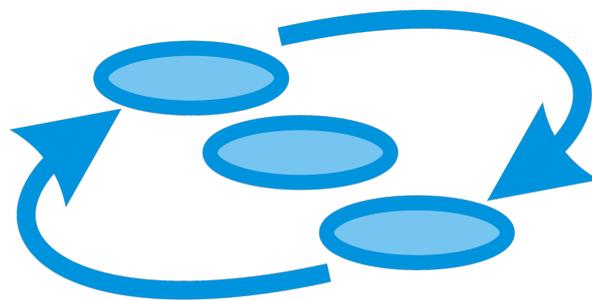




Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs



INTO-CPS

Method Guidelines 2

Deliverable Number: D3.2a

Version: 1.1

Date: December 2016

Public Document

<http://into-cps.au.dk>

Contributors:

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

Editors:

Ken Pierce, UNEW

Reviewers:

Christian König, TWT
Etienne Brosse, ST
Martin Peter Christiansen, AI

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softteam	ST		

Document History

Ver	Date	Author	Description
0.1	04-04-2016	Richard Payne	Initial document
0.2	19-05-2016	Richard Payne	Initial requirements engineering material
0.3	31-09-2016	Richard Payne	SysML extensions detailed
0.4	10-10-2016	Richard Payne	Updated requirements engineering
0.5	12-10-2016	Richard Payne	DSE SysML added
0.6	26-10-2016	Ken Pierce	Network modelling section added
0.7	01-11-2016	Ken Pierce	Workflows section added
0.8	01-11-2016	Richard Payne	Version for review
0.9	07-12-2016	Ken Pierce	First revision after comments and new section headers added
1.0	08-12-2016	Ken Pierce	New Initial Multi-modelling section added
1.1	13-12-2016	Richard Payne	SysML diagrams updated. Final version

Abstract

This document is the second of three that will give methods guidance for the INTO-CPS technologies. It is aimed at end users of the technologies. This second version presents a revised and updated concepts base, which describes the terminology used within INTO-CPS; guidelines on incorporating requirements engineering in the INTO-CPS technologies and workflows; descriptions and examples of the use of the updated INTO-SysML profile; guidance on modelling networks in multi-models; guidelines for the use of Design Space Exploration features of the INTO-CPS tool chain; and guidelines for traceability and model management in INTO-CPS.

Contents

1	Introduction	6
1.1	Overview of Sections	6
1.2	Differences over the Previous Version	7
2	Concepts and Terminology	8
2.1	Systems	8
2.2	Models	8
2.3	Tools	10
2.4	Analysis	11
2.5	Existing Tools and Languages	13
2.6	Formalisms	13
3	Workflows	15
3.1	Workflow Activities	15
3.2	Activities Covered in this Document	16
3.3	Getting Started	18
4	Requirements Engineering	19
4.1	Requirements Engineering and Cyber Physical Systems	19
4.2	Applying SoS-ACRE in the INTO-CPS Tool Chain	20
5	SysML and Multi-modelling	27
6	Initial Multi-modelling	30
6.1	The DE-first Approach	30
6.2	DE-first within INTO-CPS	32
6.3	FMU Creation	32
7	Modelling Networks in Multi-models	34
7.1	Representing VDM Values as Strings	35
7.2	Using the Ether FMU	35
7.3	Consequences of Using the Ether	37
7.4	Modelling True Message Passing and Quality of Service	37
8	Design Space Exploration	39
8.1	Guidelines for Designing DSE in SysML	39
8.2	An Approach to Effective DSE	47
9	Forward Look	52
A	Glossary	57
B	Ether Class Listing	63

1 Introduction

The material in this document is aimed primarily at new and prospective users of the INTO-CPS technologies. Readers seeking a progress report on the methods work should refer to the companion deliverable, D3.1b Methods Progress Report.

The INTO-CPS technologies bring together a variety of baseline tools and technologies. 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 aim of the methods work is to understand how best to use these technologies, to pilot approaches and techniques, and to distill this into a set of guidelines aimed at the users of the technologies—engineers wishing to build cyber-physical systems (CPSs).

This is the second version of the guidelines document. Now that the INTO-CPS technologies have begun to form a stable tool chain, with the INTO-CPS Application as a central point of interaction, we present more concrete guidance on using the technologies. We will continue to expand this guidance as we discover the best ways to work with the INTO-CPS technologies. The next and final version of this document will cover all the engineering processes enabled by the new technology, comprehensively covering INTO-CPS workflows.

1.1 Overview of Sections

Concepts and Terminology (Section 2) This section 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).

Workflows (Section 3) This section identifies activities that engineers designing CPS will undertake. These activities are elements of both existing workflows and those enabled by the INTO-CPS technologies. Various sections of this document provide further guidance on specific activities identified here. This section also presents some initial “getting started” workflows that engineers wishing to try out the INTO-CPS technologies can follow (supported by training material which reflect these).

Requirements Engineering (Section 4) This section 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.

SysML and Multi-modelling (Section 5) A SysML profile for representing CPSs has been defined (see Deliverable D2.2a [ACM⁺16]) and implemented in Modelio (see Deliverable D4.1c [BQ16]). This profile is used to generate model descriptions (to be imported into modelling tool for creating FMUs) and to configure multi-models and co-simulations. The profile is being extended to allow for modelling of CPSs in a way which better reflects the architecture of the system, and to allow for description of analysis techniques such as DSE (see below). This section motivates the extensions to the profile and explains how they affect architectural modelling in INTO-CPS with an illustrative example.

Initial Multi-modelling (Section 6) This section 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].

Modelling Networks in Multi-models (Section 7) Designing software for distributed controllers is an important aspect of CPS design. When using multi-modelling as a design approach, it is useful to also model realistic communications between controllers as well. This section describes how this can be achieved by introducing an FMU that represents an abstract communication mechanism, the *ether*. Initial guidance on the consequences of adopting such an approach is included, as well as extensions to cover quality-of-service modelling.

Design Space Exploration (Section 8) As DSE (Design Space Exploration) is a key analysis technique offered by the INTO-CPS technologies, this section gives guidance on DSE, including the types of search algorithms that can be used to explore a design space, and how the upcoming SysML profile extensions help in the design of experiments. An illustrative example is included based on the line-following robot example. The line-following example is available in the INTO-CPS Application and is described in the Examples Compendium, Deliverable D3.5 [PGP⁺16].

1.2 Differences over the Previous Version

Since this document builds on Deliverable D3.1a [FGPP15a], some material is retained and updated, while other material is entirely new. The list below gives an overview of new and updated material for each section.

Section 2: Concepts and Terminology The concepts base appeared in the previous version, but has been updated to include new terms from the tool chain.

Section 3: Workflows This section appeared in the previous version and primarily gives context to the later sections. The “getting started” material is new.

Section 4: Requirements Engineering This section is new and was produced after a need was identified to provide guidance on requirements within INTO-CPS.

Section 5: SysML and Multi-modelling This section has been updated significantly to reflect the expanded INTO-CPS SysML profile.

Section 6: Initial Multi-modelling This section is entirely new.

Section 7: Modelling Networks in Multi-models This section is entirely new.

Section 8: Design Space Exploration This section is entirely new.

2 Concepts and Terminology

This section introduces the basic concepts used in the INTO-CPS project. This is an update of the concept base in [FGPP15a] with new and adjusted terminology. 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.

2.1 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.

2.2 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

¹Further 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.

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)* 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 2.5. 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 Section 5. 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 2.4 for concepts relating to design space exploration.

2.3 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**.

2.4 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].

2.5 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-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.

2.6 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 [Plø81]. 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.

²<http://overturetool.org/>

³<http://www.20sim.com/>

⁴<https://www.openmodelica.org/>

⁵<http://www.modelio.org/>

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 specification 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.

3 Workflows

In this section, we list the types of activities that we expect engineers to perform when using the INTO-CPS technologies. The later sections of this document provide specific guidance on some of these activities, and we aim in the final version of this document to have complete coverage of the activities, as well as descriptions of how they come together to form different workflows that suit different modelling contexts.

3.1 Workflow Activities

The choice of granularity for defining “activities” naturally affects the size of a list such as the one below. We have tried to select a level that is instructive for describing workflows, but one that does not make the described workflows overly long. Activities are grouped into broad categories. Note that these include both existing, embedded systems activities and activities enabled by INTO-CPS, since there is obviously overlap between traditional embedded systems design and CPS design.

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.

 The Crescendo tool.

 OpenModelica.

 20-sim.

Descriptions of these tools can be found in the concepts base at the beginning of this document in Section 2.5.

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 5. 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* (  ) 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 tools 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.

The above activities are summarised in Table 1. Terms in *italics* correspond to INTO-CPS activities that produce traceable artifacts, as described in the traceability ontology in Deliverable D3.1b [FGPP16].

3.2 Activities Covered in this Document

The remaining sections of this document give guidance relevant to the following activities (see Table 1):

- Section 3.3 describes suggested workflows for training including Co-simulation, Design Architectural Modelling, Import a Model Description, and *Multi-modelling* activities.
- Section 4 covers *Design Notes* and Requirements from the **Requirements and Traceability** category.
- Section 5 covers Holistic Architectural Modelling and Design Architectural Modelling from the **Architectural Modelling** category.
- Section 6 and Section 7 cover material relevant to *Multi-modelling* and Cyber Modelling (*Simulation Modelling*) from the **Modelling** and **Design** categories.
- Section 8 covers the Co-simulation and *Define Design Space Exploration Configurations* activities under the **Analysis** category.

