Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Foundations of the SysML profile for CPS modelling

Deliverable Number: D2.2a

Version: 1.0

Date: December 2016

Public Document

http://into-cps.au.dk

Contributors:

Nuno Amálio, UY
Ana Cavalcanti, UY
Alvaro Miyazawa, UY
Richard Payne, UNEW
Jim Woodcock, UY

Editors:

Jim Woodcock, UY

Reviewers:

Luis Couto, UTRC
Carl Gamble, UNEW
Bernhard Thiele, LIU

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
Softeam	ST		

Document History

Ver	Date	Author	Description
0.1	15-06-2016	Jim Woodcock	Initial version with writing plan
0.2	31-10-2016	Jim Woodcock	Draft for internal review
0.3	05-12-2016	Jim Woodcock	Draft, including comments from internal reviewers
1.0	12-12-2016	Jim Woodcock	Final version for external review

Abstract

Deliverable 2.2a report on work carried out in Task 2.1 in Year 2 of the INTO-CPS project. The objective in this is to give semantics to SysML to enable it to be used as the entry-level modelling notation for the INTO-CPS tool chain. This entails extracting structural information from SysML models and carrying out some healthiness checks. We report on our approach to verify the healthiness and well-formedness of an architectural design, expressed using a profile of SysML. Our checks guarantee the conformity of component connectors and the absence of algebraic loops, necessary for co-simulation convergence. The checks are carried out using a combination of theorem proving and model-checking using the Isabelle/HOL proof assistant and the FDR3 refinement model checker.

We instantiate our approach by applying it to the engineering of mobile and autonomous robot applications; current practice suffers from costly iterations of trial and error, with hardware and environment in the loop. We propose the adoption of an approach to simulation and co-simulation, where designs, and (co-)simulations are traceable and amenable to verification. In this approach, designs are composed of several constituent models whose relationship is defined using our INTO-CPS SysML profile. Our approach supports automatic generation of simulations, and validation and verification beyond what can be achieved with simulation.

Contents

Task Objectives

There were two objectives for Task 2.1 in Year 2:

1. To give semantics to SysML as a CPS process that captures the graph dependencies in the Internal Block Diagram (IBD).
2. To define and implement a technique to check loops in the cycle of dependencies.

We have established an approach to verify both healthiness and well-formedness of an architectural design, expressed using a profile of SysML, as a prelude to FMI co-simulation. We check the conformity of component connectors and the absence of algebraic loops, necessary for co-simulation convergence. Verification of these properties involves theorem proving and model-checking using Fragmenta, a formal theory for representing typed visual models, with its mechanisation in the Isabelle/HOL proof assistant, and the CSP process algebra and its FDR3 model-checker.

We have instantiated this approach using RoboChart, a modelling notation for robotic controllers.

1 Introduction

Cyber-physical systems (CPSs) are designed to actively engage with the physical world in which they reside. They tend to be heterogeneous: their subsystems tackle a wide variety of domains (such as, mechanical, hydraulic, analogue, and a plethora of software domains) that mix phenomena of both continuous and discrete nature, typical of physical and software systems, respectively.

CPSs are often handled modularly to tackle both heterogeneity and complexity. To effectively separate concerns, the global model of the system is decomposed into subsystems, each typically focussed on a particular phenomenon or domain and tackled by the most appropriate modelling technique. Simulation, the standard validation technique of CPSs, is often carried out modularly also, using co-simulation [40], the coupling of subsystem simulations. This constitutes the backdrop of the industrial Functional Mockup Interface (FMI) standard [8, 7] for co-simulation of components built using distinct modelling tools.

We present an approach to formally verify the well-formedness and healthiness of SysML CPS architectural designs as a prelude to co-simulation. The designs are described using INTO-SysML [5], a profile for multi-modelling and FMI co-simulation. The well-formedness checks verify that designs comply with all the required constraints of the INTO-SysML meta-model; this includes *connector conformity*, which checks the adequacy of the connections between SysML blocks (denoting components) with respect to the types of the ports being wired. The healthiness checks concern detection of *algebraic loops*, a feedback loop resulting in instantaneous cyclic dependencies; this is relevant because a desirable property of co-simulation, which often reduces to coupling of simulators, is *convergence* (where numerical simulations approximates the solution), which is dependent on the structure of the subsystems and cannot be guaranteed if this structure contains algebraic loops [40, 11]. The work presented here demonstrates the capabilities of our verification workbench for modelling languages and engineering theories, which rests on FRAGMENTA [4], a theory to formally represent designs of visual modelling languages, and its accompanying mechanisation in the Isabelle proof assistant [49], and the CSP process algebra [34] with its accompanying FDR3 refinement-checker [29].

Our contributions are as follows:

1. A novel SysML profile for architectural modelling of CPSs that tackles heterogeneity by providing support for multi-modelling and co-simulation in compliance with the FMI standard.
2. An approach to statically check the adequacy of a SysML architectural model for co-simulation, supporting connector conformity and algebraic loops detection, by using a theorem prover and a model-checker.
3. A prototyping environment for FRAGMENTA [5], a mathematical theory to represent typed visual models, based on the proof assistant Isabelle/HOL that enables model verification and transformation.
4. A CSP-based solution to the detection of algebraic loops, which is based on a novel approach to represent graphs in CSP.
5. An evaluation of approaches to the detection of algebraic loops.

The report has three main sections. Sect. 2 gives some background on the formalisms that we use. Sect 3 presents our approach to represent architectural designs in INTO-SysML, highlighting verification of well-formedness, and our approach for representing directed graphs in CSP and detecting algebraic loops through refinement model checking. Finally, Sect. 4 instanti-

ates our approach using RoboChart, a modelling technique for robotic controllers.

2 Preliminaries

2.1 SysML and the INTO-SysML profile

The Systems Modeling Language (SysML) [50] is a general-purpose graphical notation for systems engineering applications, defined as an extension of a subset of the Unified Modeling Language (UML) [30]. This extension is achieved by using UML's profile mechanism, which provides a generic technique for customising UML models for particular domains and platforms. A profile creates a conservative extension of UML, refining its semantics in a consistent fashion.

There are commercial and open-source SysML tools. These include IBM's Rational Rhapsody Designer,[1] Atego's Modeler,[2] and Modeliosoft's Modelio.[3] They support model-based engineering and have been used in complex systems.

A SysML block is a structural element that represents a general system component, describing functional, physical, or human behaviour. The SysML Block Definition Diagram (BDD) shows how blocks are assembled into architectures; it is analogous to a UML Class Diagram. A BDD represents how the system is composed from its blocks using associations and other composition relations.

A SysML Internal Block Diagram (IBD) allows a designer to refine a block's structure; it is analogous to UML's Composite Structure Diagram, which shows the internal structure of a class. In an IBD, parts are assembled to define how they collaborate to realise the block's overall behaviour.

The INTO-SysML profile [2] customises SysML for architectural modelling for FMI co-simulation. It embraces the many themes of the INTO-CPS project: tool interoperability, semantic heterogeneity, holistic modelling, and co-simulation, and provides the modelling gateway for the INTO-CPS approach.

[1] See `sysml.tools/review-rhapsody-developer/`.
[2] See `http://www.atego.com/de/products/atego-modeler/`.
[3] See `www.modelio.org/`.

It specialises blocks to represent different types of components, that is, constituent models, of a CPS, constituting the building blocks of a hierarchical description of a CPS architecture. A component is a logical or conceptual unit of the system: software or a physical entity.

INTO-SysML comprises two diagram types, Architecture Structure Diagrams (ADs) and Connection Diagrams (CDs), specialising SysML BDDs and IBDs.

In our examples, the constituent models are written in Robochart (described in Section 2.5) and Simulink [46]. The latter, developed by MathWorks, is a graphical programming environment for modelling, simulating, and analysing multi-domain dynamic systems. Its primary interface is a graphical block diagramming tool and a customisable set of block libraries.

2.2 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) [25] is an industry standard for collaborative simulation of separately developed models of CPS components: co-simulation. The key idea is that, if a real product is assembled from components interacting in complex ways, each obeying physical laws (electronic, hydraulic, mechanical), then a virtual product can be created from models of those physical laws and a model of their control systems. Models in these different engineering fields are heterogeneous: they use different notations and simulation tools.

The purpose of FMI is to support this heterogeneous modelling and simulation of CPSs. FMI is used in a number of different industry sectors, including automotive, energy, aerospace, and real-time systems integration. There is a formal development process for the standard, and many tools now support FMI.

An FMI co-simulation consists of Functional Mock-up Units (FMUs), which are models encapsulated in wrappers, interconnected through inputs and outputs. FMUs are slaves: their collective simulations are orchestrated by a master algorithm. Each FMU simulation is divided into steps with barrier synchronisations for data exchange; between these steps, the FMUs are simulated independently.

A master algorithm communicates with FMUs through the FMI API, whose most important functions are those to exchange data, `fmi2Set` and `fmi2Get`, and that to command the execution of a simulation step, `fmi2DoStep`. The FMI standard does not specify master algorithms, but places restrictions on

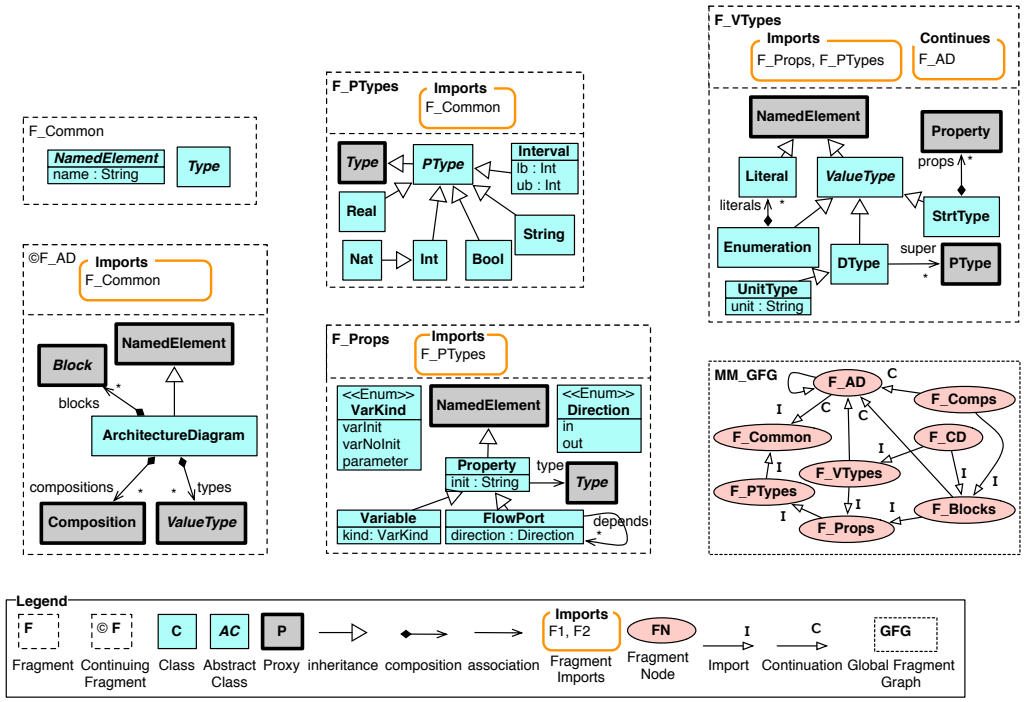

Figure 1: Some fragments of metamodel of INTO-SysML.

the use of the API functions that constrain how a master algorithm can be defined and how an FMU may respond. Formal semantics for FMI can be found in [10, 14].

