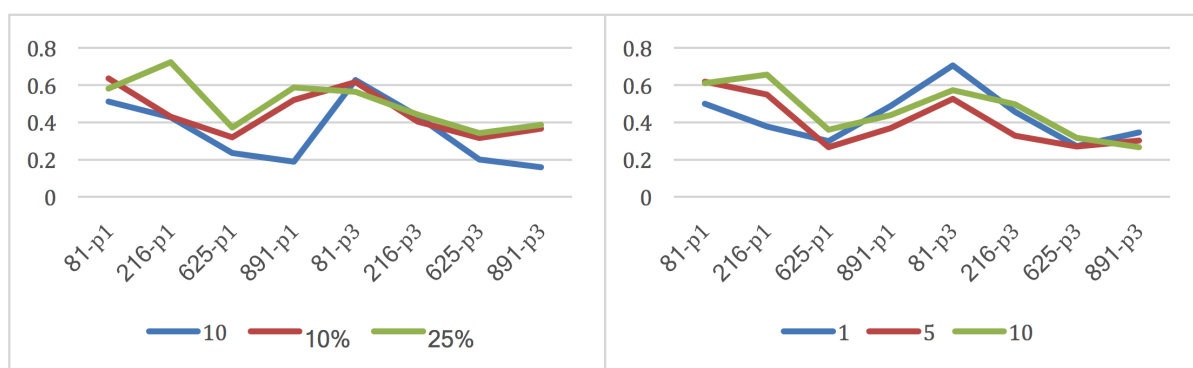

Figure 48: Accuracy of DSE, proportion of genetic NDS found in exhaustive NDS.
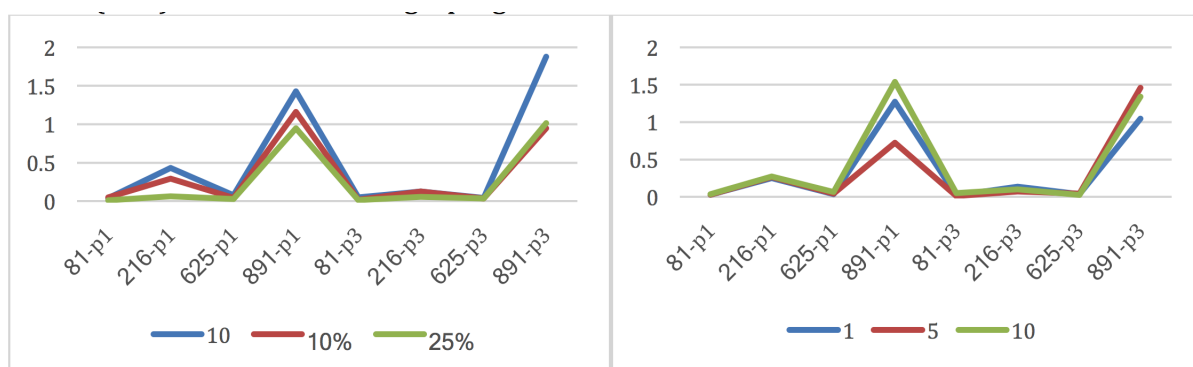
Figure 49: Gap between Genetic NDS and Exhaustive NDS. The vertical axis has no meaningful units, a smaller number is better.

valid designs will be lower since some parameter combinations will fail to meet the constraints. For example, A line follower robot with two sensors, where each sensor has three possible x position values and three possible y position values, would have a design space size of 81, however if those parameters are constrained so designs must be symmetrical, i.e. the x and y values of each sensor must be identical, then the design space only has nine points.

If a simulation model contains random elements, such a noisy inputs to sensors or models of dropped messages on a network, then this leads to the simulation results being non-deterministic. In this case, it will be necessary to perform repeated simulations of the same design with the same starting conditions to account for the random variation.