Table 1: Activities in existing embedded systems design workflows or enhanced INTO-CPS workflows. Entries in italics correspond to traceable artifacts in INTO-CPS (see Deliverable D3.1b [FGPP15b])

Requirements and Traceability	
<i>Design Notes</i>	
Requirements	
Validation	
Architectural Modelling	
Holistic Architectural Modelling	
Design Architectural Modelling	
<i>Export Model Descriptions</i>	
Modelling	
<i>Import a Model Description</i>	  
Physical Modelling (<i>Simulation Modelling</i>)	 
Cyber Modelling (<i>Simulation Modelling</i>)	
<i>Co-modelling</i>	
<i>Multi-modelling</i>	
Design	
Supervisory Controller Design	
Low Level Controller Design	
Software Design	
Hardware Design	
Analysis	
<i>Create Tests</i>	
<i>Model Checking</i>	
<i>Create Test Oracle</i>	
<i>Create a Configuration</i>	
<i>Define Design Space Exploration Configurations</i>	
<i>Export FMU</i>	  
Co-simulation	 
Prototyping	
<i>Generate Code</i>	  
Hardware-in-the-Loop (HiL) Simulation	
Software-in-the-Loop (SiL) Simulation	
Manual Code Writing	

3.3 Getting Started

A key requirement identified by engineers [FGPP15a] for adoption of new technologies such as INTO-CPS was the need for training materials and “toy examples” to be available. These allow for rapid assessment of the capabilities of technologies, such as identifying technologies and methods might be integrated into current practice, as well provide materials for internal training. For the INTO-CPS technologies, we identify the following three mini-workflows as first steps.

A — Interested User

1. Install tool(s) 
2. Load *Three Tank Water Tank* example 
3. Run co-simulation 

B — Keen User

1. Install tool(s)  
2. Load *Line-following Robot* example 
3. Alter SysML to add new block and connections 
4. Export new multi-model configuration
5. Associate a pre-compiled FMU with the multi-model configuration 
6. Configure and run a co-simulation 

C — Adventurous User

1. Install tool(s)   
2. Load *LineFollowingRobot* exercise 
3. Alter SysML to add new block and connections 
4. In Overture 
 - (a) Import model description
 - (b) Complete controller model
 - (c) Export FMU
5. Add the new FMU to project 
6. Create multi-model configuration 
7. Configure and run a co-simulation 

These workflows are reflected in tutorial material that includes instructions and initial models (*Tutorial 1 — Getting Started*, *Tutorial 2 — Adding FMUs*, and *Tutorial 3 — Generating FMUs*). They intentionally start with analysis activities (i.e. co-simulation) to give context before extending backwards in the design process to modelling activities. We will extend the above by including a variation of *Generating FMUs* for 20-sim and OpenModelica, and will also extend to additional first-steps in architectural modelling, DSE, test automation, traceability, HiL simulation, and code generation.

4 Requirements Engineering

In this section, 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. In this project, we consider the state in the art of RE in both CPS and Systems of Systems (SoSs), reusing a previously defined approach to RE as applicable.

4.1 Requirements Engineering and Cyber Physical Systems

In the academic literature, 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. In Deliverable D3.1b [FGPP15b], we also surveyed several projects that had RE as a focus, or part of their focus. As research in RE in CPS is a nascent field, we consider approaches to RE from the SoS world, rather than defining an approach specifically for CPSs. In particular, we consider SoS-ACRE (System of Systems Approach to Context-based Requirements Engineering) [HPP⁺15], an approach adapted from standard systems engineering, tailored for SoSs – enabling the identification and reasoning about requirements across constituent systems of an SoS and understanding multi-stakeholder contexts.

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

- 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.

4.2 Applying SoS-ACRE in the INTO-CPS Tool Chain

The Views of SoS-ACRE

We first consider the collection of views defined in SoS-ACRE, their applicability to CPS engineering and the INTO-CPS tool chain. Examples of each view are shown in Figures 1, 2, 3 and 4 using technologies relevant to INTO-CPS.

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 – typical for SoSs. 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 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. We feel this is not an obvious fit in CPS engineering and is therefore not required in INTO-CPS.

The SoS-ACRE RE Process

We think that the SoS requirements engineering process of SoS-ACRE may be useful as a starting place for defining requirements in a CPS engineering workflow. The requirements management is considered to be too heavyweight a target for translation. This is largely due to the fact that we are currently less concerned with requirement change/different processes.

We can consider the simplified CPS RE process to be:

1. Identify and record source elements. This would be using a SEV, or simply recording paths to relevant files/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-System 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 above⁶
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. We consider this slightly in the next section, however the bulk of the traceability and provenance work shall be reported in Deliverable 3.2b [FGPP16] and in deliverables next year.

Using technologies with SoS-ACRE

In this final section, we consider initial approaches to realise the relevant SoS-ACRE views using the INTO-CPS technologies and those used by industrial partners. This is not expected to constitute final guidelines on this area, as we would make use of INTO-CPS technology currently in development – namely traceability and provenance support. We therefore describe a range of permutations of the use of models and documents for recording the requirements engineering process described above. In addition, we include discussions on the links between requirements and architectural models – identified above

⁶In 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.

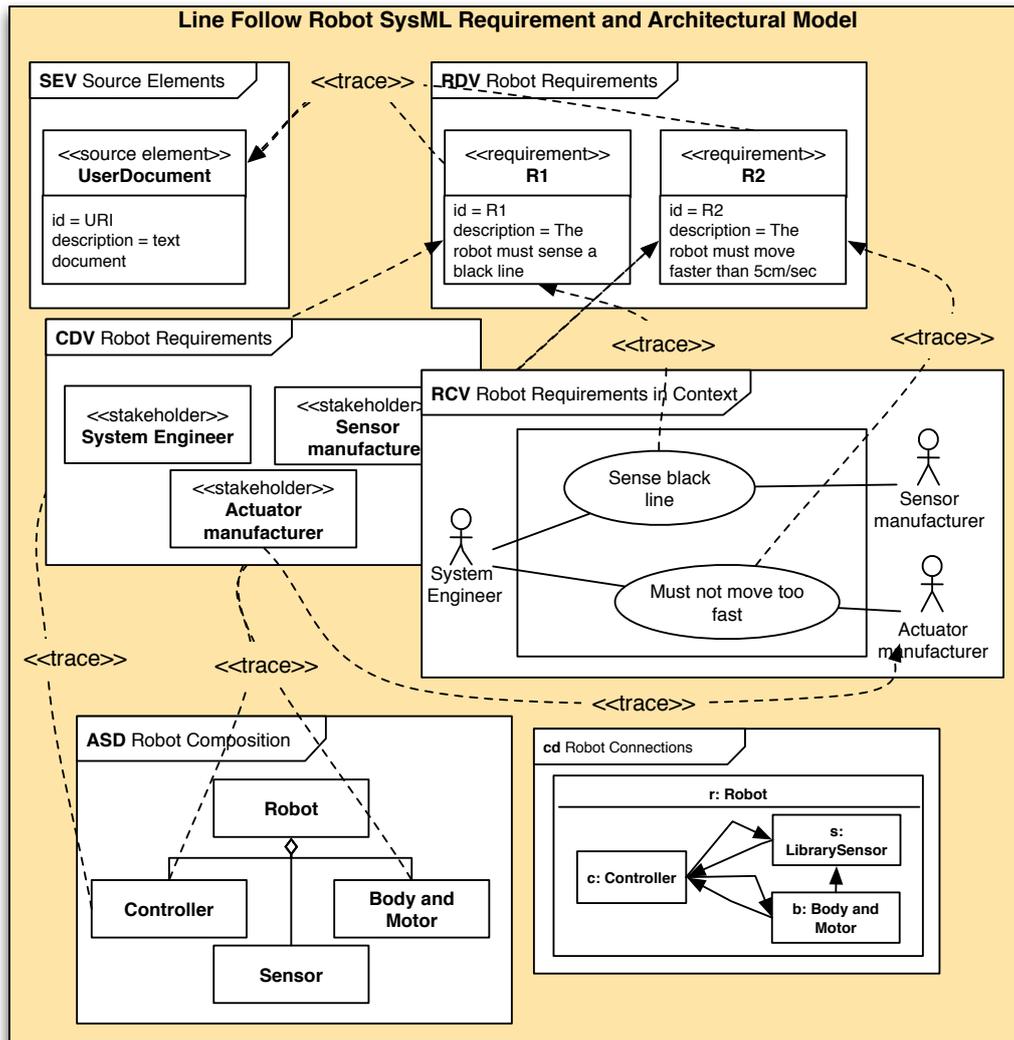


Figure 1: Single SysML model – model overview

as a key method for requirements engineering in CPSs. In the subsequent permutations, we refer to OSLC and Prov links. For more information of the types of links, and their descriptions, see Deliverable D3.2b [FGPP16].

Single SysML model The first permutation is to use a single SysML model for both requirements engineering and architectural modelling. Such a model will contain all SoS-ACRE views (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, using `<< trace >>` relationships. Figure 1 presents an example SysML model with trace relationships.

SysML requirements and SysML architectural models The second option 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