2.3 Fragmenta and its Isabelle Mechanisation

FRAGMENTA [4] is a graph-based theory to represent modularised (or fragmented) typed class models. It is based on the algebraic theory of graphs and their morphisms [20]. FRAGMENTA represents designs of visual modelling languages whose structure is defined by class metamodels (domain-specific languages (DSLs)) and their resulting instance models. Its overall models are a collection of sub-models called *fragments*. Type and instance models are related through morphisms. A major novelty lies in FRAGMENTA's *proxies*: representatives of other nodes. A fragment is as a graph that supports proxies.

Figure 1 portrays five fragments and one global fragment graph (GFG) from INTO-SysML's metamodel. It highlights how fragments build up on other

fragments either in a bottom-up (through imports) or top-down (through continues) fashion and the use of proxies for inter-fragment referencing. Importing is bottom-up because the bigger fragments are built from smaller ones. Continuation is top-down because it starts by specifying a summary model (or a skeleton) with points of continuation, represented as proxies, to be continued by other fragments. Fragment `F_PTypes` is an increment to `F_Common`; node `Type` from `F_Common` is referenced through the proxy with same name; likewise in `F_Props` with proxy `NamedElement`. Fragment `F_AD`, which summarises the metamodel of the INTO-SysML architecture diagrams (ADs), is a continuing fragment; `F_VTypes` continues `F_AD`.

FRAGMENTA proposes two composition operators: (a) union composition ($\cup_F$) merges fragments without resolving the proxies, and (b) colimit composition (based on category theory) joins fragments by resolving the proxies.

Further details are given in [4].

The theory introduces the following sets (see [4] for details):

- *Fr*, of well-formed fragments, requires that: (a) the underlying graph is well-formed, (b) the inheritance hierarchy is acyclic, (c) the source of composition relations has multiplicity 1 or $0\,..\,1$ and (d) proxies do not inherit[4]. All fragments in Fig. 1 are members of *Fr*.
- *GFGr*, of acyclic GFGs: `MM_GFG` (Fig. 1) $\in$ *GFGr*.
- *Mdl*, of all well-formed models, requires that the model's fragments are disjoint. A model M is a tuple (GFG, fd), made up of a $GFG \in GFGr$ and a total function $fd : Ns_{GFG} \rightarrow Fr$ mapping *GFG* nodes to fragments. INTO-SysML's metamodel, partially described in Fig. 1, is a member of *Mdl*.
- $F_1 \rightarrow_F F_2$, of all well-formed fragment morphisms, which impose the required graph commuting constraints in the setting of fragments.
- *FrTy*, of well-formed typed fragments $FT = (F, TF, ty)$; F and TF are instance and type fragments, respectively: $F, TF \in Fr$, and $ty \in F \rightarrow_F TF$.
- *FrTyConf*, of conformant fragments, a subset of *FrTy*, imposes the following constraints on instances: abstract nodes may not have direct instances, containments are not shared, instance relations satisfy metamodel multiplicities, and instances of containments form a forest.

[4] A local check that ensures the compositionality of FRAGMENTA's union operator.

- *MdlTy*, of all well-formed typed models $MT = (M, TM, ty)$, where M and TM are instance and type models ($M, TM \in Mdl$), and the type morphism is conformant: $(UFs\ M, UFs\ TM, ty) \in FrTyConf$, where UFs makes a single fragment out of the union of model fragments.

FRAGMENTA's Isabelle mechanisation[5] provides a verification and transformation environment for metamodel designs. One can check that:

- The individual fragments of both model and metamodel are locally consistent and well-formed. For fragment `F_Common` of Fig. 1, for instance, we need to prove $\vdash F\_Common \in Fr$[6]; likewise for the remaining fragments.
- GFGs are well-formed also. For GFG of Fig. 1: $\vdash MM\_GFG \in GFGr$.
- Overall models and metamodels are also consistent and well-formed. For the metamodel *INTO_SysML* of Fig. 1: $\vdash INTO\_SysML \in Mdl$.
- Instance models conform to the constraints imposed by the type model.

Section 3.1 gives further details on INTO-SysML inside FRAGMENTA/Isabelle.

2.4 CSP and FDR3

The CSP process algebra [34] describes communicating processes and interaction-driven computations. CSP's major structuring concept, the process, represents a self-contained component made up of interfaces to enable interaction with a multitude of environments.

Processes communicate by transmitting information along channels. A CSP channel carries messages and has, therefore, a set of associated events, corresponding to all messages that may be transmitted. Process expressions are built using a number of operators, which include:

- Event prefixing, expressed as $e \longrightarrow P$, describes a process that expects event e and then behaves as process P.
- External choice, $P_1 \Box P_2$, gives the environment the choice of events offered by P_1 and P_2. Replicated external choice $\Box\ i : \mathbb{N} \bullet P(i)$ composes the resulting processes using external choice.

[5] Available at `https://github.com/namalio/Fragmenta`

[6] Such membership predicates are represented in Isabelle as functions to booleans and they capture the well-formedness constraints associated with a FRAGMENTA set.

- Internal choice, $P_1 \sqcap P_2$, non-deterministically chooses to act like P_1 or P_2.
- Parallel composition, $P_1 \underset{A}{\parallel} P_2$, executes the two processes in parallel synchronising on the set of events A.

FDR3 [29] is CSP's refinement checker. It checks refinement according to CSP's denotational models (including traces, failures and failures-divergences), and other properties, including deadlock and livelock-freedom, and determinism.

2.5 RoboChart

RoboChart [47] is a diagrammatic notation tailored to the design of robotic systems. RoboChart models use Harel-style statecharts [31], but crucially, also include constructs that embed concepts of robotic applications. They are used to structure models for abstraction and reuse. Moreover, the statecharts use an action language that is both timed and probabilistic.

A RoboChart design centres around a robotic platform and its controllers. Communication between controllers can be either synchronous or asynchronous, but communication between state machines inside a controller is synchronous. The operations in a state machine may be given interface contracts using preconditions and postconditions, may be further defined by other state machines, or may come from a domain-specific API formalised separately. The formal semantics of RoboChart is mechanised in CSP [47].

As a simple example, we consider a Rover robot inspired by that in [32]. It is an autonomous vehicle equipped to detect certain chemicals. It randomly traverses a designated area, sniffing its path with its onboard analysis equipment. If it detects a chemical source, it turns on a light and drops a flag as a marker.

A robotic system is specified in RoboChart by a module, where a robotic platform is connected to one or more controllers. A robotic platform is modelled by variables, events, and operations that represent built-in hardware facilities. The ChemicalDetector module for our example is shown in Figure 2; it has a robotic platform Rover and controllers DetectAndFlagC and LightC.

The named boxes on the border of Rover declare events. The lightOn and lightOff events request that the built-in light is switched one way or the other.

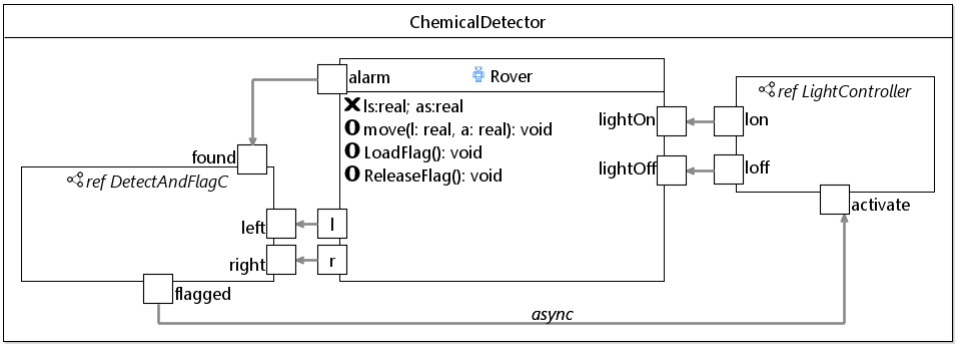

Figure 2: RoboChart module

The sensor events l and r record the detection of a wall on one side or the other. The alarm represents the detection of a chemical source by the built-in sensor.

The variables ls and as of Rover record its linear and angular speeds. The move(l,a) operation commands the Rover to move with speeds l and a; it is part of the RoboChart API. The operations LoadFlag() and ReleaseFlag(), on the other hand, are not in the RoboChart API, since they are particular to this example. They are declared, but not further defined.

The two controllers DetectAndFlagC and LightC define the behaviour of Rover. DetectAndFlagC controls the events left, right, found, and flagged (see the bordered boxes), thereby interacting with Rover and LightC. These events are associated with l, r, and alarm of Rover, and activate of LightC, as indicated by the arrows, whose directions define information flow. So, when Rover finds a chemical, it sends an alarm to DetectAndFlagC. LightC uses events lon, loff, and activate to communicate with Rover and DetectAndFlagC.

In Sect 4, we propose a way of considering RoboChart models in the context of a variation of the INTO-SysML profile. In Section 4.4, we give a semantics based on the FMI API for the co-simulation specified in SysML.

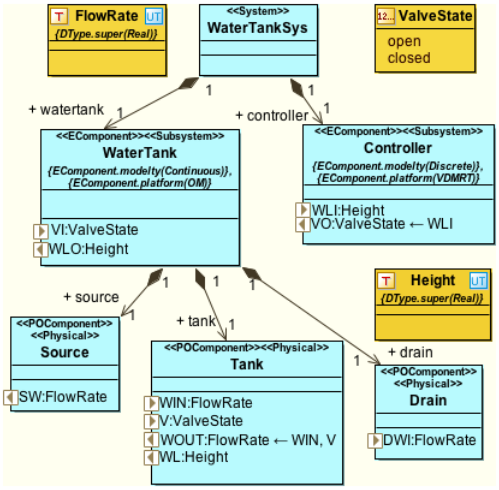

Figure 3: INTO-SysML AD

3 Checking SysML Models for Co-Simulation

3.1 Architectural Modelling in INTO-SysML

The INTO-SysML profile [5] introduces specialisations of SysML blocks (known as stereotypes) to represent different types of CPS components, constituting the building blocks that enable a hierarchical description of the CPS architectures that we need. A component is a logical or conceptual unit of the system, corresponding to a software or a physical entity. The profile's component constructs comprise: `System`, `EComponent` (encapsulating component) and `POComponent` (part-of component). A system is decomposed into subsystems (represented as `EComponent`s), which are further decomposed into `POComponent`s. `EComponent`s and `POComponent`s may be further classified as `Subsystem` (a collection of inner components), `Cyber` (an atomic unit that inhabits the digital or logical world) or `Physical` (an atom unit pertaining to the physical world). Furthermore, their characterising phenomena may be classified as `discrete` or `continuous`.