'Scenarios' refer to the environment around the actual system-under-test in the simulation, for example, in the case of a line following robot, the environment could include the map that the robot is to follow along with other factors such as the intensity of the ambient light. If there is a desire to perform simulations under different scenarios, then the number of scenarios must also be taken into account when determining the required number of simulations. The final required number of simulations then is the product of the design space size, the number of repetitions and the number of scenarios.

The simulation budget term refers to the maximum number of simulations that a user may perform as part of DSE experiment. It is a matter for the user to determine the value for this budget, but it could be determined by determining the amount of CPU time allocated to the DSE and dividing it by the time to run a simulation and compute the objective values.

The decision of whether to use the exhaustive search algorithm or a closed loop search can be made by comparing the number of simulations required with the simulation budget. If the simulation budget is greater than or equal to the required number of simulations, then an exhaustive search should be used as this guarantees to find the optimal designs given the design parameter values, if the budget is less than the required number of simulations required then a closed loop approach is needed.

31.4.1 Parameters for a Genetic Search

If the decision is made to perform a genetic search, then it is important to note that there are two parameters that affect how the algorithm behaves and will have an effect on the outcome. The first of these parameters is the initial population size, this defines how many random designs are generated at the start of the search as a seed for the process. A general rule for this initial population size is that it should be 10 times the number of dimensions (parameters) [DGH07], with the caveat that as the number of dimensions increases, this multiplier must also also increase.

The second parameter is the termination condition, or the number generations the algorithm will continue without seeing progress before it terminates. The genetic algorithm measures progress by looking at the designs that make up the non-dominated set of the Pareto analysis. The only way membership of this set can change between two generations is if better designs, according to the objective measures, have been found, so if membership changes then the search is making progress towards finding better designs. It is not unusual for the algorithm to breed new designs that are not better than those currently in the non-dominated set and so to have a generation that does not show progress, but then to make progress in a subsequent generation. Thus, the number of generations without progress parameter is used to relax the termination condition to permit generations without progress without stopping. Increasing this value will

increase the probability that the search will not become stuck in some local optima in the results and may progress to find better designs. There is a cost associated with increasing this value since, as the algorithm produces two new designs per generation, there will be two times the number of generations without progress simulations run at the end of the process that do not lead to better results [FGPL17].

31.4.2 Iterative Exhaustive Search

An alternative to the genetic search, which is automated, is to use repeated exhaustive searches to home in on better regions of the design space. In this approach the user would plan to perform multiple DSE experiments, each using some portion of their total simulation budget. The first DSE experiment is used to cover the whole range of the design space, but not including all values for each parameter. In this way the first DSE is used to locate regions of interest within the design space. The regions of interest are areas of the design space that produced the better designs according to the ranking results, with the bounds of the 'area' defined by the parameter values that produced good results. The user then divides up their remaining simulation budget between the one or more areas of interest and perform further DSE on those areas. Figure 50 shows an example initial search, with the blue dots indicating simulations performed and the green areas giving the best results. The user then divides their remaining simulation budget among the three green areas of interest, searching each with a higher resolution in an attempt to extract the best results from each, Figure 51.
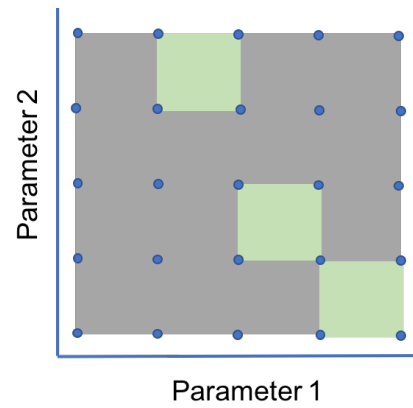
Figure 50: Step 1 of an iterative search. The best results being found in the green regions
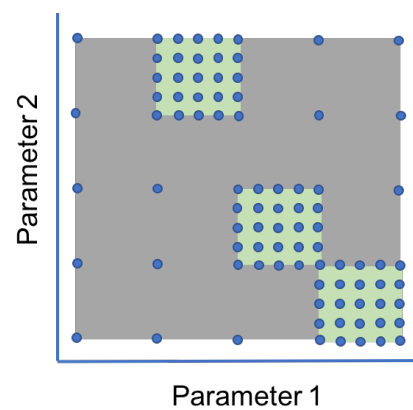
Figure 51: Step 2 of an iterative search. The green regions are searched with a higher resolution to find the best results

References

[ACM⁺16] Nuno Amalio, Ana Cavalcanti, Alvaro Miyazawa, Richard Payne, and Jim Woodcock. Foundations of the SysML for CPS modelling. Technical report, INTO-CPS Deliverable, D2.2a, December 2016.