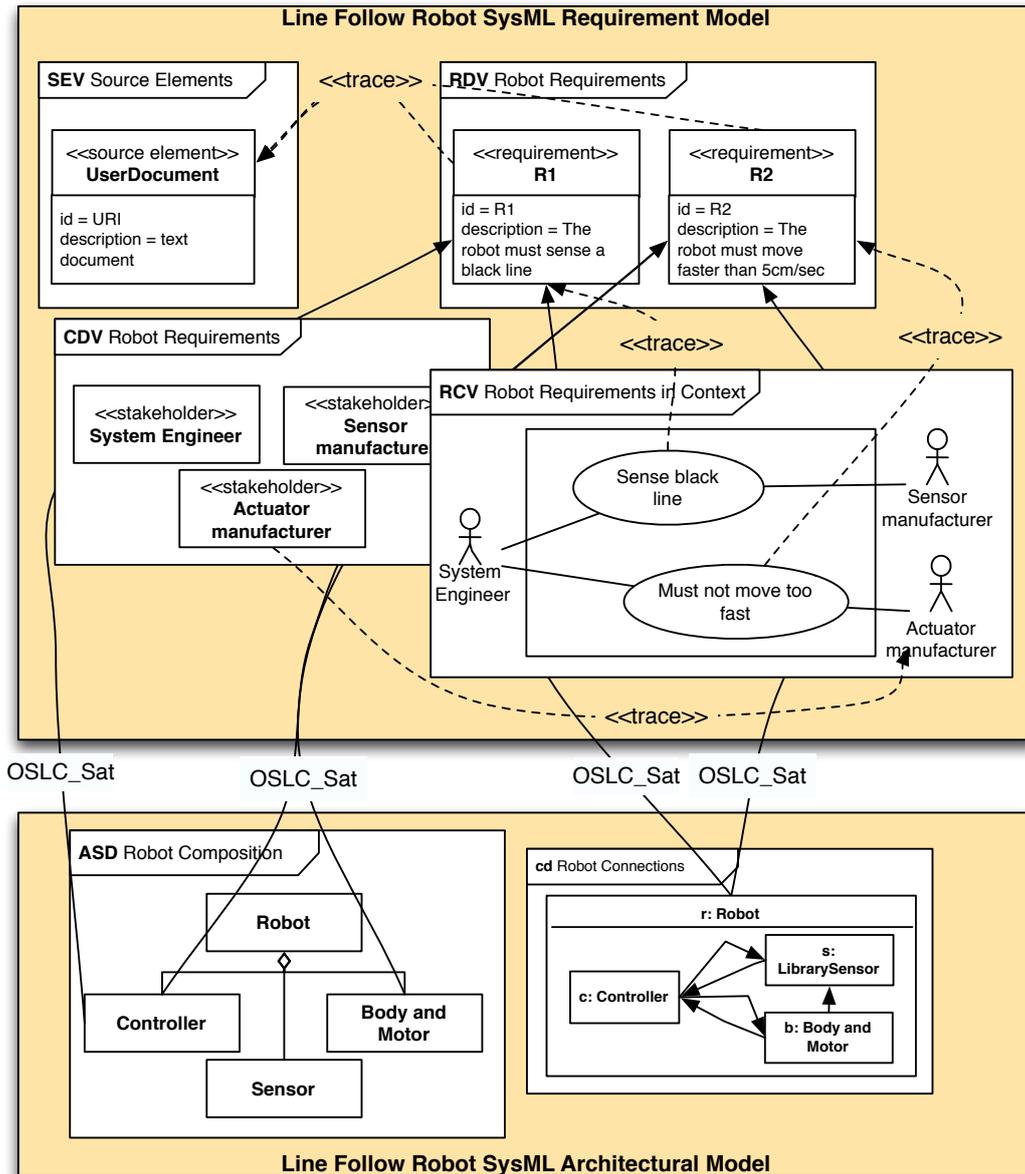


Figure 2: SysML requirements and SysML architectural models – model overview

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 2 presents an example with two SysML models with trace relationships and OSLC links between the models.

URI, Excel and SysML Next, we 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. As above, SysML can then be used to define the architecture in a single SysML model. Trace links using OSLC may then be used

to link the source elements, rows of Excel documents (with internal tracing using unique identifiers referenced between sheets), and architectural elements of the SysML architectural model. Figure 3 presents an example with URI, Excel and SysML models and OSLC links between the artefacts.

Excel and SysML The final approach defined here 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. OSLC links may trace between elements of the Excel requirements and context views in SysML, and between the different requirement artefacts and the architectural model. Figure 4 presents an example with URI, Excel and two SysML models with OSLC links between the artefacts.

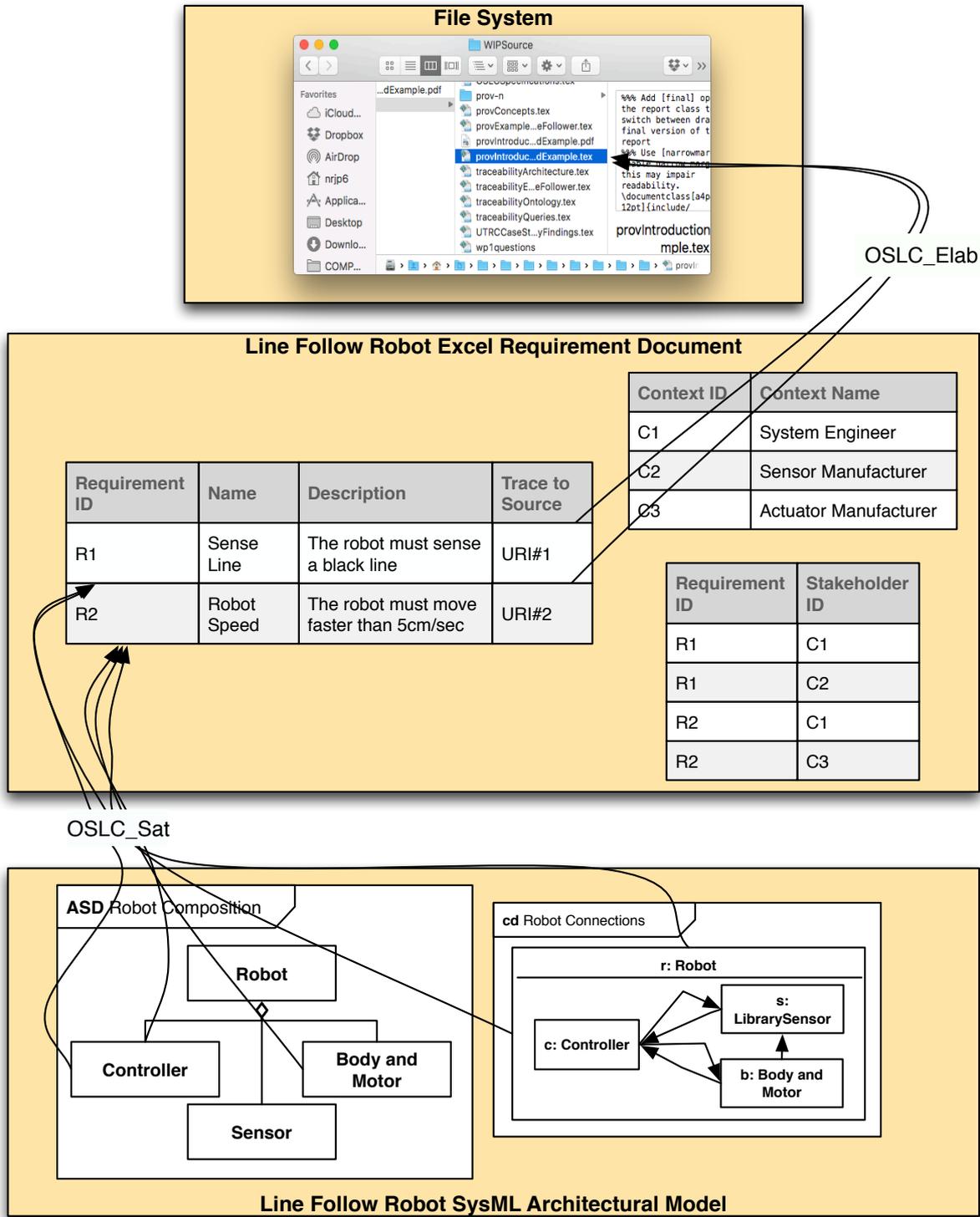


Figure 3: URI, Excel and SysML – model overview

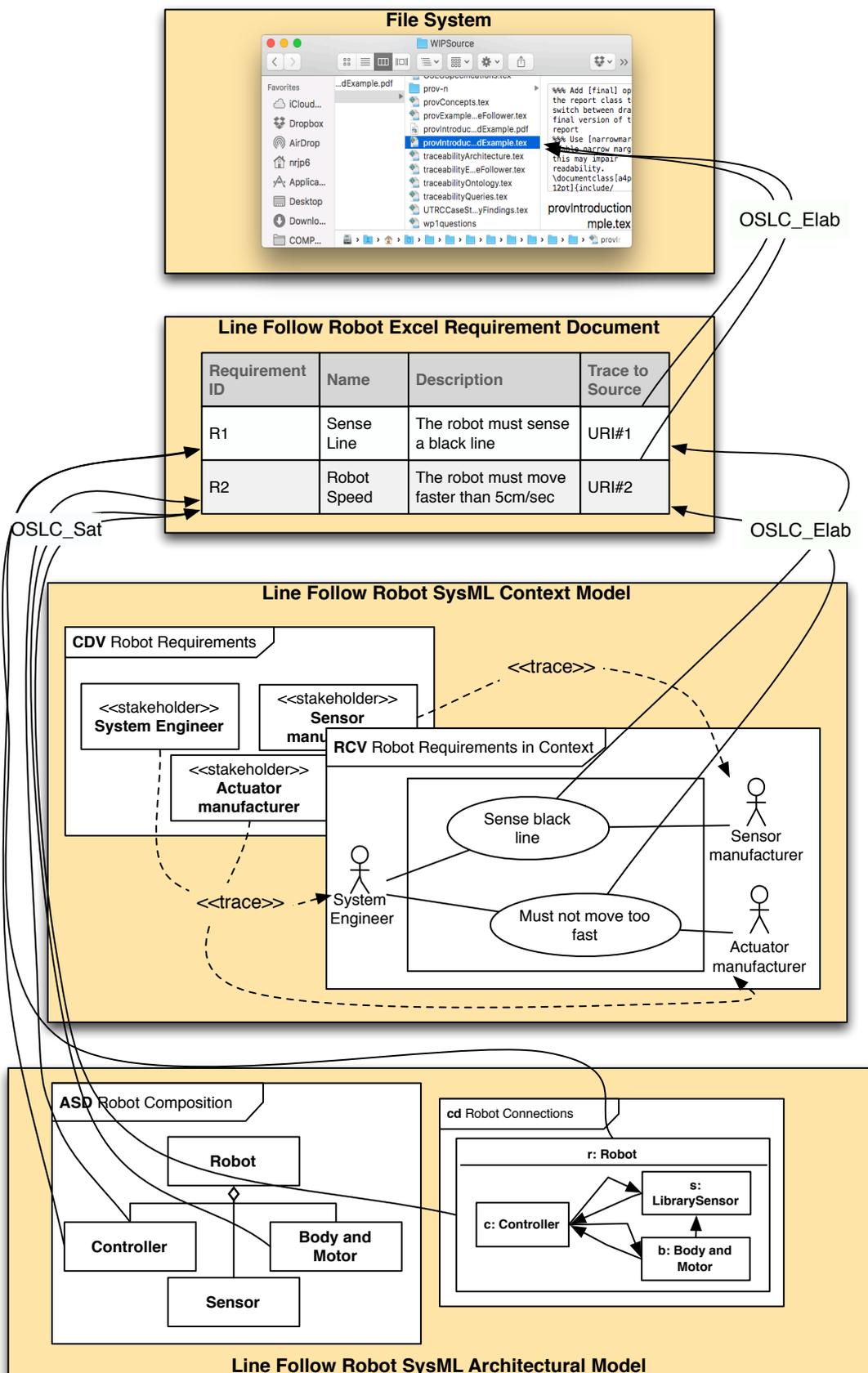


Figure 4: Excel and SysML – model overview

5 SysML and Multi-modelling

In the previous version of this deliverable (Deliverable D3.1a [FGPP15a]), we outlined guidelines for architectural modelling moving from a ‘holistic’ to a ‘design’ architecture using the INTO-SysML profile [APCB15]. In this document, we present extensions to the INTO-SysML profile, which allow us to model a CPS for co-simulation in a way which better reflects the architecture of the system.