Currently, INTO-SysML comprises two diagram types: *architecture diagrams* (ADs) and *connections diagrams* (CDs), specialising SysML block definition and internal block definition diagrams, respectively. They are as follows:

- ADs (see Fig. 3) describe a decomposition in terms of the types of system components and their relations. They emphasise multi-modelling:

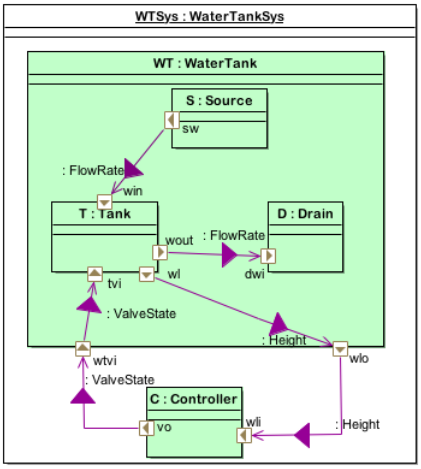

Figure 4: INTO-SysML CD

certain components encapsulate a model built using some modelling tool (such as VDM/RT [44], 20-sim [38] or Open Modelica [27]).

- CDs (see Fig. 4) are AD instances. They convey the configuration of the system's components, highlighting flow and connectedness.

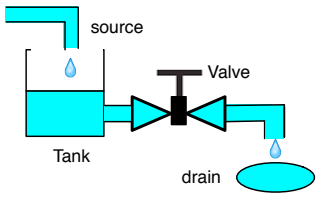

Figure 5: Water Tanks system.

The water tanks system, sketched in Fig. 5, is used as one of our running examples. A source of water fills a tank whose water outflow is controlled by a valve; when the valve is open the water flows into the drain. The valve, managed by a software controller, is opened or closed depending on the tank's water level. We also consider a variant of this system with the drain connected to the tank.

Fig. 3 portrays the architectural model of water tanks, built using INTO-SysML's Modelio implementation[7]. The AD is as follows:

- The overall system (`WaterTankSys`) comprises two major subsystems, `WaterTank` and `Controller`, which are `EComponents`: they encapsulate separate models. `WaterTank` deals with `continuous` phenomena modelled in Open Modelica. `Controller` is `discrete` and modelled in VDM/RT.
- `WaterTank` has three physical sub-components: `Source`, `Tank` and

[7]Available from `http://forge.modelio.org/projects/intocps-modelio34`.

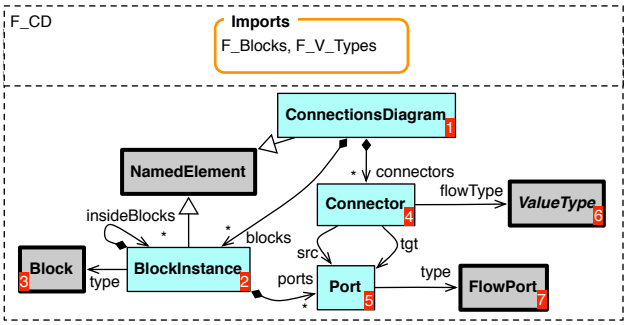

Figure 6: Metamodel of INTO-SysML CDs

`Drain`: they are `POComponent`s (part-of of a subsystem).

- Enumeration `ValveState` captures the valve's state. Unit types `FlowRate` and `Height`, built from reals, deal with flow rates and water levels.
- Each component provides flow ports to enable communication and the flow of material; the outputs indicate the inputs ports on which they depend.

CD of Fig. 4 describes the system instance (`WTSys`) composed of one `WaterTank` (`WT`) with its sub-components. The `Controller` instance (`C`) receives the water height from `WT` and, in return, directs `WT` to open or close the valve.

3.2 Well-formedness Checking using Fragmenta/Isabelle

INTO-SysML's metamodel and the instance model of Fig. 5 are represented according to the FRAGMENTA [4] graph-based theory in the Isabelle proof assistant. FRAGMENTA constructs overall models as a collection of fragmented sub-models. It can be used to represent designs of visual modelling languages whose structure is defined by class metamodels. Currently, there is a mechanisation of FRAGMENTA in the Isabelle proof assistant[8]. The fragmented metamodels of the INTO-SysML profile are available from [5].

Fig. 7 gives the FRAGMENTA representation of CD in Fig. 4 and Fig 6 is the metamodel of INTO-SysML CDs; the correspondence from CD to metamodel, entailed by the type morphism, is represented as labels with numbers. In Fig. 7, the proxies reference elements from the AD of Fig. 3, nodes labelled

[8]This is available from: `https://github.com/namalio/Fragmenta`

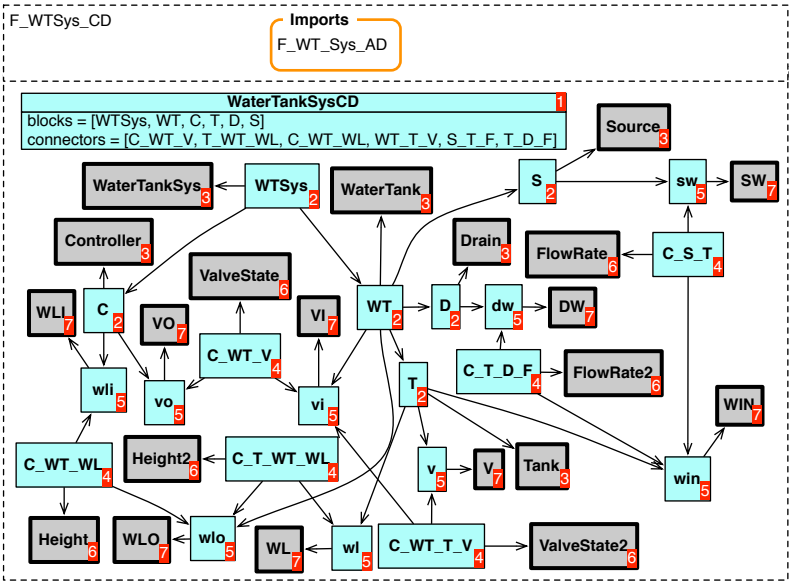

Figure 7: INTO-SysML CD of water tanks system (Fig. 4) in FRAGMENTA

4 correspond to the connectors of the CD, and those labelled 5 correspond to ports.

From the FRAGMENTA base sets of Section 2.3, we build a set of well-formed INTO-SysML models $INTO\_Mdls$, catering for all profile-specific invariants. The AD invariants are: (i) there is one system block, (ii) `EComponent`s are not nested, and (iii) `POComponent`s are contained by `EComponent`s. The CD invariants are: (iv) instance ports are correctly typed with respect to AD flow ports, (v) connection's flow types correspond to types consistent with the ports being connected (conformity of connectors), and (vi) the CD satisfies multiplicities imposed by AD.

The model M_WTs is subject to the following checks:

- Fragments of AD and CD are well-formed: $\vdash F\_AD \in Fr$, $\vdash F\_CD \in Fr$.
- The model's GFG is well-formed: $\vdash GFG\_WTs \in GFGr$.
- Overall model is well formed: $\vdash M\_WTs \in Mdl$.
- M_WTs must be a valid INTO-SysML model. Given a type morphism ty (illustrated in Fig 7), we prove: $\vdash (M\_WTs, ty) \in INTO\_Mdls$, which entails $\vdash (M\_WTs, INTO\_SysML, ty) \in MdlTy$.

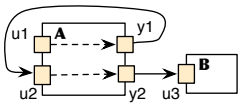

Figure 8: A topology without algebraic loops

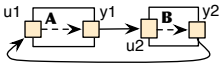

Figure 9: A topology with an algebraic loop

These are the checks required for any INTO-SysML model.

3.3 Fragmenta/Isabelle as a Transformation Engine

To enable usage of model-checkers, FRAGMENTA/Isabelle is used as a transformation engine in the algebraic loops check, which finds cycles in a topology of dependencies in instantaneous component communication.

Fig. 8 portrays a self-cycle component that is algebraic loop free. Output `y1` of `A` is connected to `A`'s input `u2`, but this does not entail an algebraic loop. The topology in Fig. 9, on the other hand, contains an algebraic loop.

Finding algebraic loops equates to detecting cycles in a directed graph describing port dependancy relations. An edge between two ports indicates that the target node is instantaneously dependent on the source. This constitutes a *port dependancy graph* (PDG), illustrated in Fig. 11, which portrays a PDG with an algebraic loop corresponding to the variant INTO-SysML model that connects the `Drain` to the `Tank` (`dwo` to `win`).

The Isabelle mechanisation introduces a function that produces a PDG from a INTO-SysML model. The resulting PDG, obtained from the ports and connections of CD and the internal dependancies between output and input port types of AD, is derived from both metamodel and model. Another function takes the PDG and produces the CSPm specification to be checked in FDR3.

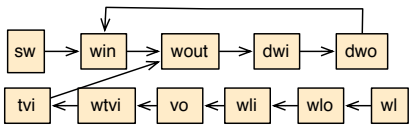

Figure 10: PDG derived from *M_WTs*

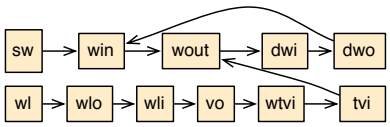

Figure 11: PDG with algebraic loop

3.4 Algebraic Loop Verification using CSP

We represent graphs in CSP and detect cycles on them via a traces-refinement check executed in the FDR3 refinement checker[9]. This is illustrated with the PDG of Fig. 12, containing labelled edges and numbers assigned to nodes with outgoing edges. We represent edges as CSP channels and nodes as CSP processes. Representation is oriented towards the edges of the graph as that suits CSP's communication model based on channels. Hence, each edge of the graph is represented as a CSP channel:

channel *sw_win*, *win_wout*, *wout_dwi*, *tvi_wout*,
wtvi_tvi, *vo_wtvi*, *wli_vo*, . . .