[APCB15] Nuno Amalio, Richard Payne, Ana Cavalcanti, and Etienne Brosse. Foundations of the SysML profile for CPS modelling. Technical report, INTO-CPS Deliverable, D2.1a, December 2015.

[BFG⁺12] Jan F. Broenink, John Fitzgerald, Carl Gamble, Claire Ingram, Angelika Mader, Jelena Marincic, Yunyun Ni, Ken Pierce, and Xiaochen Zhang. Methodological guidelines 3. Technical report, The DESTECS Project (INFSO-ICT-248134), October 2012.

[BFL⁺94] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[BH17] Jörg Brauer and Miran Hasanagic. Implementation of a model-checking component. Technical report, INTO-CPS Deliverable, D5.3c, December 2017.

[BLL⁺17] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Casper Thule, Anders Franz Terkelsen, Carl Gamble, Adrian Pop, Etienne Brosse, Jörg Brauer, Florian Lapschies, Marcel Groothuis, Christian Kleijn, and Luis Diogo Couto. INTO-CPS Tool Chain User Manual. Technical report, INTO-CPS Deliverable, D4.3a, December 2017.

[Blo14] Torsten Blochwitz. Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads, July 2014.

[BQ15] Etienne Brosse and Imran Quadri. COE Contracts from SysML. Technical report, INTO-CPS Deliverable, D4.1c, December 2015.

[BQ16] Etienne Brosse and Imran Quadri. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.2c, December 2016.

[Bro17] Etienne Brosse. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.3c, December 2017.

[BW98] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CE81] E. M. Clarke and A. E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *IBM Logics of Programs Workshop*, volume LNCS 131. Springer Verlag, 1981.

[CFT⁺17] Ana Cavalcanti, Simon Foster, Bernhard Thiele, Jim Woodcock, and Frank Zeyda. Final Semantics of Modelica. Technical report, INTO-CPS Deliverable, D2.3b, December 2017.

[CFWZ17]   Ana Cavalcanti, Simon Foster, Jim Woodcock, and Frank Zeyda. Multi-Model Linking Semantics. Technical report, INTO-CPS Deliverable, D2.3d, December 2017.

[CGP99]   E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[CKOS04]   Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[CKOS05]   Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[CMC+13]   Joey Coleman, Anders Kaels Malmos, Luis Couto, Peter Gorm Larsen, Richard Payne, Simon Foster, Uwe Schulze, and Adalberto Cajueiro. Third release of the COMPASS tool — symphony ide user manual. Technical report, COMPASS Deliverable, D31.3a, December 2013.

[CV03]   E.M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[DAB+15]   Lipika Deka, Zoe Andrews, Jeremy Bryans, Michael Henshaw, and John Fitzgerald. D1.1 definitional framework. Technical report, The TAMS4CPS Project, April 2015.

[DGH07]   Pedro A Diaz-Gomez and Dean F Hougen. Initial population for genetic algorithms: A metric approach. In *GEM*, pages 43–49, 2007.

[FE98]   Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[FGP17]   John Fitzgerald, Carl Gamble, and Ken Pierce. Methods Progress Report 3. Technical report, INTO-CPS Deliverable, D3.3b, December 2017.

[FGPL17]   John Fitzgerald, Carl Gamble, Richard Payne, and Benjamin Lam. Exploring the Cyber-Physical Design Space. In *Proc. INCOSE Intl. Symp. on Systems Engineering*, volume 27, pages 371–385, Adelaide, Australia, 2017.