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. As is defined in D2.1a [APCB15], a component is “a logical or conceptual unit of the system, corresponding to software or a physical entity”. This is a problem when we wish to visualise the model using 3D visualisations as provided by 20-sim with an FMU defined 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.

Finally, we wish to provide the ability to logically group EComponents⁷ into collections – allowing system engineers to better reflect the holistic architecture. We propose the addition of a new component type – ‘CComponent’ – corresponding to a component collection. The CComponent has no ports or behaviour itself, and exists purely to allow logical groupings.

The extension of the INTO-CPS SysML profile to incorporate these changes is reported in Deliverable D4.2c [BQ16]. 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) to illustrate the use of the new `ComponentKind` enumerations and the `CComponent` stereotype.

The INTO-SysML architecture structure diagram in Figure 5 gives examples of these extensions. The *System_Env* block is an `«EComponent»`, defined as an `Environment` FMU, the *3D_View* block is an `«EComponent»`, defined as an `Visualisation` FMU. Finally, the *Example_Robot* block is an `«EComponent»`, defined as an `Composition` of two FMUs.

The example has two Connection Diagrams, shown in Figures 6 and 7. The first – in Figure 6 – shows only 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 Connection Diagram in Figure 7 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 6.

⁷An EComponent, as defined in Deliverable D2.1a [APCB15] is an *Encapsulating Component*, corresponding to an element of a system that encapsulates a model.

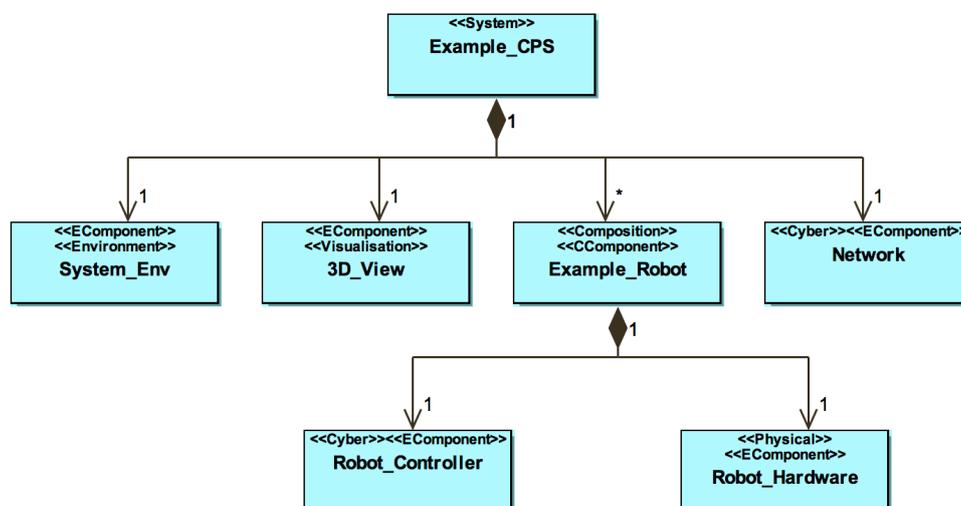


Figure 5: Example Architecture Structure Diagram of robot system

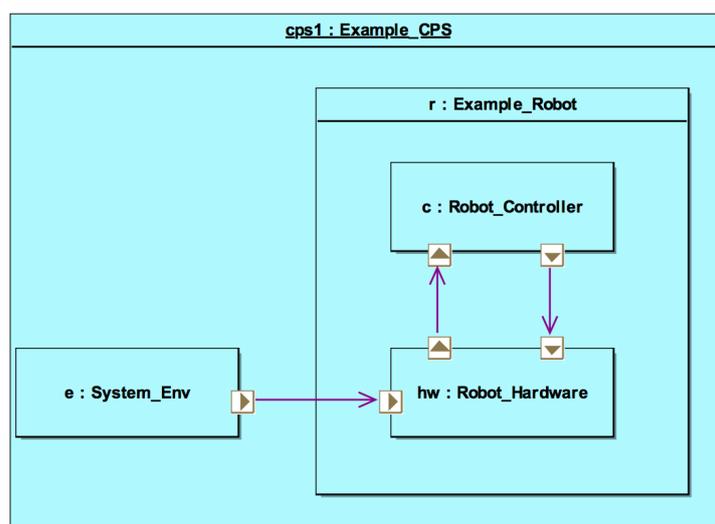


Figure 6: Connections Diagram for robot showing only system and environment connectors

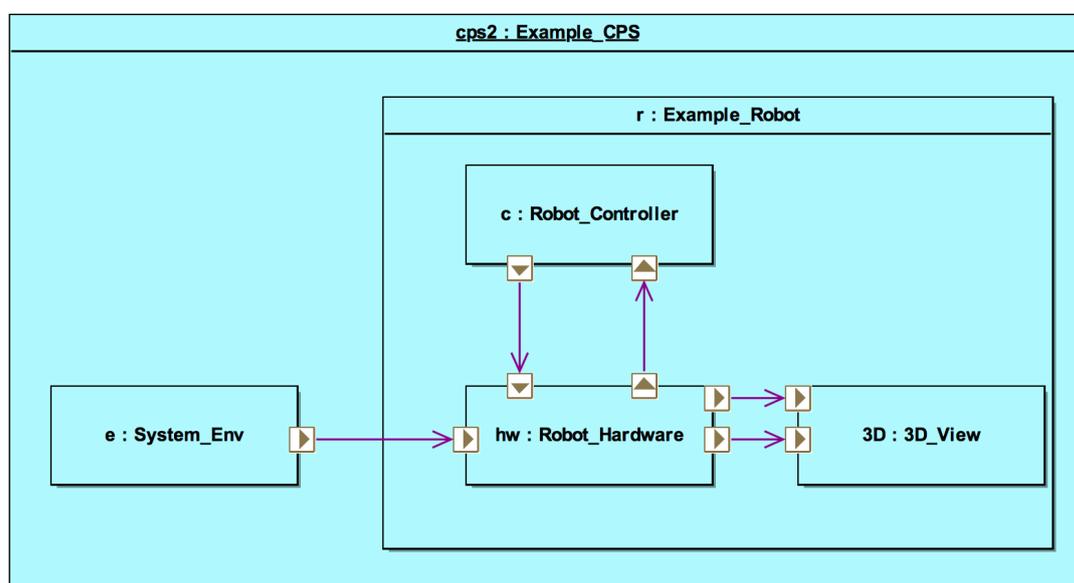


Figure 7: Connections Diagram for the robot system showing the system *and* visualisation components

6 Initial Multi-modelling

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]. The principles outlined in this section can be applied even if the SysML profile is not used, and multi-models are configured manually. In future, these guidelines will be expanded to include how and when to continuous-time (CT) formalisms in initial modelling.

6.1 The DE-first Approach

After carrying out requirements engineering (RE), as described in Section 4, and design architectural modelling in SysML, as described in Section 5, 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 website⁸.

⁸See <http://crescendotool.org/documentation/>

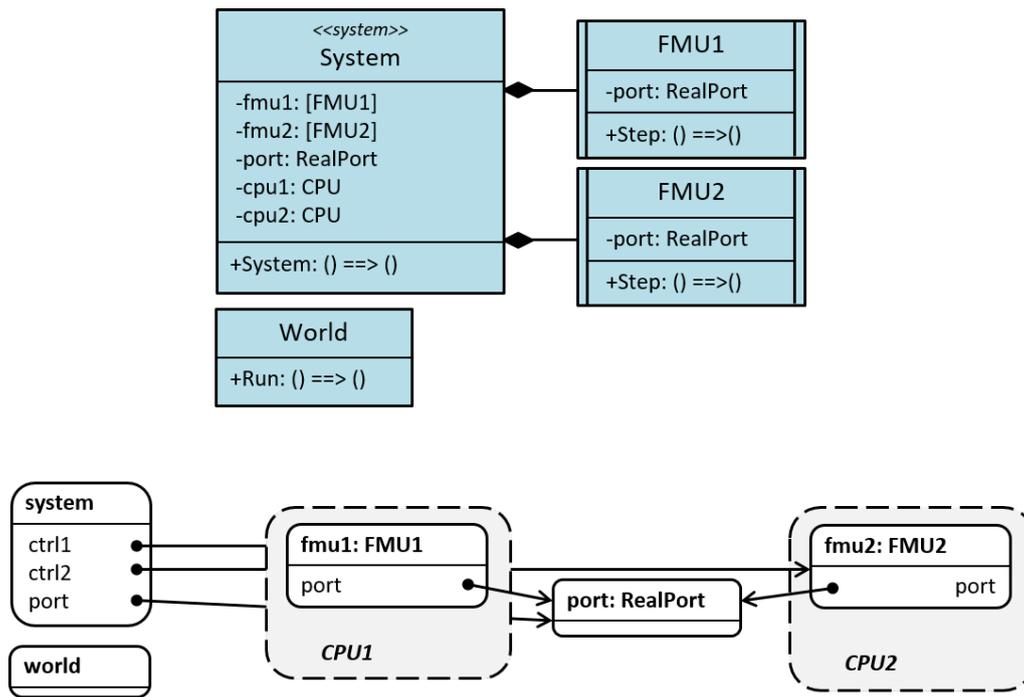


Figure 8: Class diagram showing two simplified FMU classes created within a single VDM-RT project, and an object diagram showing them being instantiated as a test.