The overall graph is a CSP process constructed from sub-processes representing each node. The node processes are an external choice of CSP prefixed expressions for each edge that starts at the node. They offer the events on the corresponding channel and then behave as the process at the end of the edge. An edge to a sink node (no outgoing edges) results in a transition to

[9] https://www.cs.ox.ac.uk/projects/fdr/

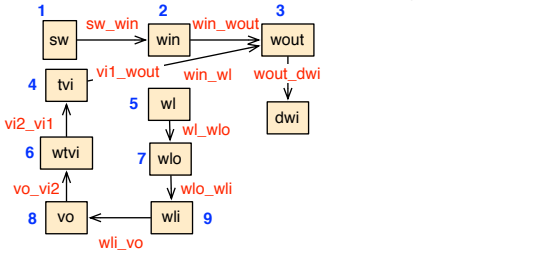

Figure 12: A PDG with labelled nodes and edges

$SKIP$. The main process is the external choice of all sub-processes. The process for PDG of Fig. 12 is:

$$
\begin{array}{l}
PortDependancyGraph = \\
\quad \text{let } P(1) = sw\_win \longrightarrow P(2) \\
\qquad P(2) = win\_wout \longrightarrow P(3) \\
\qquad P(3) = wout\_dwi \longrightarrow SKIP \\
\qquad \vdots \\
\quad \text{within } \Box\, i : 1..9 \bullet P(i)
\end{array}
$$

Cycles are detected through traces refinement. The abstract CSP process to be refined defines all finite paths whose size is at most the number of edges in the graph (those that can be built by combining the graph's edges):

$$
\begin{array}{l}
edges = \{sw\_win, win\_wout, wout\_dwi, tvi\_wout, wtvi\_tvi, vo\_wtvi, \ldots\} \\
Limited = \\
\quad \text{let } Limited0(E, n) = \\
\qquad\quad \text{if } n > 0 \text{ then } \Box\, e : E \bullet e \longrightarrow Limited0(E, n-1) \sqcap SKIP \text{ else } STOP \\
\quad \text{within } Limited0(edges, 9)
\end{array}
$$

The traces refinement check to be executed in FDR3 is then:

$$\textbf{assert } Limited \sqsubseteq_{\mathrm{T}} PortDependancyGraph$$

All counter-examples are cycles. The function *toCSP* of FRAGMENTA/Isabelle (Section 3.3) yields CSP specifications as outlined above. For the PDG of Fig. 12, FDR3 gives no counter-examples; for Fig. 11 FDR3 yields one counter-example.

3.5 Evaluation

FDR3 is a tool based on model-checking, a verification technique whose drawback is scalability. We compare our CSP approach to detect algebraic loops (Section 3.4) against one approach based on Alloy [35] and one graph algorithm [36], to gauge scalability.

3.6 Experimental Setup

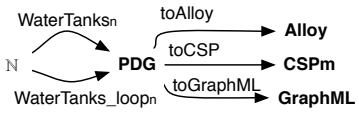

Figure 13: The experiment's generation functions

Scalability is evaluated against growing PDGs based on the water tanks running example (Fig. 5). We keep adding tanks to a base water tanks systems to produce systems of cascading water tanks having two versions: one with algebraic loops (drain is connected to first tank) and one without (as per Fig. 5).

The generation of files to execute in either FDR3, Alloy 4[10] or the implementation of Johnson's algorithm in JGraphT [36][11], involves Isabelle functions that yield PDGs given the number of tanks. We then define functions from PDGs to the abstract syntax of CSP (as per Section 3.4), Alloy (see below) and Graph ML[12] as per diagram of Fig. 13[13]. This relies on Isabelle's code generation. Pretty printing and file outputting is done by an ML program that builds up on the Isabelle generated ML.

The graph checks and data collection were performed by a Java program that reads the files and calls either Alloy 4 (using the minisat SAT solver), FDR3 or JGraphT, executed on a MacBook Pro with a 2.5 GHz Intel core i7 processor and 16GB RAM memory. The resulting data was subject to a statistical analysis carried out in the R statistical package [53].

3.7 The Alloy Model

Alloy [35] is a declarative modeling language based on first-order logic with transitive closure. It is used for data modelling and provides an automatic bounded analysis of a model. Our Alloy model of PDGs is based on the signature *Port*:

```
abstract sig Port {tgt : set Port}{tgt ≠ this}
```

Above, we declare a set of `Port` instances (`abstract` says that `Port` has no instances of its own and that all its instances belong to its extensions (subsets)) with the relation `tgt` between `Ports` declared to be non-reflexive: the `tgt` of some `Port` cannot be itself (`this`).

[10] `http://alloy.mit.edu/alloy/download.html`.

[11] A Java library of graph algorithms `https://github.com/jgrapht/jgrapht`.

[12] A standard for graphs exchange that enables a direct representation of PDGs `http://graphml.graphdrawing.org/`.

[13] The Isabelle file that performs the generation, the actual generated files, and the Java code that runs the three approaches, can be found at `http://bit.ly/1WKTIC7`.

The actual nodes of the PDG of Fig. 12 extend `Port`:

```
one sig sw, win, wout, dwi, wl, wlo, wli, vo, wlvi, tvi
   extends Port  {}
```

Above, the nodes are singletons (constraint `one`) that subset `Port` (`extends`).

The following Alloy fact defines the edges of the graph:

```
fact {sw.tgt = win
   win.tgt = wout
   wout.tgt = dwi
   no dwi.tgt ... }
assert AcyclicTgt {no ^tgt & iden}
check AcyclicTgt for 10
```

Above, each edge is declared through relation `tgt`: `sw.tgt =win` says that there is an edge from `sw` to `win` (operator . is the relational image), `win.tgt =wout` says that there is an edge from `win` to `wout`, and `no dwi.tgt` says that `dwi` has no outgoing edges (set is empty).

Finally, we assert the acyclicity of the relation `tgt` representing the PDG and declare the command to check the assertion:

```
assert AcyclicTgt {no ^tgt & iden}
check AcyclicTgt for 10
```

Above, the assertion says that there can be no elements (operator `no`) in the set resulting from the intersection (operator `&`) of the relation's transitive closure (`^tgt`) with the identity relation (`iden`). The `check` command includes a scope declaration: the analysis should consider at most 10 PDG nodes.

3.8 Comparisons

The plots of Fig. 14 and 15 depict the data obtained from running the experiments. They display the number of nodes of the analysed graph in the abscissa and the duration of the check (in seconds) in the ordinate.

Fig. 14 shows that there is an overwhelming difference in favour of CSP against Alloy. CSP's maximum duration is $8.58s$, Alloy's is $652.59s$. The two approaches start to diverge with small to medium size graphs (number of nodes > 17). The p-value, obtained from the paired data plotted in Fig. 14

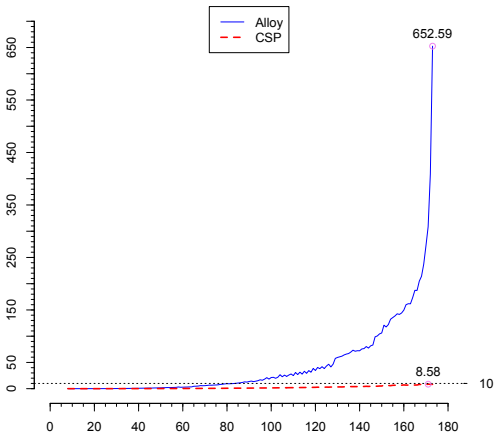

Figure 14: Alloy vs CSP

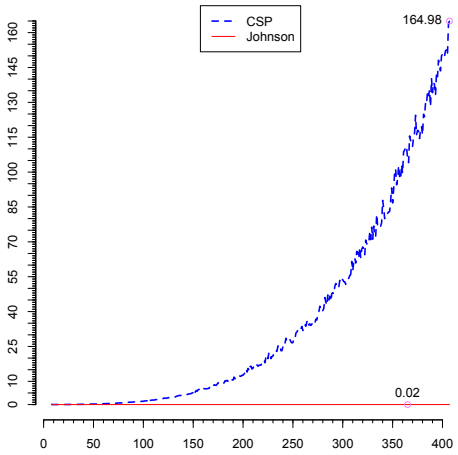

Figure 15: CSP vs Johnson algorithm

using the Wilcoxon statistical test[14], of $< 2.2^{-16}$ (< 0.001) indicates a very large difference. We derived estimates of functions that fit the data of both Alloy and CSP to yield estimates of time complexity: Alloy has complexity $O(Exp)$, whereas CSP has complexity $O(n^3)$ (n is number of nodes of the graph).

Fig. 15, on the other hand, shows that Johnson's algorithm performs substantially better than CSP. The former's maximum duration is $0.02s$, CSP's is $164.98s$. The p-value of $< 2.2^{-16}$ (< 0.001) signals a very large difference. The estimated function that fits the data endorses the algorithm's linearity

[14]It is a non-parametric test that compares the two sampled distributions without assuming that they follow the normal distribution.

claim.

3.9 Discussion

The work presented here statically checks an architectural design of a CPS in preparation for co-simulation. This is done at the high-level architectural design to provide early warnings of any issues so that the appropriate remedial action can be taken. It is a preliminary check (done before delving into the details of global co-simulation and local modelling and analysis of each component) to ensure that the models to be co-simulated are, among other things, free of connector inconformities and algebraic loops. These checks are performed using the Isabelle proof assistant and the FDR3 model-checker; both constitute an intimate part of our verification toolset.

We present a profile of SysML (originally defined in [5]), designed as a DSL, for architectural modelling of CPSs supporting multi-modelling and FMI co-simulation. The profile embodies an implicit systems decomposition paradigm driven by multi-modelling: the overall system architecture is a decomposition of subsystems (`E-component`s), encapsulating their own models, which are further decomposed into `POComponents` to give an account of the inner structure of each subsystem. The profile enables a holistic algebraic loop analysis that considers the inner details of each subsystem. Guidance on the definition of SysML models for multi-modelling is provided in [24], aiding CPS engineers in modelling a CPS architecture both holistically and in a decomposed form suitable for co-simulation.

The profile's design caters for FMI co-simulation. The `E-component` subsystems of the architecture result in FMI's Functional Mock-up Units (FMUs) to be co-simulated; FMUs are generated by the corresponding modelling framework.

The profile's DSL design was brought to life by Fragmenta and its accompanying Isabelle mechanisation. Fragmenta/Isabelle, built as part of the work presented here, constitutes a prototyping environment built on top of Fragmenta's mathematical theory that provides reasoning and transformation capabilities for metamodels and their instances. As we demonstrate, it can be used in real-world settings; ideally, however, Fragmenta designs should be specialised and optimised as part of fully fledged visual modelling environments. We also highlight how a model-driven engineering technology, with a mathematical foundation, is used to solve practical problems related with CPSs.