[FGPP15a] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Method Guidelines 1. Technical report, INTO-CPS Deliverable, D3.1a, December 2015.

[FGPP15b] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 1. Technical report, INTO-CPS Deliverable, D3.1b, December 2015.

[FGPP16a] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Method Guidelines 2. Technical report, INTO-CPS Deliverable, D3.2a, December 2016.

[FGPP16b] John Fitzgerald, Carl Gamble, Richard Payne, and Ken Pierce. Methods Progress Report 2. Technical report, INTO-CPS Deliverable, D3.2b, December 2016.

[FL98]   John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[FLPV13]   John Fitzgerald, Peter Gorm Larsen, Ken Pierce, and Marcel Verhoef. A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. *Mathematical Structures in Computer Science*, 23(4):726–750, 2013.

[FLV13]   John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2013.

[FLV14]   John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[Gam17]   Carl Gamble. Comprehensive DSE Support. Technical report, INTO-CPS Deliverable, D5.3e, December 2017.

[GF94]   Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. 1st Intl. Conf. on Requirements Engineering*, pages 94–101, April 1994.

[HIL+14]   J. Holt, C. Ingram, A. Larkham, R. Lloyd Stevens, S. Riddle, and A. Romanovsky. Convergence report 3. Technical report, COMPASS Deliverable, D11.3, September 2014.

[HJ98]   Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.

[Hoa85]   Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.

[HPP+15]   Jon Holt, Simon Perry, Richard Payne, Jeremy Bryans, Stefan Hallerstede, and Finn Overgaard Hansen. A model-based approach for requirements engineering for systems of systems. *IEEE Systems Journal*, 9(1):252–262, 2015.

[INC15]   INCOSE. Systems Engineering Handbook. A Guide for System Life Cycle Processes and Activities, Version 4.0. Technical Report INCOSE-TP-2003-002-04, International Council on Systems Engineering (INCOSE), January 2015.

[Jif94]   He Jifeng. A classical mind. chapter From CSP to Hybrid Systems, pages 171–189. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[KLN+17]   Christian König, Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Implementation. Technical report, INTO-CPS Deliverable, D4.3d, December 2017.

[KS08]   Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[LMP+09]   Grace A. Lewis, Edwin Morris, Patrick Place, Soumya Simanta, and Dennis B. Smith. Requirements Engineering for Systems of Systems. In *Systems Conference, 2009 3rd Annual IEEE*, pages 247–252. IEEE, March 2009.

[LNH+16]   Kenneth Lausdahl, Peter Niermann, Jos Höll, Carl Gamble, Oliver Mölle, Etienne Brosse, Tom Bokhove, Luis Diogo Couto, and Adrian Pop. INTO-CPS Traceability Design. Technical report, INTO-CPS Deliverable, D4.2d, December 2016.

[MG13]   Luc Moreau and Paul Groth. PROV-Overview. Technical report, World Wide Web Consortium, 2013.

[MGP+17]   Martin Mansfield, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 3. Technical report, INTO-CPS Deliverable, D3.6, December 2017.

[Mor90]   Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[NLF+13]   Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Model-based engineering of systems of systems. *Submitted to ACM Computing Surveys*, June 2013.

[NN92]   Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications – A Formal Introduction*. John Wiley & Sons Ltd, 1992.

[OLF+17]   Julien Ouy, Thierry Lecomte, Frederik Forchhammer Foldager, Andres Villa Henriksen, Ole Green, Stefan Hallerstede, Peter Gorm Larsen, Luis Diogo Couto, Pasquale Antonante, Stylianos Basagiannis, Sara Falleni, Hassan Ridouane, Hajer Saada, Erica Zavaglio, Christian König, and Natalie Balcu. Case Studies 3, Public Version. Technical report, INTO-CPS Public Deliverable, D1.3a, December 2017.

[PE12]   Birgit Penzenstadler and Jonas Eckhardt. A Requirements Engineering content model for Cyber-Physical Systems. In *RESS*, pages 20–29, 2012.