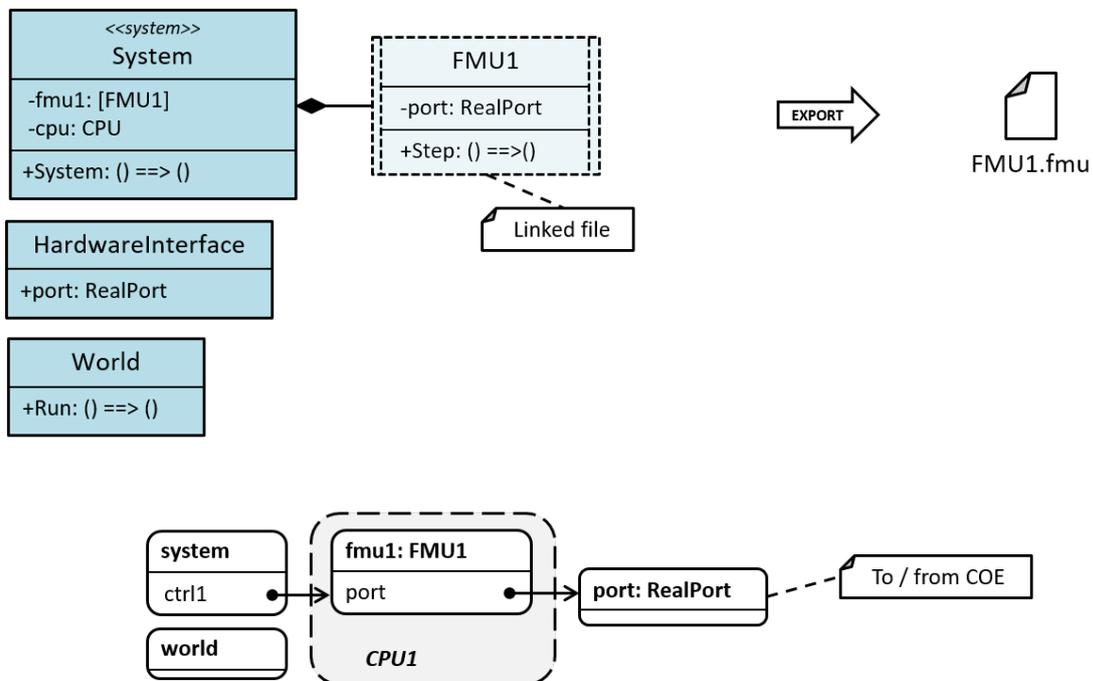


Figure 9: Class and object diagrams showing a linked class within its own project for FMU creation.

6.2 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:

- 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 8. 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 Section 7 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.

6.3 FMU Creation

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 9. 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 10). 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.

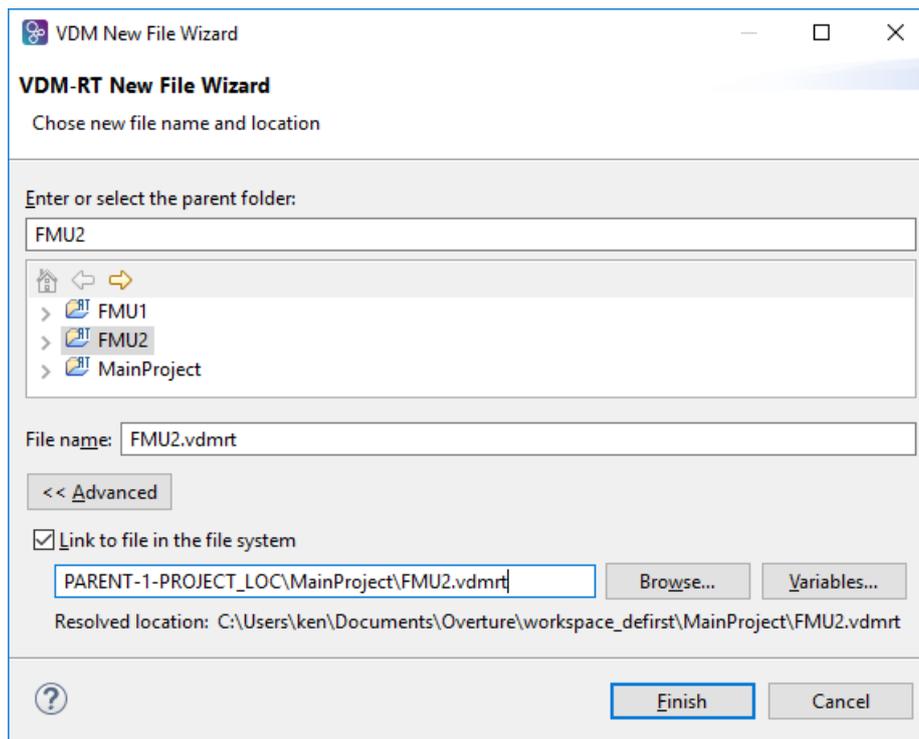


Figure 10: Linking files in the *New > Empty VDM-RT File* dialogue.

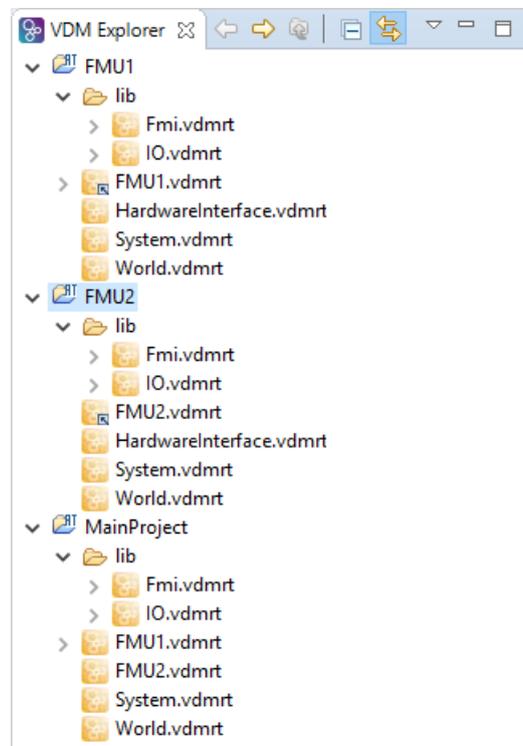


Figure 11: Project structure of an Overture workspace showing a main project and two projects used for generating FMUs from linked class files.

7 Modelling Networks 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 12. 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.

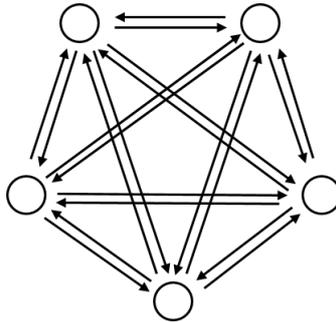


Figure 12: Topology of five controllers connected to each other

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 13.

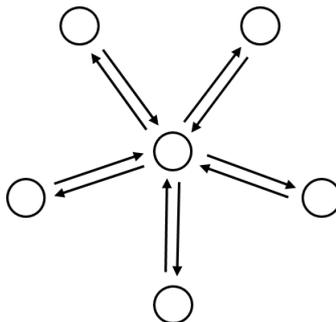


Figure 13: Topology of five controllers connected via a shared medium

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. A class listing for a reference implementation of the `Ether` class is given in Appendix B. An example multi-model built using this class, called *Case Study: Ether*, is available from the INTO-CPS Application. It is also described in the Examples Compendium, Deliverable D3.5 [PGP⁺16].

7.1 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 string 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.

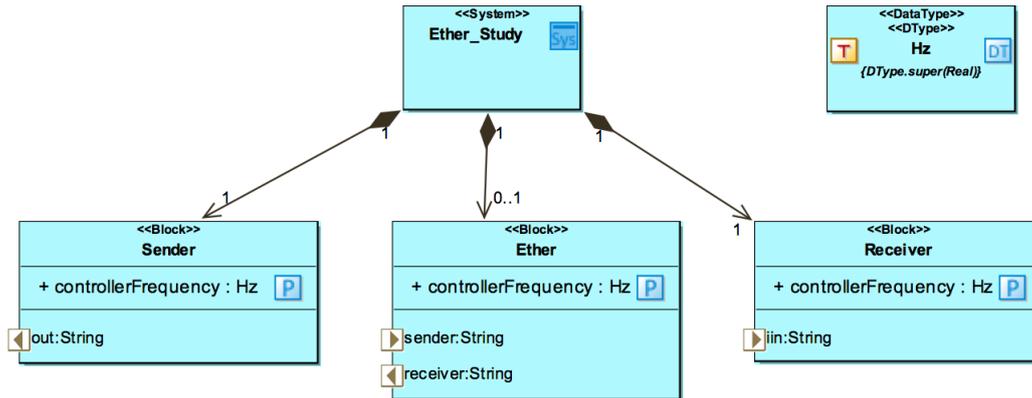
Note that currently (as of Overture 2.4.0), the `VDMUtil` library is called in the default scope, meaning that it does not know about custom types defined in the model. Therefore it is recommended to pack values in a tuple (as in the above example) for message passing, then convert to and from any custom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

7.2 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 14. 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)`.

Figure 14: *Case Study: Ether* example

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 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.

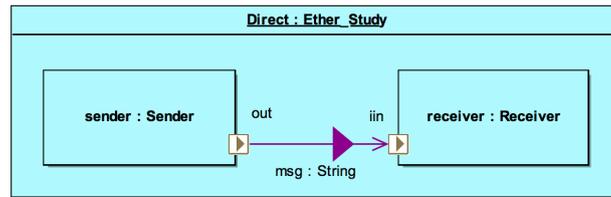
The *Case Study: Ether* example contains two multi-models that allow the sender and receiver to be connected directly (connection diagram shown in Figure 15(a)), or to be connected via the ether (connection diagram shown in Figure 15(b)). The description in the Examples Compendium, Deliverable D3.5 [PGP⁺16], 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.