The algebraic loops healthiness check is performed on a graph describing the instantaneous dependencies between ports extracted from INTO-CPS architectural models; external port connections are derived from the CD and internal ones from the AD. Internal and external port dependencies of the INTO-SysML model must be consistent with the underlying model equations.

It is interesting to contrast the two model-based approaches to check algebraic loops. Alloy represents a graph directly (Section 3.7) as a relation between nodes; the property to check is stated as an ordinary relational calculus formula. The CSP approach (Section 3.4), on the other hand, is edge-oriented to suit CSP's communication model based on channels; a graph is the communications established between nodes (CSP processes) chosen from the environment (external choice); the property is expressed in an ingenious, but less evident way: through an abstract process and a traces refinement check. The life of a graph is simply the possible paths (communications) that can be chosen from the environment.

FDR3 and Alloy 4 are both based on model-checking; however, the CSP solution outperforms Alloy overwhelmingly. Alloy's exponential time complexity is attributed to the complexity of SAT whose worst-case time complexity is exponential [48, 41]; the Alloy solution resorts to the transitive closure, a computationally demanding operation (specialised algorithms do it in $O(n^3)$). An important factor in CSP's lower $O(n^3)$ time complexity lies in the use of traces refinement, founded on the simplest denotational model of CSP and with the least expensive time complexity (polynomial according to [37]).

The outstanding performance of Johnson's algorithm in our experiment just endorses its linear claim [36]. It is interesting to explain the oscillation portrayed in Fig. 15. The algorithm's complexity bound is $O((n + e)(c + 1))$, where n is the number of nodes, e the number of edges and c the number of cycles. In our experiment, a graph with a cycle (obtained from $WaterTanks\_loop_n$) is always followed by one without a cycle, hence, the oscillation.

Our CSP solution is beaten by Johnson's algorithm, but it is used in our verification approach, which employs FDR3 for more sophisticated checks of FMI co-simulations [3, 15]. It is difficult for general-purpose model-checking to outperform specialised algorithms taking advantage of problem specificities.

The experimental setup varies size but not structure, which remains essen-

tially the same throughout the different water tanks systems. However, as the results show, this is enough to expose differences; furthermore, as discussed above, the obtained results are consistent with theoretical results.

3.10 Related Work

Feldman et al [23] generate FMI model descriptions from Rhapsody SysML models and FMUs from statecharts to enable integration with continuous models. Unlike our work, this does not define a profile embodying a paradigm designed for multi-modelling and FMI-co-simulation; furthermore, formal static checks covering connector conformity and absence of algebraic loops are not covered. Pohlmann et al [52] propose a UML-based DSL for real-time systems; FMI FMUs are generated from model components described as real-time statecharts; our work specialises the SysML block diagrams, a standard notation for architectural modelling, and supports multi-modelling.

Our application of the FRAGMENTA theory presented in [4] required an extension to the Isabelle/HOL theory of [4], developed to prove that paper's main theorem. This extension builds an infrastructure to support automated verification and transformation for visual modelling languages. FRAGMENTA/Isabelle constitutes a prototyping environment supporting all the novel aspects of FRAGMENTA, namely: a formal theory of proxies and its verified theory of decomposition and the support for fragmentation strategies. To our knowledge, this is the first prototyping environment based on a proof assistant that provides formal reasoning and transformation capabilities for visual models.

The approach to connector conformity used here is based on typing. It supports sub-typing according to the inheritance relations specified in the metamodel; for instance, in INTO-SysML, natural numbers may be used when integers are expected because the metamodel says that the former is a subtype of the latter. This is checked as part of FRAGMENTA's typing morphisms. This is different from the *connector compatibility* of [19], which performs validations based on interface contracts, a relation between allowed inputs and outputs [67].

Broman et al [11] require that FMI component networks are algebraic-loop free as a pre-condition to the deterministic composition results of their FMI master algorithms, proposing port-dependency graphs as a means to perform such checks. Unlike the work presented here, [11] does not study different approaches to detect algebraic loops; it suggests algorithms that topologically

sort a graph, which yield an error if the graph has a cycle. Our algebraic loop analysis provides actual cycles as feedback to designers.

3.11 Conclusions on Checking SysML Models

We presented our approach to check a SysML model in preparation for co-simulation. This involves checking the consistency and well-formedness of the INTO-SysML model, which involves checking the conformance of the model with respect to its metamodel based on FRAGMENTA's representation. The actual checks are carried out using FRAGMENTA's Isabelle mechanisation, ensuring, among other things, connector conformity. We then showed how the INTO-SysML models could be transformed into other modelling languages to perform a check for the absence of algebraic loops using FRAGMENTA's Isabelle mechanisation as a transformation engine. We presented a novel CSP approach to detect algebraic loops by checking a traces refinement in FDR3. Our evaluation highlighted how our CSP approach based on refinement-checking performs well when compared with an Alloy SAT-based model-checking approach, but that it falls well short of a special-purpose graph algorithm. The work presented in this report is done in tandem with the effort on the formal semantics of FMI in CSP [3, 15].

Our contributions are as follows: (a) the SysML profile for architectural modelling of CPSs in a setting of multi-modelling and co-simulation that conforms to the FMI standard, (b) our approach to check the adequacy of a SysML model for co-simulation using a theorem prover and a model-checker, (c) the fact that we bring the theory for the representation of visual models presented in [4] into a practical setting to solve a real-world problem in an automated fashion based on FRAGMENTA's Isabelle mechanisation, which required enriching the Isabelle mechanisation presented in [4], (d) our CSP-based solution to the detection of algebraic loops, which is based on a novel approach to represent graphs in CSP, and (e) our evaluation of approaches to the detection of a algebraic loops.

Specialised algorithms for topological sorting are also able to disclose algebraic loops to a designer. For example, the Modelica tools use variants of an efficient algorithm from Tarjan for strong component analysis [65]. We choose exploit our verification tools directly.

4 Multi-modelling of Robots in SysML

For the development of robotic controllers, we propose the combined use of RoboChart with the notation of a simulation tool for continuous systems: any of Simulink, 20-sim, or OpenModelica, for instance. For illustration, we consider here control-law diagrams used in Simulink. The goal is to support the addition of detailed models for the robotic platform and for the environment. To identify these constituent models and their relationships, we propose to adapt the INTO-SysML profile.

A RoboChart module includes exactly one robotic platform, for which it gives a very abstract account. As already said, a RoboChart robotic platform defines just the variables, events, and operations available. In many cases, the operations are left unspecified, or are described just in terms of their effect on variables. In our ChemicalDetector module, for example, the operation move is specified just in terms of its effect on the variables ls and as of the robotic platform. There is no account of the actual laws of physics that control movement.

In a Simulink model, on the other hand, we can define the expected effect of the operations on the actual behaviour of the robot by capturing the laws of physics. In addition, we can also capture physical features of the environment that have a potential effect on the robot. To integrate the models, however, they need to share and expose events and variables. It is the purpose of the SysML model to depict the multi-models and their connections.

Next, we present the extensions (Sect. 4.1) and restrictions (Sect. 4.2) to the INTO-SysML profile that we require, mainly for the specification of RoboChart and Simulink model composition. They are mostly confined to the Architecture Structure Diagrams. Some are of general interest, and some support the use of RoboChart as a constituent model, in particular. In Sect. 4.3 we explain how we expect the resulting INTO-SysML profile to be used.

4.1 Extensions to INTO-SysML

The extensions are outlined in Table 1, and the restrictions later in Table 2. Both tables identify if the changes are specific to the needs of multi-models involving RoboChart diagrams, or if they are more generally useful for cyber-physical systems and, therefore, could be included in the original INTO-SysML profile.

#	**Description**	**INTO-SysML/RoboCalc**
E1	The components of the `System` block can include `LComponent` blocks.	Both
E2	We can have specialisations of `LComponent` blocks with stereotypes `Environment` and `RoboticPlatform` to group models for the environment and for the robotic platform. We can have any number of `Environment` blocks, but at most one block `RoboticPlatform`. These blocks can themselves be composed of any number of `LComponent` or `EComponent` blocks.	INTO-SysML may be extended to include an `Environment` block; however, `RoboticPlatform` should be RoboChart-specific.
E3	The type of a flow port optional.	RoboChart-specific
E4	The `Platform` of an `EComponent` is a `String` and can include any simulation tools. Alternatively, RoboChart and Simulink need to be admitted.	RoboChart-specific

Table 1: Overview of proposed extensions to the INTO-SysML profile

Figure 16 presents the definition of the Architecture Structure Diagram for the multi-models for the chemical detection system. The block ChemicalDetector represents the RoboChart module in Figure 2. The interface of a RoboChart module is defined by the variables and events in its robotic platform, which become ports of the SysML block that represents the module. The blocks Arena, WallSensor, MobilityHw, and ChemDHw represent Simulink models.

The first extension (E1) is about new `LComponent` blocks, which can be used to group models of the robotic platform or of the environment. They are logical blocks: they do not correspond to an actual component of the co-simulation. In our example, the `LComponent` block Rover represents the models for the robotic platform. It is composed of three `EComponent` blocks, representing Simulink models for the hardware for wall sensing, mobility, and chemical detection.

In fact, it is possible to define Rover as a `RoboticPlatform` block using the extension E2. On the other hand, we have just one `EComponent` that models one aspect of the environment, namely, Arena.

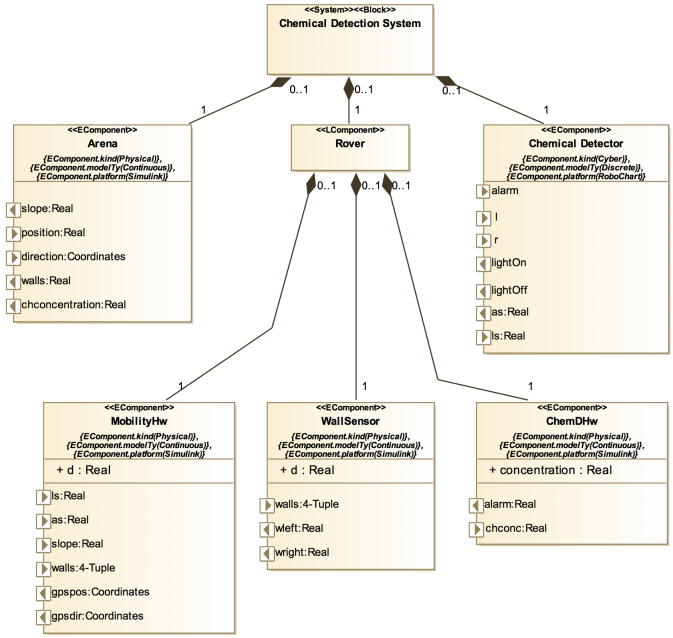

Figure 16: Chemical Detector Architecture Structure Diagram

Event-based communications are required in RoboChart, so we propose in E3 that ports may not carry any values. For instance, in our model, left and right are events of the robotic platform, as defined in RoboChart. As already said, they represent an indication of the presence of a wall from the sensors in the robotic platform. They carry no values.