[PF10]   Richard J. Payne and John S. Fitzgerald. Evaluation of Architectural Frameworks Supporting Contract-based Specification. Technical Report CS-TR-1233, School of Computing Science, Newcastle University, December 2010.

[PGP+16]   Richard Payne, Carl Gamble, Ken Pierce, John Fitzgerald, Simon Foster, Casper Thule, and Rene Nilsson. Examples Compendium 2. Technical report, INTO-CPS Deliverable, D3.5, December 2016.

[PHP+14]   Simon Perry, Jon Holt, Richard Payne, Jeremy Bryans, Claire Ingram, Alvaro Miyazawa, Luís Diogo Couto, Stefan Hallerstede, Anders Kaels Malmos, Juliano Iyoda, Marcio Cornelio, and Jan Peleska. Final Report on SoS Architectural Models. Technical report, COMPASS Deliverable, D22.6, September 2014. Available at http://www.compass-research.eu/.

[Plo81]   Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pnu77]   Amir Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57. ACM, November 1977.

[SS71]   Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer language. Technical Report PRG-6, Oxford Programming Research Group Technical Monograph, 1971.

[Sys12]   OMG Systems Modeling Language (OMG SysML^{TM}). Technical Report Version 1.3, SysML Modelling team, June 2012. http://www.omg.org/spec/SysML/1.3/.

[Tho13]   Haydn Thompson, editor. *Cyber-Physical Systems: Uplifting Europe's Innovation Capacity*. European Commission Unit A3 - DG CONNECT, December 2013.

[UPL06]   Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 2006.

[vA10]   Job van Amerongen. *Dynamical Systems for Creative Technology*. Controllab Products, Enschede, Netherlands, 2010.

[WG-92]   RTCA SC-167/EUROCAE WG-12. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178B, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 1992.

[WGS$^+$14] Stefan Wiesner, Christian Gorldt, Mathias Soeken, Klaus-Dieter Thoben, and Rolf Drechsler. Requirements engineering for cyber-physical systems - challenges in the context of "industrie 4.0". volume 438 of *IFIP Advances in Information and Communication Technology*, pages 281–288. Springer, 2014.

[ZCWO17]   Frank Zeyda, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. SysML Foundations: Case Study. Technical report, INTO-CPS Deliverable, D2.3a, December 2017.

[ZFC$^+$17] Frank Zeyda, Simon Foster, Ana Cavalcanti, Jim Woodcock, and Julien Ouy. A Mechanised FMI Semantics. Technical report, INTO-CPS Deliverable, D2.3c, December 2017.

Part III

Appendices

Glossary

20-sim The 20-sim tool can represent continuous time models in a number of ways. The core concept is that of connected *blocks*.

Abstraction Models may be abstract "in the sense that aspects of the product not relevant to the analysis in hand are not included" [FL98]. CPS models may reasonably contain multiple levels of abstraction, for representing views of individual constituent systems and for the view of the CPS level. Adapted from [HIL$^+$14].

Architecture The term architecture has many different definitions, and range in scope depending upon the scale of the product being 'architected'. In the INTO-CPS project, we use the simple definition from [PHP$^+$14]: "an architecture defines the major elements of a system, identifies the relationships and interactions between the elements and takes into account process. An architecture involves both a definition of structure and behaviour. Importantly, architectures are not static but must evolve over time to reflect the change in a system as it evolves to meet changes to its requirements."

Architecture Diagram In the INTO-CPS project, a diagram refers to the symbolic representation of information contained in a model.

Architectural Framework "A defined set of viewpoints and an ontology" and "is used to structure an architecture from the point of view of a specific industry, stakeholder role set, or organisation. [HIL$^+$14]. [HIL$^+$14].

Architecture Structure Diagram (ASD) The INTO-CPS SysML profile ASDs specialise SysML block definition diagrams to support the specification of a system architecture described in terms of a system's components.

Architecture View "work product expressing the architecture of a system from the perspective of specific system concerns" [PHP$^+$14].