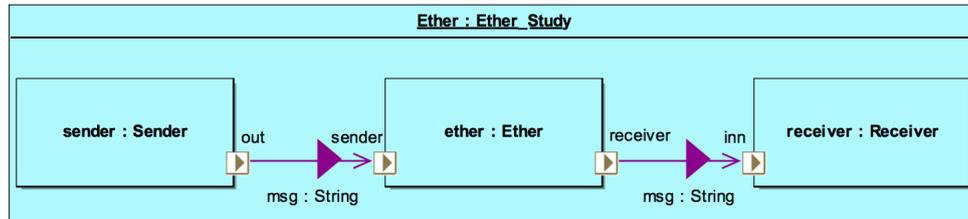
The ether defined in this example is intended to be generic enough that it can be used in other case studies that need a simple broadcast ether without guarantees of delivery. To use it, you can:

1. Import the *Ether* model from the *case-study_ether/Models* directory into Overture;
2. Update the `HardwareInterface`⁹ 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.

⁹A 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 15: Alternative multi-models in the *Case Study: Ether* example

7.3 Consequences of Using the Ether

The ether as presented above, and listed in Appendix B, 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 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.

7.4 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:

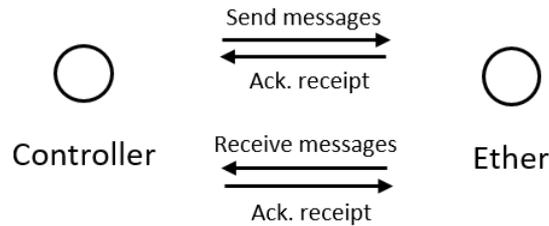


Figure 16: Topology of controller to *Ether* connection with dedicated channels for messages and acknowledgements

```

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 16.

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 Section 8). 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.

8 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 8.1); and (b) enable the tuning of DSE methods for given domains and systems of interest (Section 8.2).

8.1 Guidelines for Designing DSE in SysML

8.1.1 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 [FGPP16]) 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].

8.1.2 Requirements

We propose the use of a subset of the SoS-ACRE method detailed in Section 4 (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 17, 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 $X \text{ ms}^{-1}$
5. The robot sensor positions may be altered to achieve global goals

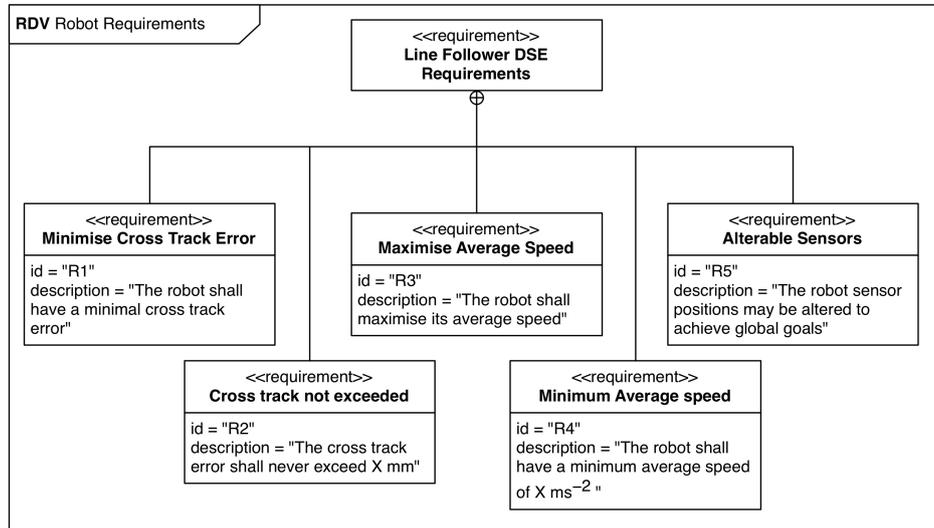


Figure 17: Subset of the Requirements Definition View for requirements of the Line Following Robot

8.1.3 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 18.

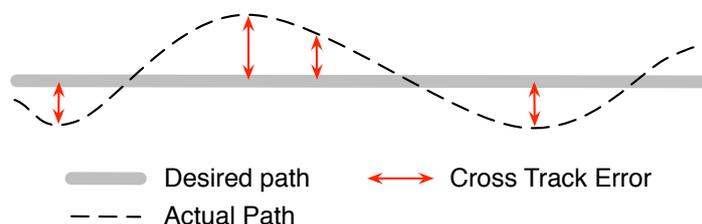


Figure 18: Cross track error at various points for a robot trying to follow a desired line

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 19.

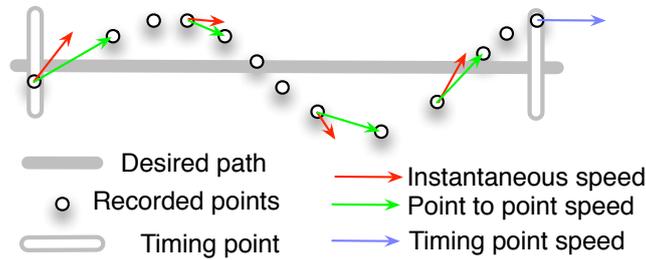


Figure 19: Cross track error at various points for a robot trying to follow a desired line

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.

8.1.4 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 20 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 21 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 22 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 23. 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

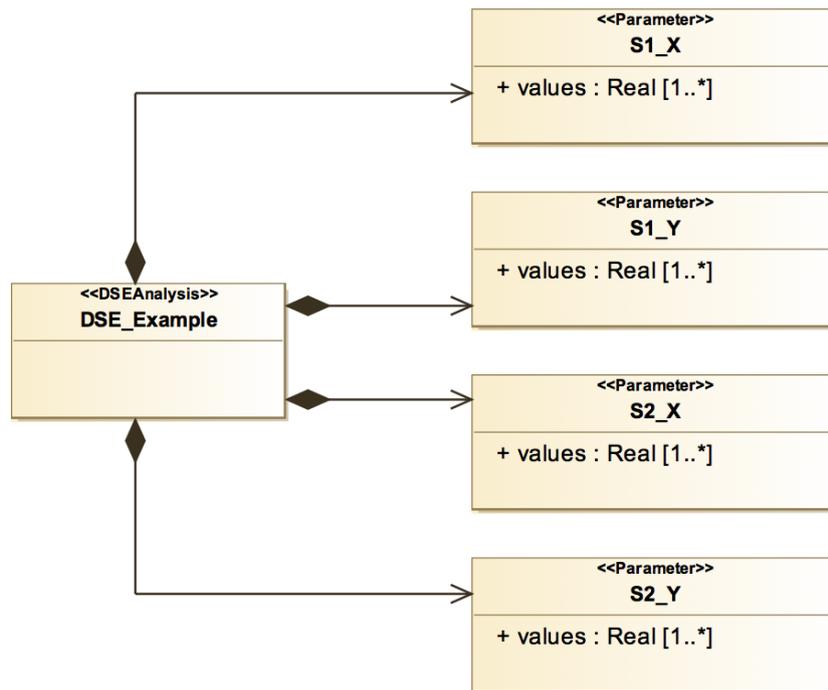


Figure 20: DSE-SysML Parameter Definition Diagram of Line Following Robot example

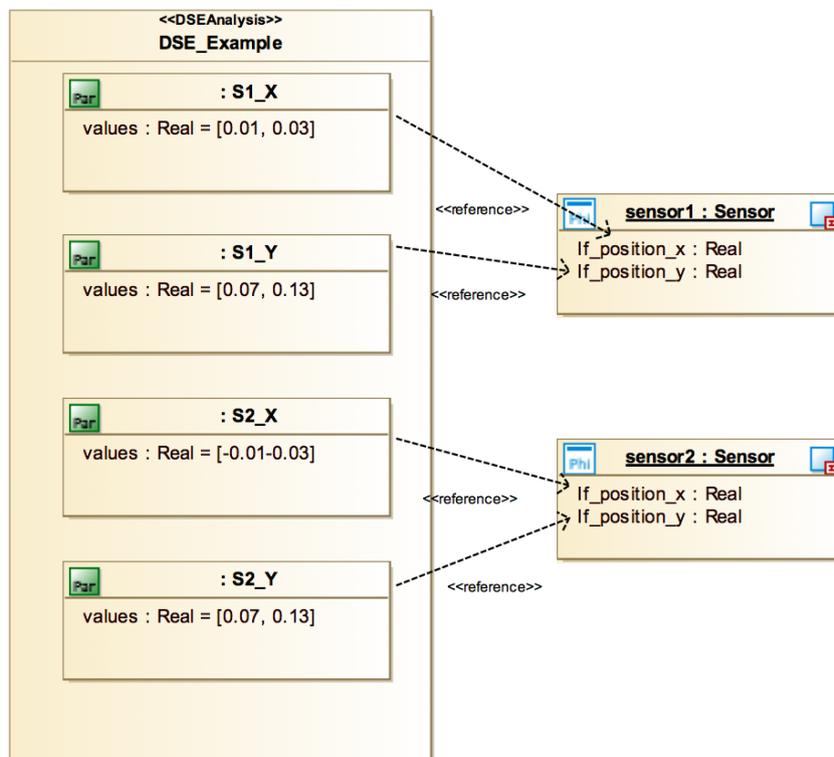


Figure 21: DSE-SysML Parameter Connection Diagram of Line Following Robot example

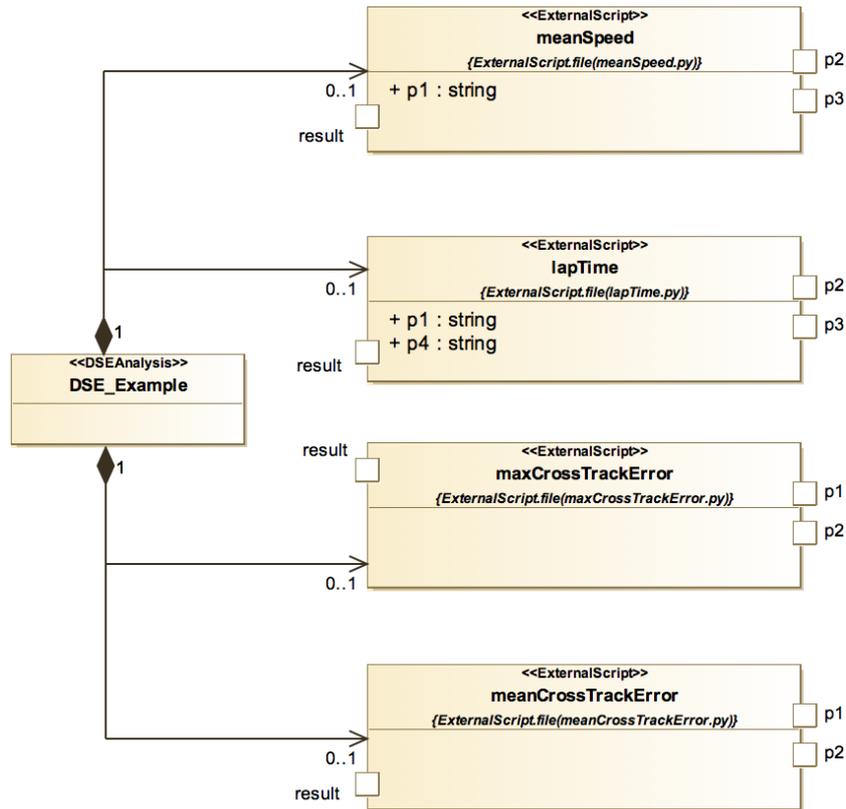


Figure 22: DSE-SysML Objective Definition Diagram of Line Following Robot example

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.

Ranking Finally, the DSE Ranking Diagram in Figure 24 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.

8.1.5 DSE script

These diagrams may then be translated to the json config format required by the DSE tool (see the INTO-CPS User Manual, Deliverable D4.2a [BLL⁺16] for more details). At present this is a manual process, however tool support in Modelio is in preparation and shall be available early in Year 3 of the project. This tool support will provide the automated generation of a skeleton configuration, specifying the parameters, objectives and ranking to use. This leaves the choice of DSE algorithm and simulation timing settings for an engineer to specify in the INTO-CPS application. Figure 25 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.

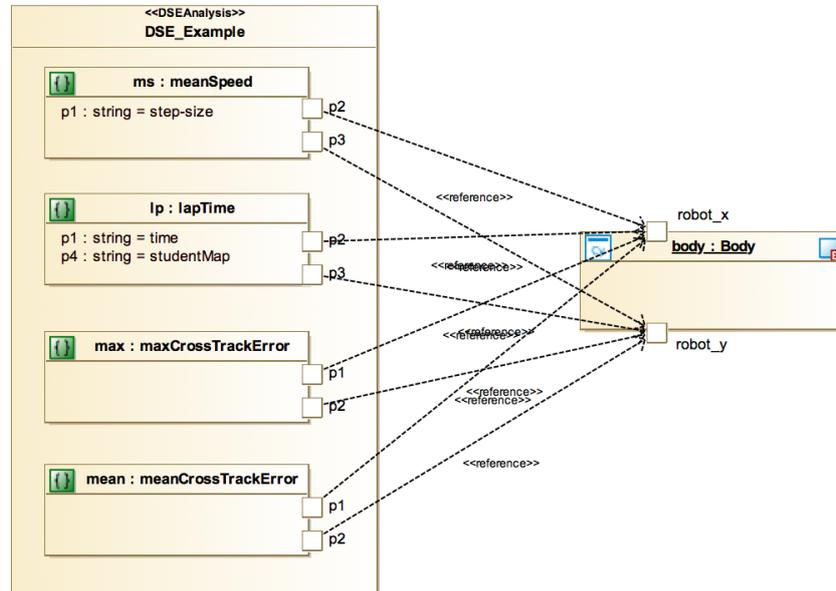


Figure 23: DSE-SysML Connection Objective Diagram of Line Following Robot example

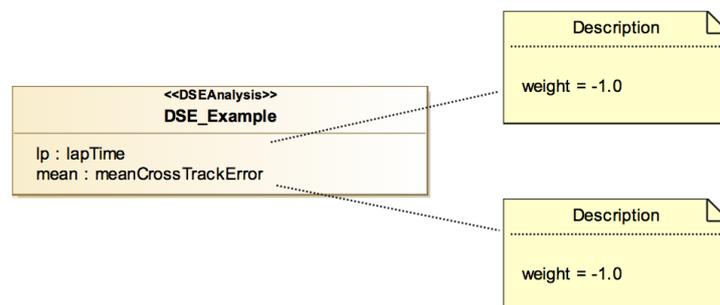


Figure 24: Example DSE-SysML Ranking Diagram of Line Following Robot example