The INTO-SysML profile does not include operations on blocks. This is due to the fact that blocks are intended to inform FMI model descriptions. The FMI standard considers interactions to be in the form of typed data passed between FMUs; this data is shared at each time step of a simulation. As such, the profile does not natively support the concepts of event-based or operation-based interactions. The new typeless ports representing RoboChart events are, therefore, handled by encoding event occurrence using real numbers 0.0 and 1.0.

Finally, the extension E4 includes extra values, namely, RoboChart and Simulink, for defining the platform of an `EComponent` block.

For the Connection Diagram, no changes to the INTO-SysML profile are required, except that we can have instances of `LComponent` blocks as well. For our example, the diagram is shown in Figure 17.

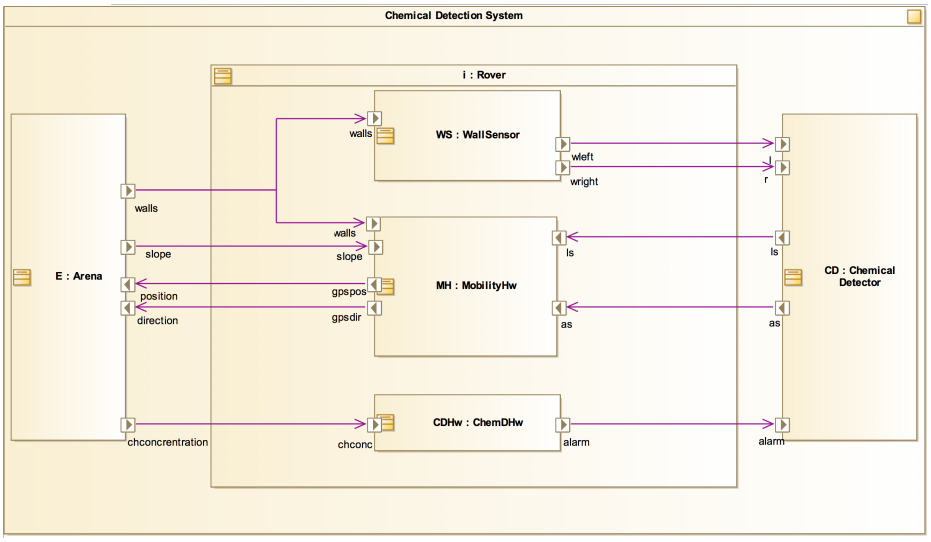

Figure 17: Chemical Detector Connection Diagram

4.2 Restrictions on INTO-SysML

As for the restrictions in Table 2, we have in R1 and R2 constraints on flow ports of `System` and `LComponent` blocks. Basically, a `System` block, Chemical Detection System in our example, is unique and can have no ports.

An `LComponent` block, like Rover in Figure 16, cannot have ports either. Since it is the `EComponent` blocks that represent multi-models, only they can contribute with inputs and outputs. For this reason, it does not make sense to include extra ports in an `LComponent` block, which just groups multi-models.

We recall that a `cyber` component models software aspects of the system. They could be specified using RoboChart and its abstraction facilities, which are tailored for mobile and autonomous robotic systems. So, R3 requires that there is just one `cyber` component: the RoboChart module that is complemented by the Simulink diagrams. In our example, this is the ChemicalDetector block.

For simulation, parameters and inputs need an initial value as enforced by R4. For a parameter, this is the default value used in a simulation, unless an alternative is provided. For the inputs, these are initial conditions that define the first set of outputs in the first step of the simulation. In our example, the initial values for position and direction, for instance, are (0.0,0.0) and (1.0,1.0). So, initially, the robot is stationary at a corner of the arena.

#	**Description**	**INTO-SysML / RoboCalc**
R1	There is exactly one `System` block and it cannot have flow ports.	Both
R2	An `LComponent` block has no ports.	Both
R3	In a diagram, we can have only one `EComponent` of `ComponentKind cyber` that must have `Platform` as RoboChart.	RoboChart-specific
R4	When the kind of a variable is `parameter` and when the `Direction` of a `FlowPort` with a type is `in`, its initial value must be defined.	Both
R5	Ports of a RoboChart block should be connected to the robotic platform, and not to the environment.	RoboChart-specific
R6	No use of `POComponent` is needed, if Simulink is used. If textual continuous models are adopted, these blocks can be used.	RoboChart-specific
R7	Typeless ports of a RoboChart block can be connected only to ports of type real of a Simulink block.	RoboChart-specific

Table 2: Overview of proposed restrictions to INTO-SysML profile

A controller can only ever sense or influence the environment using the sensors and actuators of the robotic platform. With R5, we, therefore, require that there is no direct connection between the controller and the environment. For example, Arena, representing a Simulink model of the environment, has an output port walls, that identifies as a 4-tuple the distances from the current position to the walls in the directions left, right, front, and back. This port is connected to the input port of the same name in WallSensor. It is this component that provides ports wleft and wright connected to the input ports l and r of ChemicalDetector.

We note that FMI does not have vector (array) types. So, strictly speaking, instead of ports with vector types, we should have separate ports for each component of the vectors. The inclusion of vectors in the FMI standard is, however, expected. We, therefore, make use of them in our example.

We require with R6 that no `POComponent` is included. For further detail in the

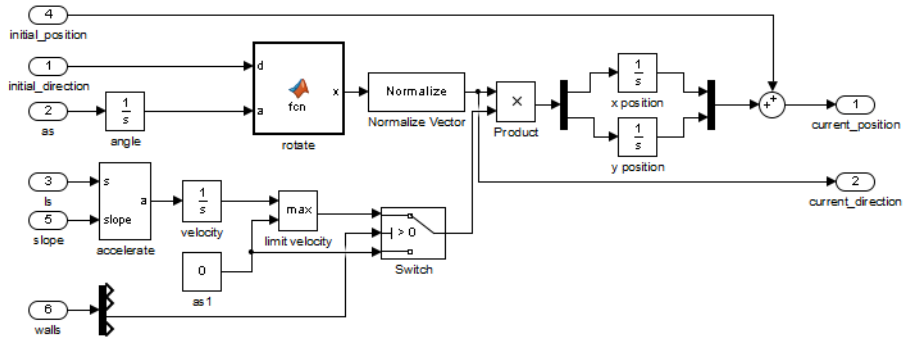

Figure 18: MobilityHw Simulink Diagram

models, we use RoboChart and Simulink, which are both diagrammatical. Of course, if only a textual continuous model is available, `POComponent` blocks can improve readability of the overall architecture of the system.

The only restriction relevant to a Connection Diagram is R7. Only ports of compatible types can be connected; compatibility between RoboChart and Simulink types is as expected. On the other hand, as already said, events in the RoboChart model that do not communicate values are represented by typeless ports of its SysML block. These ports can be connected to ports of a Simulink block representing a signal of type real. The values 0.0 and 1.0 can be used to represent the absence or occurrence of the event, for example.

The Simulink model for MobilityHw is shown in Figure 18. The input ports of MobilityHw, namely, ls, as, slope, and walls (see Figure 16) correspond to the input ports of the same name in the Simulink diagram. Moreover, the output ports gpspos and gpsdir correspond to the output ports current_position and current_direction of the Simulink diagram.

The Simulink diagram consists of two main sybsystems: rotate takes an angle and a vector as input and provides as output the result of rotating the vector by the given angle; and accelerate takes a linear speed (ls) and a slope and calculates the necessary acceleration profile to reach that velocity taking the slope into account. Essentially, given an initial_position and initial_direction, angular and linear speeds as and ls, a slope, and the distances walls to obstacles, the model calculates the movement of the robot. Restrictions over the speed are established by the block limit velocity. Restrictions over the position are described by the block Switch, which sets the speed to zero if there is a wall in front of the robot.

4.3 Use of INTO-SysML with RoboChart

In addition to the proposed required changes to the INTO-SysML profile, there are some specific issues to consider when defining a multi-model for RoboChart using the INTO-SysML profile. We outline these below.

Given a RoboChart module, the corresponding `EComponent` block has a particular form. First of all, it must have: kind as `cyber`; the `Platform` as RoboChart; and the `ModelType` as `discrete`. This is illustrated in Figure 16.

The variables of the RoboChart robotic platform become output ports. In our example, the variables ls and as of the Rover in Figure 2 become output ports of ChemicalDetector in Figure 16. The variables record the speeds required by the controller. This is used to define the behaviour of the mobility hardware.

The events of the RoboChart robotic platform are part of the visible behaviour of the RoboChart module. For this reason, they become flow ports in the `cyber EComponent`. In our example, we have events l, r, alarm, lightOn and lightOff in the block ChemicalDetector of the Architecture Structure Diagram in Figure 16, just like in the Rover of the module in Figure 2.

In the RoboChart module, the definition of the direction of the events is from the point of view of the software controllers. In the Architecture Structure Diagram, the point of view is that of the hardware and the environment. So, their directions are reversed. For instance, the Rover in Figure 2 can send the events l and r to the controller DetectAndFlagC indicating the presence of a wall on the left or on the right. In ChemicalDetector, however, these are input events. The relevant part of the hardware itself is modelled by the `EComponent` WallSensor, where the matching events wleft and wright are indeed outputs.

Operations of the RoboChart robotic platform, on the other hand, are not part of the visible behaviour of the module. So, they are not included. For instance, move, LoadFlag and ReleaseFlag are not in the `EComponent` Rover.

The flow ports of the `cyber EComponent` become flow ports of at least one of the `EComponent` blocks that represent the robotic platform. It is possible, however, that there are extra flow ports for communication between the models of the robotic platform and of the environment. In our example, for instance, an extra flow port slope is used for the environment model Arena to

inform the hardware model MobilityHw of the inclination of the floor, which has an effect on its control of movement to achieve the targeted speed.

4.4 Semantics

A CSP semantics for INTO-SysML is already defined in [2]. Our semantics here is different in two ways: it considers extensions and restrictions described above, and it is based on events that represent calls to one of three functions of the FMI API: `fmi2Set`, `fmi2Get`, and `fmi2DoStep`. In contrast, the semantics in [2] identifies events with flows. It gives a simulation view of the model, where behaviour proceeds in steps, but a data flow is one interaction. In FMI, a flow is established by a pair of calls to `fmi2Set` and `fmi2Get` functions.

As a consequence of our approach here, our semantics is useful to define specifications for FMI simulations. In [14], we present a CSP semantics for such simulations that can be automatically generated from a description of the FMUs and their connections, and a choice of master algorithm. Our semantics can be used to verify the correctness of those models.