Bond graph Bond graphs offer a domain-independent description of a physical system's dynamics, realised as a directed graph. The vertices of these graphs are idealised descriptions of physical phenomena, with their edges (*bonds*) describing energy exchange between vertices.

Co-model "The term *co-model* is used to denote a model comprising a DE model, a CT model and a contract" [BFG$^+$12]. A related term ***multi-model*** is a model comprising any combination of constituent DE and CT models.

Code generation Transformation of a model into generated code suitable for compilation into one or more target languages (e.g. C or Java).

Collaborative simulation (co-simulation) The simultaneous, collaborative, execution of models and allowing information to be shared between them. The models may be CT-only, DE-only or a combination of both.

Co-simulation Configuration The configuration that the COE needs to initialise a co-simulation. It contains paths to all FMUs, their inter connection, parameters and step size configuration. When this is combined with a start and end time, a co-simulation can be performed.

Co-simulation Orchestration Engine (COE) The Co-simulation Orchestration Engine combines existing co-simulation solutions (FMUs) and scales them to the CPS level, allowing

CPS co-models to be evaluated through co-simulation. The COE will also allow real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

Component The constituent elements of a system.

Connections Diagram (CD) The INTO-CPS SysML profile CDs specialise SysML internal block diagrams to convey the internal configuration of the system's components and the way they are connected.

Constituent Model A constituent model comprising a multi-model.

Continuous Time (CT) model A model with state that can be changed and observed *continuously* [vA10], and are described using either explicit continuous functions of time either implicitly as a solution of differential equations.

Context In requirements engineering, a *context* is the point of view of some system component or domain, or interested stakeholder.

Cyber Physical System (CPS) Cyber-Physical Systems "refer to ICT systems (sensing, actuating, computing, communication, etc.) embedded in physical objects, interconnected (including through the Internet) and providing citizens and businesses with a wide range of innovative applications and services" [Tho13, DAB⁺15].

Discrete Event (DE) model A model with state that can be changed and observed only at fixed, *discrete*, time intervals [vA10].

Denotational Semantics Where an operational semantics defines how a program is executed, a denotational approach defines a language in terms of denotations, in the form of abstract mathematical objects, which represent the semantic function that maps over the inputs and outputs of a program [SS71].

Design Alternatives Where two or more models represent different possible solutions to the same problem. Each choice involves making a selection from alternatives on the basis of criteria that are important to the developer, such as cost or performance. The alternative selected at each point constrains the range of design alternatives that may be viable next steps forward from the current position.

Design Architecture The design architectural model of the system is effectively a multi-model. The INTO-CPS SysML profile [APCB15] is designed to enable the specification of CPS design architectures, which emphasises a decomposition of a system into *subsystems*, where each subsystem is modelled separately in isolation using a special notation and tool designed for the domain of the subsystem.

Design Parameter A *design parameter* is a property of a model that can be used to affect the model's behaviour, but that remains constant during a given simulation [BFG⁺12].

Design Space "The *design space* is the set of possible solutions for a given design problem" [BFG⁺12].

Design-Space Exploration (DSE) "an activity undertaken by one or more engineers in which they build and evaluate co-models in order to reach a design from a set of requirements" [BFG⁺12].

Effort and Flow The energy exchanged in 20-sim is the product of *effort* and *flow*, which map to different concepts in different domains, for example voltage and current in the electrical domain.

Environment A system's *environment* is everything outside of the system. The behaviour exhibited by the environment is beyond the direct control of the developer [BFG+12].

Evolution This refers to the ability of a system to benefit from a varying number of alternative system components and relations, as well as its ability to gain from the adjustments of the individual components' capabilities over time (Adjusted from SoS [NLF+13]).

Foundations Developer An individual who uses the developed foundations and associated tool support (see Section 8) to reason about the development of tools.

Functional Mockup Interface (FMI) The Functional Mock-up Interface (FMI) is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [Blo14].

Functional Mockup Unit (FMU) Component that implements FMI is a Functional Mockup Unit (FMU) [Blo14].