```

{
  "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 25: A complete DSE configuration json file for the line follower robot example

8.1.6 DSE results

DSE is performed in the DSE tool (again, see the INTO-CPS User Manual, Deliverable D4.2a [BLL⁺16] 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 26 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 in 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.

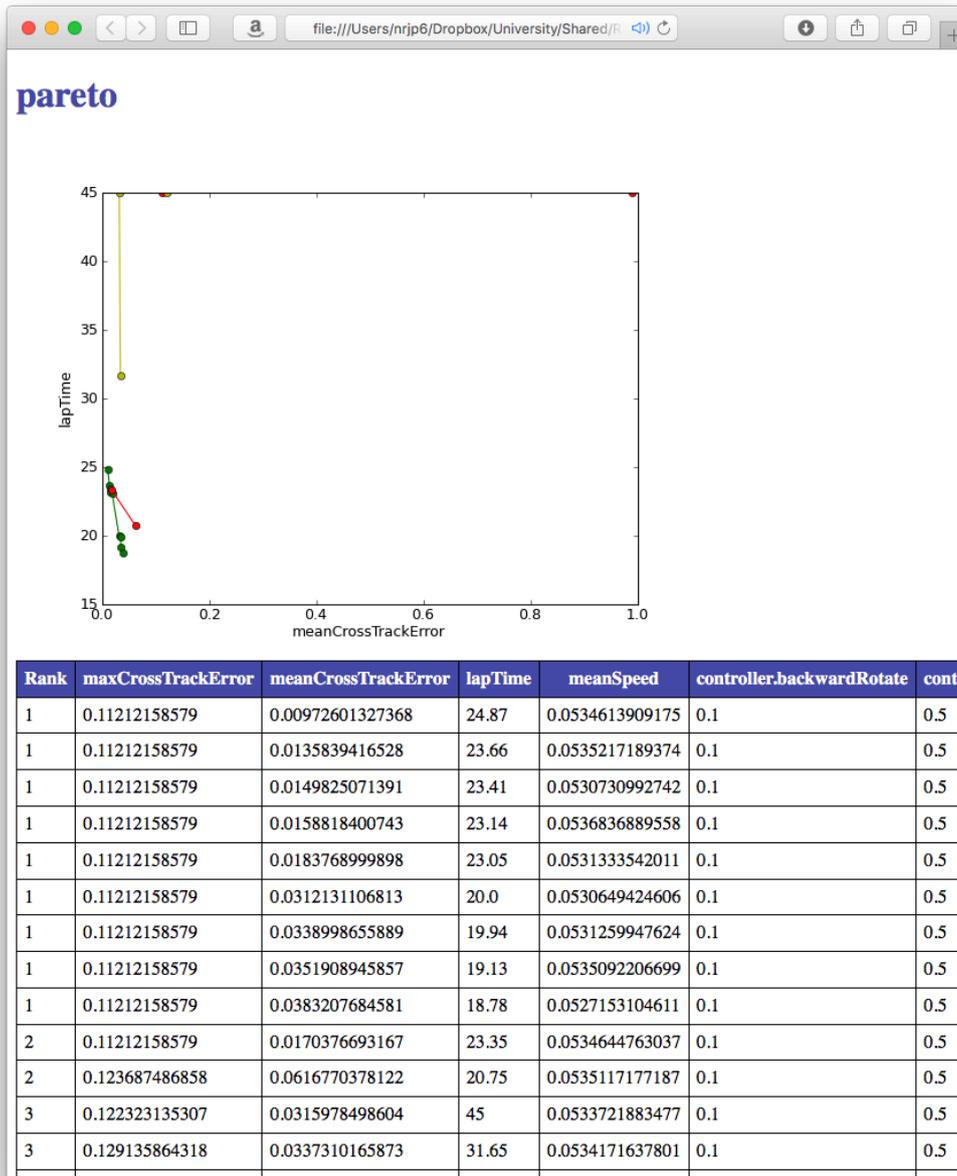


Figure 26: DSE results

8.2 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.

8.2.1 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 27 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.

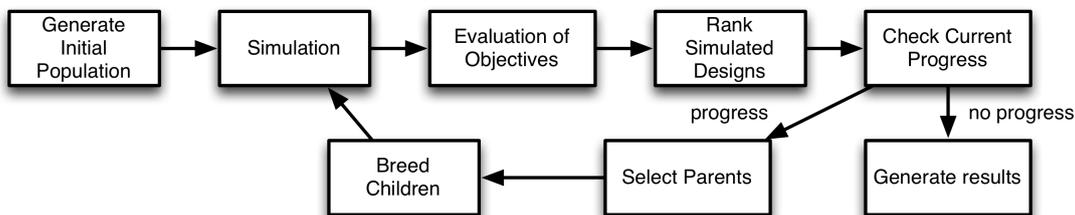


Figure 27: High-level process for DSE Genetic Algorithm

Generating initial population: Two methods for generating an initial population of designs are supported: randomly, or uniformly across the design space. Generating an initial 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 28(a) shows the fitness roulette by which how much a design solution in Figure 28(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:

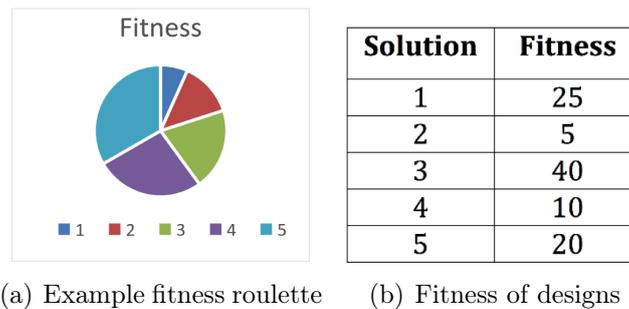


Figure 28: Genetic Algorithm fitness selection

Breeding children: After the parents are selected, the algorithm creates two new children using a process of crossover. Figure 29 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.