The CSP process that defines the semantics of an INTO-SysML model uses communications on the following channels.

channel $fmi2Get : FMI2COMP \times PORT \times VAL \times FMI2STATUSF$
channel $fmi2Set : FMI2COMP \times PORT \times VAL \times FMI2STATUS$
channel $fmi2DoStep : FMI2COMP \times TIME \times NZTIME \times FMI2STATUSF$

The types of these channels match the signature of the corresponding FMI API functions. $FMI2COMP$ contains indices for each of the used instances of `EComponent` blocks, which represent FMUs in the INTO-SysML profile.

$PORT$ contains indices, unique to each `EComponent` instance, to identify ports, which represent input and output variables of the FMU. VAL is the type of valid values; we do not model the SysML or the FMI type system. For ports corresponding to a RoboChart event, special values (perhaps just 0 and 1) represent absence or presence of an event occurrence. VAL must include these values.

In FMI, there is one `fmi2Get` and one `fmi2Set` function for each data type. We, however, consider just one generic channel for each of them, since the overall behaviour of these functions is the same.

$FMI2STATUS$ and $FMI2STATUSF$ contain flags returned by a call to the API functions. In our model, all calls return the flag $fmi2OK$, indicating

$$
\begin{array}{l}
EComponent(ec) = \text{let} \\
\quad Init(setup) = \\
\quad\quad (|||\, var : setup \bullet fmi2Set.ec.var.InitialValues(ec, var).fmi2OK \longrightarrow SKIP) \\
\quad TakeOutputs(outs) = \\
\quad\quad sync \longrightarrow (|||\, var : outs \bullet fmi2Get.ec.var?x.fmi2OK \longrightarrow SKIP) \\
\quad DistributeInputs(inps) = \\
\quad\quad sync \longrightarrow (|||\, var : inps \bullet fmi2Set.ec.var?x.fmi2OK \longrightarrow SKIP) \\
\quad Step = sync \longrightarrow fmi2DoStep.ec?t?ss.fmi2OK \longrightarrow SKIP \\
\quad Cycle = TakeOutputs(Outputs(ec));\ DistributeInputs(Inputs(ec));\ Step;\ Cycle \\
\text{within} \\
\quad Init(Parameters(ec) \cup Initials(ec));\ Cycle
\end{array}
$$

Figure 19: CSP model of an `EComponent` block

success. So, the scenarios that it defines do not cater for the possibility of errors.

Finally, the types *TIME* and *NZTIME* define a model of time, using natural numbers, for instance. In the case of *NZTIME*, it does not include 0, since `fmi2DoStep` does not accept a value 0 for a simulation step size.

In what follows, we use the above channels to define CSP processes that correspond to `EComponent` instances (Section 4.5), and to co-simulations defined by a Connection Diagram for a `System` block (Section 4.6).

4.5 `EComponent` instances

The process $EComponent(ec)$ that defines the semantics of an `EComponent` block instance of index ec is specified in Figure 19. Its behaviour is described by an initialisation defined by the process *Init*, followed by a process *Cycle*.

We use functions *Parameters*, *Inputs*, *Outputs*, and *Initials*, which, given an index ec, identify the parameters, input and output ports, and input ports with initial values, of the block instance ec. Instances of the same `EComponent` block have the same parameters, inputs, outputs, and input ports with initial values.

In *Init*, the parameters and the input ports are initialised using values defined by a fifth function *InitialValues*, which, given a block *ec* and a parameter or input port *var*, gives the value of the parameter or the initial value of the port. Initialisation is via the channel $fmi2Set$.

Cycle defines a cyclic behaviour in three phases for an `EComponent`. It continuously provides values for its outputs, as defined by $TakeOutputs(Outputs(ec))$, takes values for its inputs, as defined by $DistributeInputs(Inputs(ec))$, and then carries out a step of simulation, as defined by *Step*. The channel $fmi2Get$ is used to produce the values of the outputs, $fmi2Set$ to take input values, and $fmi2DoStep$ to mark a simulation step. We use two functions *Outputs* and *Inputs* that define the output and input ports of a given `EComponent` *ec*.

$TakeOutputs(Outputs(ec))$ offers all the outputs *var* in $Outputs(ec)$ via the channel $fmi2Get$ in interleaving ($|||$). The particular value x output is not defined (as indicated by the ? preceding x). This value can be determined only by a particular model (in RoboChart or Simulink, for instance) for the `EComponent`. A channel *sync* is used to mark the start of the outputting phase.

$DistributeInputs(Inputs(ec))$ is similar, taking the inputs in $Inputs(ec)$ via $fmi2Set$. Finally, *Step*, after accepting a *sync*, takes an input via $fmi2DoStep$ of a time t and a step size ss, and terminates ($SKIP$).

In the semantics of a co-simulation, defined in the next section, we use the parallel composition ($[\![...]\!]$) below of instances of $EComponent(ec)$ for each `EComponent` block instance. The processes $EComponent(ec)$ synchronise on *sync* to ensure that they proceed from phase to phase of their cycles in lock step.

$$BlockInstances = ([\![sync]\!]\ ec : FMI2COMP \bullet EComponent(ec)) \setminus \{sync\}$$

The communications on *sync*, however, are hidden ($\setminus$). Therefore, as already indicated, the collective behaviour of the block instances is specified solely in terms of communications on the FMI API channels.

4.6 Co-simulation

A Connection Diagram for a `System` block instance characterises a co-simulation by instantiating blocks of the Architecture Diagram and defining how their ports are connected. So, we define the semantics of the `System` block instance as a co-simulation specified by the CSP process *CoSimulation* defined

$$
\begin{array}{l}
Connections = \text{let} \\
\quad Init = |||ec : FMI2COMP \bullet \\
\qquad (\Big|\Big|\Big| var : Initials(ec) \bullet fmi2Set.ec.var?x.fmi2OK \longrightarrow SKIP) \\
\quad Step = \|\ c : ConnectionIndex \bullet [AC(c)]\, Connection(c) \\
\quad Cycle = Step;\ Cycle \\
\text{within} \\
\quad Init;\ Cycle
\end{array}
$$

Figure 20: CSP model of a Connection Diagram

in the sequel. We note that the instances of `LComponent` blocks plays no role in the co-simulation semantics, since these blocks do not represent any actual component of a co-simulation, but just a logical grouping of constituent models.

Besides *BlockInstances* above, *CoSimulation* uses the process *Connections* shown in Figure 20. This is defined in terms of a parallel composition ($\|$) *Step* of processes $Connection(c)$ that define each of the connections c in a Connection Diagram, identified by indices in a set *ConnectionIndex*.

The *Init* process defines that, first of all, the variables corresponding to input ports of each component *ec* are initialised. Afterwards, *Connections* is defined by *Cycle*, which continuously behaves like *Step*.

A $Connection(c)$ process takes an output from the source port of the connection and gives it to the target port. In our example, we can, for instance, give the connection between the ports wright of WallSensor and r of ChemicalDetector the index 3. In this case, $Connection(3)$ is as follows, where *WallSensor* and *ChemicalDetector* are the indices in *FMI2COMP* for these `EComponent` block instances, and *WSwright* and *CDr* are variables corresponding to wright and r.

$$
\begin{array}{rl}
Connection(3) = & fmi2Get.WallSensor.WSwright?x.fmi2OK \longrightarrow \\
& fmi2Set.ChemicalDetector.CDr.x.fmi2OK \longrightarrow SKIP
\end{array}
$$

$Connection(3)$ ensures that the value x output via wright is input to the r port.

In the parallel composition in *Step*, each process $Connection(c)$ is associated with an alphabet $AC(c)$, which includes the communications over $fmi2Get$

and $fmi2Set$ that represent the connection it models. In our example, $AC(3)$ contains all communications over $fmi2Get$ with parameters $WallSensor$ and $WSwright$, and over $fmi2Set$ with parameters $ChemicalDetector$ and CDr. These alphabets ensure that if there are several connections with the same source port, they share the output in the port by synchronising on that communication. In our example, the output $Awalls$ of $Arena$ is shared between the processes for the connections between Arena and WallSensor and between Arena and MobilityHw.

Finally, the semantics for the co-simulation defined by a Connection Diagram for a `System` instance is the parallel composition below.

$$CoSimulation = BlockInstances \llbracket FMIGetSet \rrbracket Connections$$

The processes synchronise on communications on the set $FMIGetSet$ containing the union of the alphabets of the $Connection(c)$ processes.

If there are ports that are not associated with a connection, their corresponding communications via $fmi2Set$ or $fmi2Get$ are not included in $FMIGetSet$. These communications are restricted only by the process $BlockInstances$. In our example, the ports lightOn and lightOff are not connected to any other ports. Their outputs are visible to the environment of the chemical detection system, but not connected to any other modelled components.

The behaviour defined by $CoSimulation$ specifies cyclic simulation whose steps contains three phases: all outputs are taken in any order, used to provide all inputs, also in any order, and then the time advances via a simulation step of each multi-model. As already said, a CSP semantics for an FMI co-simulation is available [14]. With that, $CoSimulation$ can be used as a specification to validate an FMI co-simulation where the FMUs correspond to the `EComponent` block instances and must be orchestrated as indicated in the connection diagram.

4.7 Related work

Dhouib *et al.* describe a UML profile for a graphical domain-specific language for robotics [18] that supports model development and automatic platform-independent code generation. Their language is designed for modelling and reasoning about non-functional properties. Lima *et al.* also propose a model-based engineering approach to robotic systems [60] in their UML-based component framework. A variety of inter-component communication patterns are involved; however, the semantics for component controllers are not defined.

RoboChart is a domain-specific language for designing and verifying robot controllers; it is small and has a well-defined semantics to support sound generation of formal models and their simulations.

There are (famously) many different semantics for UML state machines [21]. Kuske *et al.* [42] give semantics for UML class, object, and state-machine diagrams using graph transformation. Rasch and Wehrheim [55] use CSP to give semantics for extended class diagrams and state machines. Davies and Crichton [16] also use CSP to give semantics for UML class, object, state-chart, sequence, and collaboration diagrams. Broy *et al.* [12] present a foundational semantics for a subset of UML2 using state machines to describe the behaviour of objects and their data structures. RoboChart state machines have a precise semantics in CSP in the spirit of [55] and [16]; however, for the sake of compositionality, RoboChart state machines do not include history junctions and inter-level transitions.

UML 2.0 includes a timing diagram, a specific kind of interaction diagram with added timing constraints. The UML-MARTE profile [63] provides richer models of time based on clocks, including notions of logical, discrete, and continuous time. The Clock Constraint Specification Language (CCSL) provides for the specification of complex timing constraints, including time budgets and deadlines. This is accomplished with sequence and time diagrams; it is not possible to define timed constraints in terms of transitions or states.