Hardware-in-the-Loop (HiL) Testing In *HiL* there is (target) hardware involved, thus the FMU representing the hardware in a co-simulation is mainly a wrapper that interacts (timed) with this hardware; it is perceivable that realisation heavily depends on hardware interfaces and timing properties.

Holistic Architecture The aim of a holistic architecture is to identify the main units of functionality of the system reflecting the *terminology and structure of the domain of application*. It describes a conceptual model that highlights the main units of the system architecture and the way these units are connected with each other, taking a holistic view of the overall system.

Hybrid-CSP This is a continuous version of CSP defined originally by He Jifeng [Jif94]. It will be used as a basis to inform the design of INTO-CSP.

Hybrid Model A model which contains both DE and CT elements.

Interface "Defines the boundary across which two entities meet and communicate with each other" [HIL+14]. Interfaces may describe both digital and physical interactions: digital interfaces contain descriptions of operations and attributes that are *provided* and *required* by components. Physical interfaces describe the flow of physical matter (for example fluid and electrical power) between components.

INTO-CPS Application The INTO-CPS Application is a front-end to the INTO-CPS tool chain. The application allows the specification of the co-simulation configuration to be orchestrated by the COE, and the co-simulation execution itself. The application also provides access to features of the tool chain without an existing user interface (such as design space exploration and model checking).

INTO-CPS tool chain The INTO-CPS tool chain is a collection of software tools, based centrally around FMI-compatible co-simulation, that supports the collaborative development of CPSs.

INTO-CSP A version of CSP, which will be used to provide a model for the SysML-FMI

profile, FMI, VDM-RT and Modelica semantics. It is a front end for a UTP theory of reactive concurrent continuous systems customised for the needs of INTO-CPS.

Master Algorithm A Master Algorithm (MA) controls the data exchange between FMUs and the synchronisation of all simulation solvers [Blo14].

Model A potentially partial and abstract description of a system, limited to those components and properties of the system that are pertinent to the current goal [HIL$^+$14]. "A model is a simplified description of a system, just complex enough to describe or study the phenomena that are relevant for our problem context" [vA10]. A model "may contain representations of the system, environment and stimuli" [FLV14]

Model Checking (MC) An analysis technique that exhaustively checks whether the model of the system meets its specification [CGP99], which is typically expressed in some temporal logic such as *Linear Time Logic (LTL)* [Pnu77] or *Computation Tree Logic (CTL)* [CE81].

Model Description The model description file is an XML file that supplies a description of all properties of a model (for example input/output variables) [Blo14].

Model-in-the-Loop (MiL) Testing in *MiL* the test object of the test execution is a (design) model, represented by one or more FMUs. This is similar to the SiL (if e.g., the SUT is generated from the design model), but MiL can also imply that running the SUT-FMU has a representation on model level; e.g., a playback functionality in the modelling tool could some day be used to visualise a test run.

Modelling "The activity of creating models" [FLV14]. See also **co-modelling** and **multi-modelling**.

Modelica Modelica is an "object-oriented language for modelling of large, complex, and heterogeneous physical systems" [FE98]. Modelica models are described by *schematics*, also called *object diagrams*, which consist of connected components. Components are connected by ports and are defined by sub components or a textual description in the Modelica language.

Multi-model "A model comprising *multiple* constituent DE and CT models".

Non-dominated Set The Non-dominated set (NDS) is the current set of best results according to a Pareto analysis. For a result exist in the NDS it must be true that it is not possible to find another result in the set of all results that improves on one property of the result without degrading another property of the result.

Non-functional Property Non-functional properties (NFPs) pertain to characteristics other than functional correctness. For example, reliability, availability, safety and performance of specific functions or services are NFPs that are quantifiable. Other NFPs may be more difficult to measure [PF10].

Objective Criteria or constraints that are important to the developer, such as cost or performance

Port 20-sim blocks may have input and output *ports* that allow data to be passed between them. In SysML, blocks own ports — the points of interaction between blocks.