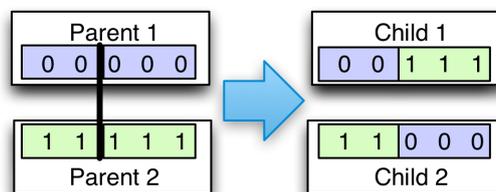


Figure 29: Depiction of genetic crossover

Checking current progress: Progression is determined by the change in the NDS on each iteration. It is possible to tune the number of iterations without progress before termination.

8.2.2 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.

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

8.2.3 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 run of each set of parameters. Figure 30, 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 31 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 32 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 5.

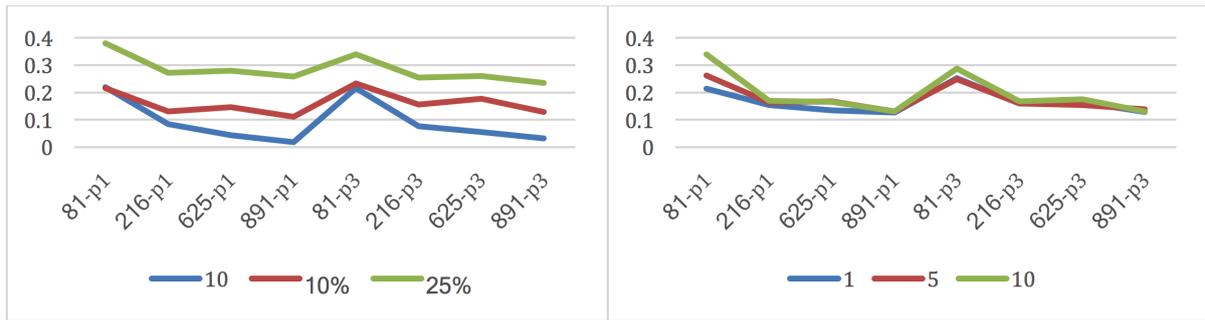


Figure 30: Cost of DSE, number of simulations run as proportion of the total design space

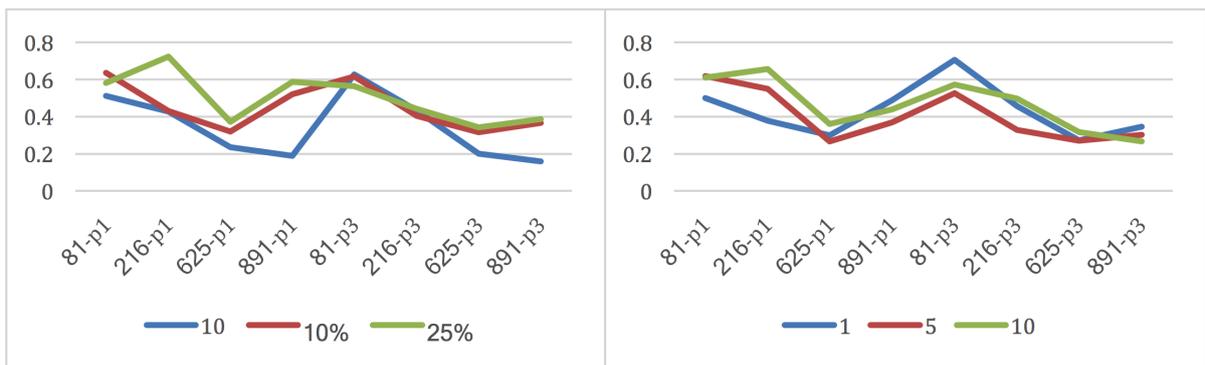


Figure 31: Accuracy of DSE, proportion of genetic NDS found in exhaustive NDS

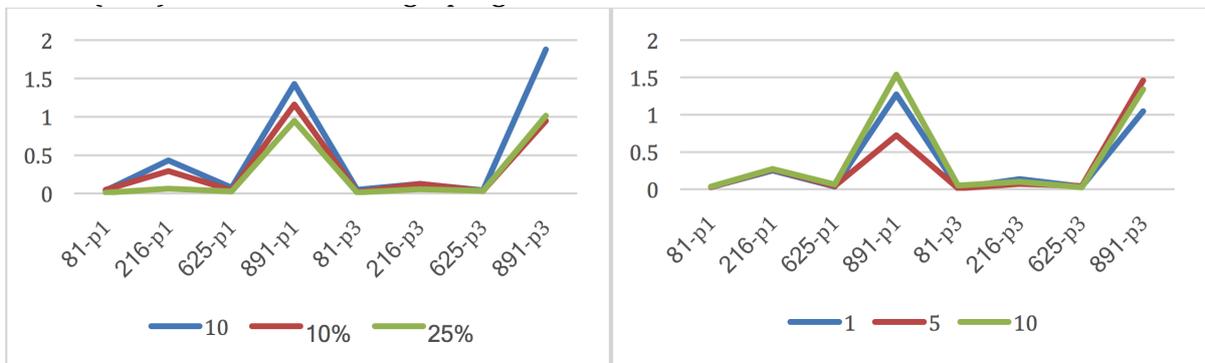


Figure 32: Gap between Genetic NDS and Exhaustive NDS

9 Forward Look

In the last year of the project, a final version of the document will be produced (Deliverable D3.3a) covering the entire engineering process covered by the INTO-CPS tool chain. We will provide new or extended guidance in the following areas:

Workflows Adoption of INTO-CPS into existing practice, including initial modelling using CT-first and other approaches; end-to-end workflows for the tool chain; and “getting started” material for all techniques enabled by INTO-CPS.

Requirements Engineering Links from requirements to modelling, and to analysis results. For example, guidelines for specifying requirements that influence objectives defined in DSE. This is alluded to in Section 8.1. We shall also consider how this relates to co-simulation and the existing work in test automation, as reported in Deliverable D5.1b [MPB15].

SysML Use of SysML for architectural modelling of CPSs in combination with the INTO-CPS DSE and test automation SysML profiles.

Design Space Exploration DSE in the cloud, for example, how does the number of parallel simulations affect the cost, accuracy and time to complete a DSE; and architectural DSE (how to describe architectural changes/change points, how to present results, how should search algorithms work with architectural DSE).

Traceability and Provenance How to make use of the traceability and provenance features of the tool chain using examples from the line-follower pilot study.

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.
- [BLL⁺16] 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, D4.2a, December 2016.
- [Blo14] Torsten Blochwitz. Functional Mock-up Interface for Model Exchange and Co-Simulation. <https://www.fmi-standard.org/downloads>, July 2014.
- [BQ16] Etienne Brosse and Imran Quadri. SysML and FMI in INTO-CPS. Technical report, INTO-CPS Deliverable, D4.2c, December 2016.
- [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.
- [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.
- [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.
- [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.
- [FGPP16] 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.

- [GF94] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference 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.
- [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.
- [MG13] Luc Moreau and Paul Groth. PROV-Overview. Technical report, World Wide Web Consortium, 2013.
- [Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.
- [MPB15] Oliver Möller, Adrian Pop, and Jörg Brauer. Distributed Testing and Simulation Network. Technical report, INTO-CPS Deliverable, D5.1b, December 2015.
- [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.
- [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™). 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.

A 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]).

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-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.

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.

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.

B Ether Class Listing

This section gives a listing for the `Ether` class described in Section 7. An example multi-model built using this class, called *Case Study: Ether*, is available from the INTO-CPS Application and is described in the Examples Compendium, Deliverable D3.5 [PGP⁺16].

Listing 1: Ether class

```

class Ether

types

public Id = seq of char

instance variables

-- thread period
private period: nat := 1E9;

-- access shared variables
incoming: map Id to StringPort;
outboxes: map Id to seq of seq of char;
outgoing: map Id to StringPort;
connects: set of (Id * Id);
inv forall mk_(i,o) in set connects &
  (i in set dom incoming and
   o in set dom outgoing);

operations

-- constructor for Ether
public Ether: nat1 * map Id to StringPort * map Id to StringPort *
  set of (Id * Id) ==> Ether
Ether(p,ins,outs,c) == (
  period := frequency_to_period(p);
  incoming := ins;
  outgoing := outs;
  connects := c;
  outboxes := {id |-> [] | id in set dom outs}
)
pre forall mk_(i,o) in set c &
  (i in set dom ins and
   o in set dom outs);

-- constructor for Ether (100Hz)
public Ether: map Id to StringPort * map Id to StringPort *
  set of (Id * Id) ==> Ether
Ether(ins,outs,c) ==
  Ether(100,ins,outs,c)
pre forall mk_(i,o) in set c &
  (i in set dom ins and
   o in set dom outs);

operations

-- send counter out and increase counter
private Step: () ==>()
Step() == cycles(2)
(
  -- gather inputs in outboxes
  for all mk_(i,o) in set connects do (
    let x = incoming(i).getValue() in (
      outboxes(o) := outboxes(o) ^ [x];
    )
  );
  -- flush outboxes
  for all i in set dom outboxes do (
    if len outboxes(i) > 1 then (

```

```
    outgoing(i).setValue(
      seq_of_seq_of_char2seq_of_char(outboxes(i))
    );

    -- debug
    IO'printf("ETHER.FMU: Passed %s to %s at %s\n",
      [outgoing(i).getValue(), i, time/1e9]);

    outboxes(i) := []
  )
);
);

private seq_of_seq_of_char2seq_of_char: seq of seq of char ==>
  seq of char
seq_of_seq_of_char2seq_of_char(ss) == (
  dcl outstr: seq of char := "[";
  for all i in set inds ss do
    if len ss(i) > 0 then outstr := outstr ^ ss(i) ^ [',''];
  outstr(len outstr) := ']';
  return outstr
);

-- run as a periodic thread
thread periodic(period, 0 ,0, 0)(Step);

functions

-- convert frequency to period in nanoseconds
private frequency_to_period: real -> nat
frequency_to_period(f) == floor 1E9/f

end Ether
```