UML-RT [62] encapsulates state machines in capsules; inter-capsule communication is through ports and is controlled by a timing protocol with timeouts. More complex constraints, including deadlines, are specified only informally.

The work in [54] defines a semantics for a UML-RT subset in untimed *Circus* [68]. An extension to UML-RT is considered in [1] with semantics given in terms of CSP+T [69], an extension of CSP that supports annotations for the timing of events within sequential processes. The RoboChart timed primitives are richer and are inspired by timed automata and Timed CSP [61].

Practical work on master algorithms for use in FMI co-simulations includes generation of FMUs, their simulations, and hybrid models [6, 51, 22, 17]. FMUs can encapsulate heterogeneous models; Tripakis [66] shows how components based on state machines and synchronous data flow can be encoded as FMUs. In our approach, we have a hybrid co-simulation, but each `EComponent` is either `discrete` or `continuous`. Extensions to FMI are required to deal with that [10].

Savicks [58] shows how to co-simulate Event-B and continuous models using a fixed-step master algorithm. Savicks does not give semantics for the FMI API, but supplements reasoning in Event-B with simulation of FMUs within Rodin, the Event-B platform, applying the technique to an industrial case study [59]. The work does not wrap Event-B models as FMUs, and so it does not constitute a general FMI-compliant co-simulation. Here, we do not consider the models of FMUs, but plan to wrap CSP-based models of Simulink [13] and RoboChart [47] to obtain CSP-based FMUs models that satisfy the specification in [14].

4.8 Conclusions on Multi-Modelling of Robots

We have extended and restricted the INTO-SysML profile to deal with mobile and autonomous robotic systems. For modelling the controller(s), we use RoboChart. For modelling the robotic platform and the environment, we use Simulink. We have also given a behavioural semantics for models written in the profile using CSP. The semantics is agnostic to RoboChart and Simulink, and captures a co-simulation view of the multi-models based on the FMI API.

Our semantics can be used in two ways. First, by integration with a semantics of each of the multi-models that defines their specific responses to the simulation steps, we can obtain a semantics of the system as a whole. Such semantics can be used to establish properties of the system, as opposed to properties of the individual models. In this way, we can confirm the results of (co-)simulations via model checking or theorem proving, for example.

There are CSP-based formal semantics for RoboChart [47] and Simulink [13] underpinned by a precise mathematical semantics. Our next step is their lifting to provide an FMI-based view of the behaviour of models written in these notations. With that, we can use RoboChart and Simulink models as FMUs in a formal model of a co-simulation as suggested here, and use CSP and its semantics to reason about the co-simulation. For RoboChart, for example, the lifting needs to transform inputs of values 0.0 and 1.0 on ports for typeless events to synchronisations.

It is also relatively direct to wrap existing CSP semantics for UML state machines [55, 16] to allow the use of such models as FMUs in a co-simulation. In this case, traditional UML modelling can be adopted.

Secondly, we can use our semantics as a specification for a co-simulation. The work in [14] provides a CSP semantics for an FMI co-simulation; it covers

not only models of the FMUs, but also a model of a master algorithm of choice. The scenario defined by an INTO-SysML model identifies inputs and outputs, and their connections. The traces of the FMI co-simulation model should be allowed by the CSP semantics of the INTO-SysML model.

There is no support to establish formal connections between a simulation and the state machine and physical models (of the robotic platform and the environment). The SysML profile proposed here supports the development of design models via the provision of domain-specific languages based on familiar diagrammatic notations and facilities for clear connection of models. Complementarily, as explained above, the semantics of the profile supports the verification of FMI-based co-simulations.

There are plans for automatic generation of simulations of RoboChart models [47]. The semantics we propose can be used to justify the combination of these simulations with Simulink simulations as suggested above.

Within INTO-CPS, we plan to contribute to the robot case study and pilot projects.‘

Acknowledgements

Thanks are due to Etienne Brosse, who implemented the INTO-SysML profile in the Modelio tool, and Bernhard Thiele, who provided useful feeedback on the work presented here.

References

5 References

[1] K. B. Akhlaki, M. I. C. Tunon, J. A. H. Terriza, and L. E. M. Morales. A methodological approach to the formal specification of real-time systems by transformation of UML-RT design models. *Science of Computer Programming*, 65(1):41–56, 2007.

[2] N. Amálio. Foundations of the SysML profile for CPS modelling. Technical Report D2.1a, INTO-CPS project, 2015.

[3] Nuno Amálio, Ana Cavalcanti, Christian König, and Jim Woodcock. Foundations for FMI Co-Modelling. Technical report, INTO-CPS Deliverable, D2.1d, December 2015.

[4] Nuno Amálio, Juan de Lara, and Esther Guerra. FRAGMENTA: A theory of fragmentation for MDE. In *MODELS 2015*. IEEE, 2015.

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

[6] J. Bastian, C. Clauß, S. Wolf, and P. Schneider. Master for co-simulation using FMI. In *Modelica Conference*, 2011.

[7] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. The Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *Modelica Conference*, Munich, Germany, 2012.

[8] Torsten Blochwitz. Functional mock-up interface for model exchange and co-simulation. https://www.fmi-standard.org/downloads, July 2014. Torsten Blochwitz Editor.

[9] J. F. Broenink. Modelling, simulation and analysis with 20-sim. *Computer Aided Control Systems Design*, 38(3):22–25, 1997.

[10] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *ACM SIGBED International Conference on Embedded Software*. IEEE, 2013.

[11] D. Broman, C. Brooks, L. Greenberg, E.A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *EMSOFT*, 2013.

[12] M. Broy, M. V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.

[13] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465–512, 2011.

[14] A. L. C. Cavalcanti, J. C. P. Woodcock, and N. Amálio. Behavioural Models for FMI Co-simulations. Technical report, University of York, Department of Computer Science, York, UK, 2016. Available at `www-users.cs.york.ac.uk/~alcc/CWA16.pdf`.

[15] Ana Cavalcanti and Jim Woodcock. Foundations for FMI Co-Modelling. Technical report, INTO-CPS Deliverable, D2.2d, December 2016.

[16] J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2-3):118–145, 2003.

[17] J. Denil, B. Meyers, P. De Meulenaere, and H. Vangheluwe. Explicit semantic adaptation of hybrid formalisms for FMI co-simulation. In *Spring Simulation Multi-Conference*, 2015.

[18] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. *Simulation, Modeling, and Programming for Autonomous Robots*, chapter RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications, pages 149–160. Springer, 2012.

[19] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. Compositional semantics and analysis of hierarchical block diagrams. In *SPIN 2016*, pages 38–56, 2016.

[20] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[21] R. Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, 2009.

[22] Y. A. Feldman, L. Greenberg, and E. Palachi. Simulating rhapsody SysML blocks in hybrid models with FMI. In *Modelica Conference*, 2014.

[23] Yishai Feldman, Lev Greenberg, and Eldad Palachi. Simulating Rhapsody SysML Blocks in Hybrid Models with FMI. *Modelica Conference*, pages 43–52, 2014.

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

[25] FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org`, 2014.

[26] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, 2004.

[27] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, January 2004.

[28] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

[29] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 187–201, 2014.

[30] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.

[31] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[32] J. A. Hilder, N. D. L. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. A. Kilgour, J. Timmis, and A. M. Tyrrell. Chemical detection using the receptor density algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1730–1741, 2012.

[33] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.

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

[35] Daniel Jackson. *Software abstractions: logic, language, and analysis*. MIT Press, 2012.

[36] Donald B. Johnson. Finding all the elementary circuits in a directed graph. *SIAM Journal of Computing*, 4(1):77–84, 1975.

[37] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[38] Christian Kleijn. Modelling and Simulation of Fluid Power Systems with 20-sim. *Intl. Journal of Fluid Power*, 7(3), November 2006.

[39] J. Klein and L. Spector. 3D Multi-Agent Simulations in the breve Simulation Environment. In *Artificial Life Models in Software*, pages 79–106. Springer, 2009.

[40] R. Kübler and W. Schiehlen. Two Methods of Simulator Coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2):93–113, 2000.

[41] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theoretical Computer Science*, 223(1–2):1–72, 1999.

[42] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In M. Butler, L. Petre, and K. SereKaisa, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.

[43] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The Overture initiative – integrating tools for VDM. *SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.

[44] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[45] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[46] The MathWorks,Inc. *Simulink*. `www.mathworks.com/products/-simulink`.

[47] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, J. Timmis, and J. C. P. Woodcock. RoboChart: a State-Machine

Notation for Modelling and Verification of Mobile and Autonomous Robots. Technical report, University of York, Department of Computer Science, York, UK, 2016. Available at `www.cs.york.ac.uk/circus/publications/techreports/reports/-MRLCTW16.pdf`.

[48] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.

[49] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[50] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012.

[51] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating functional mockup units from software specifications. In *Modelica Conference*, 2012.

[52] Uwe Pohlmann, Wilhelm Schäfer, Hendrik Reddehase, Jens Röckemann, and Robert Wagner. Generating Functional Mockup Units from Software Specifications. In *Modelica Conference*, 2012.

[53] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[54] R. Ramos, A. C. A. Sampaio, and A. C. Mota. A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *Formal Methods for Open Object-based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 99–114, 2005.

[55] H. Rasch and H. Wehrheim. Checking consistency in UML diagrams: Classes and state machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2003.

[56] E. Rohmer, S. P. N. Singh, and M. Freese. V-rep: A versatile and scalable robot simulation framework. In *IEEE International Conference on Intelligent Robots and Systems*, volume 1, pages 1321–1326. IEEE, 2013.

[57] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[58] V. Savicks, M. Butler, and J. Colley. Co-simulating Event-B and Continuous Models via FMI. In *Summer Simulation Multiconference*, pages 37:1–37:8. Society for Computer Simulation International, 2014.

[59] V. Savicks, M. Butler, and J. Colley. Co-simulation Environment for Rodin: Landing Gear Case Study. In F. Boniol, V. Wiels, Y. A. Ameur, and K.-D. Schewe, editors, *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 148–153. Springer, 2014.

[60] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *14th International Conference on Advanced Robotics*, pages 1–8. IEEE, 2009.

[61] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.

[62] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 1998.

[63] B. Selic and S. Grard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers Inc., 2013.

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

[65] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. 1(2):146–160 1972.

[66] S. Tripakis. Bridging the semantic gap between heterogeneous modeling formalisms and FMI. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 60–69. IEEE, 2015.

[67] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM TOPLAS*, 33(4), 2011.

[68] J. C. P. Woodcock and A. L. C. Cavalcanti. ***Circus***: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, 2001.

[69] J. J. Zic. Time-constrained Buffer Specifications in CSP + T and Timed CSP. *ACM Transactions on Programming Languages and Systems*, 16(6):1661–1674, 1994.