Proof The process of showing how the validity of one statement is derived from others by applying justified rules of inference [BFL$^+$94].

Provenance "Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness." [MG13].

Refinement Refinement is a verification and formal development technique pioneered by [BW98] and [Mor90]. It is based on a behaviour preserving relation that allows the transformation of an abstract specification into more and more concrete models, potentially leading to an implementation.

Requirement A requirement is a statement of need and may impose restrictions, define system capabilities or identify qualities of a system and should indicate some value or use for the different stockholders of a CPS.

Requirements Engineering (RE) The process of the specification and documentation of requirements placed upon a CPS.

Semantics Describes the meaning of a (grammatically correct) language [NN92].

Software-in-the-Loop (SiL) Testing In *SiL* testing the object of the test execution is an FMU that contains a software implementation of (parts of) the system. It can be compiled and run on the same machine that the COE runs on and has no (defined) interaction other than the FMU-interface.

SoS-ACRE System of Systems Approach to Context-based Requirements Engineering [HPP$^+$15], an approach adapted from standard systems engineering, tailored for systems of systems (SoSs).

Structural Operational Semantics (SOS) Describes how the individual steps of a program are executed on an abstract machine [Plo81]. An SOS definition is akin to an interpreter in that it provides the meaning of the language in terms of relations between beginning and end states. The relations are defined on a per-construct basis. Accompanying the relations are a collection of semantic rules which describe how the end states are achieved.

SysML The systems modelling language (SysML) [Sys12] extends a subset of the Unified Modelling language (UML) to support modelling of heterogeneous systems.

System "A combination of interacting elements organized to achieve one or more stated purposes" [INC15].

System boundary The *system boundary* is the common frontier between the system and its environment. System boundary definition is application-specific [BFG$^+$12].

System of Systems (SoS) "A System of Systems (SoS) is a collection of constituent systems that pool their resources and capabilities together to create a new, more complex system which offers more functionality and performance than simply the sum of the constituent systems" [HIL$^+$14]. CPSs may exhibit the characteristics of SoSs.

System Under Test "The system currently being tested for correct behaviour. An alias for system of interest, from the point of view of the tester. The same concept can be extended from systems engineering to SoS engineering, changing the focus from a single system

of interest to an SoS under test.
The system of systems currently being tested for correct behaviour" [HIL$^+$14].

Test Automation Test Automation (TA) is defined as the machine assisted automation of system tests. In INTO-CPS we concentrate on various forms of *model-based testing*, centering on testing system models against their requirements.

Test Case A finite structure of input and expected output [UPL06].

Test model Specifies the expected behaviour of a system under test. Note that a test model can be different from a design model. It might only describe a part of a system under test that is to be tested and it can describe the system on a different level of abstraction [CMC$^+$13].

Test procedures Detailed instructions for the set-up and execution of a set of test cases, and instructions for the evaluation of results of executing the test cases [WG-92, CMC$^+$13].

Test suite A collection of test procedures.

Tool Chain User An individual who uses the INTO-CPS Tool Chain and its various analysis features.

Traceability The association of one model element (e.g. requirements, design artefacts, activities, software code or hardware) to another. ***Requirements traceability*** "refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [GF94].

Unifying Theories of Programming (UTP) The Unifying Theories of Programming (UTP) [HJ98] is a technique to for describing language semantics in a unified framework. A theory of a language is composed of an *alphabet*, a *signature* and a collection of ***healthiness conditions***.

Variable A *variable* is feature of a model that may change during a given simulation [BFG$^+$12].

VDM-RT VDM-RT is based upon the ***object-oriented*** paradigm where a model is comprised of one or more ***objects***. An object is an instance of a ***class*** where a class gives a definition of zero or more ***instance variables*** and ***operations*** an object will contain. Instance variables define the identifiers and types of the data stored within an object, while operations define the behaviours of the object.

Workflow A sequence of **activities** performed to aid in modelling. A workflow has a defined purpose, and may cover a subset of the CPS engineering development lifecycle.