Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

Foundations of the SysML profile for CPS modelling

Technical Note Number: D2.1a

Version: 0.8

Date: Month Year

Public Document

http://into-cps.au.dk

Contributors:

Nuno Amálio, UY
Richard Payne, NCL
Ana Cavalcanti, UY
Etienne Brosse, ST
Jim Woodcock, UY

Editors:

Nuno Amálio, UY

Reviewers:

Stylianos Basagiannis, 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	19-05-2015	Nuno Amálio	Initial document version
0.2	14-08-2015	Nuno Amálio	Metamodels of INTO-CPS/SysML profile and initial literature review
0.3	07-09-2015	Nuno Amálio	Minor corrections.
0.4	16-09-2015	Nuno Amálio	Added support for variable definitions in blocks
0.5	05-10-2015	Nuno Amálio	Added CSP semantics and fragmented metamodels
0.6	12-11-2015	Etienne Brosse	Added Modelio part
0.7	14-11-2015	Nuno Amálio	Added introduction, and conclusions. Revised abstract.
0.8	15-12-2015	Nuno Amálio	Revised document according to feedback coming from internal project reviews.

Abstract

This report investigates the foundations of SysML and proposes a SysML profile with a formal semantics for cyber-physical systems to be further developed and used in the context of the INTO-CPS project. The profile is based on a subset of SysML notations, namely, block-definition and internal-block diagrams, and is designed to embrace the project themes on multi- and heterogenous modelling and co-simulation. The profile's syntax is described with UML class models that define the profile's metamodels; the semantics is described using the CSP process algebra, which denotes an underlying UTP model based on the CPS's UTP semantics. The report illustrates visual modelling using the profile and its underlying CSP semantics with several examples, and presents how the profile has been implemented in the Modelio tool to enable the construction of SysML/INTO-CPS diagrams.

Contents

1 Introduction

The systems modelling language (SysML) [OMG12] builds up on the Unified Modelling language (UML) to provide a general-purpose notation for *systems engineering*. SysML aims at supporting systems that present *hybrid* phenomena, mixing the continuous phenomena of physical systems and the discrete phenomena of software systems, as is typical of cyber-physical systems (CPSs). Following suite on UML's success, SysML is seen as a notation that may have an impact on the mainstream development of CPSs.

A main difference and novelty with respect to UML lies in SysML's emphasis on communication ports and system architectures. Ports provide the means for communication between components. SysML supports two different kinds of ports: (i) *standard ports*, which support a communication mechanism based on events and message passing that is typical of software systems, and (ii) *flow ports*, which is closer to the way component communication works in continuous systems, where information flows from one component to the other through variables.

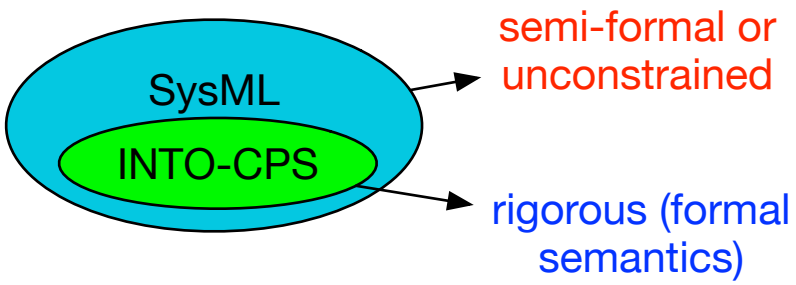


Figure 1: Usage of SysML in the INTO-CPS project is divided into zones. The INTO-CPS (or rigorous zone) is given a formal semantics.

This reports investigates the foundations of SysML and proposes a SysML profile for the INTO-CPS project with well-defined foundations. It is this profile that includes a subset of SysML notations that are given a formal semantics, which is also described here. The INTO-CPS project follows, therefore, an approach to model-based systems development with SysML that is *hybrid* with respect to the semantics as described in Figure 1: the whole SysML may be used, but it is only the INTO-CPS profile that is given a formal semantics, the remaining notations (such as use case diagrams) are used in its semi-formal form.

The profile has been designed to embrace the main themes of the INTO-CPS project, namely multi- and heterogenous modelling and co-simulation. In the first year of the INTO-CPS project, the profile's diagram types specialise the SysML notations of block-definition and internal-block diagrams to describe architectures that take multi-modelling and co-simulation into account.

The syntax of the SysML/INTO-CPS profile is described using UML class diagrams that define the *metamodels* of profile's diagram types. To master the complexity of the metamodels, we split them into fragments following the FRAGMENTA theory [AdLG15]. The SysML/INTO-CPS profile is given a semantics in the CSP process algebra [Hoa85], which acts as a front-end for an underlying UTP (Unifying Theories of Programming [HJ98]) model based on CSP's UTP semantics. The SysML/INTO-CPS profile has been implemented in the Modelio tool, which provides diagram editors that enable the construction of SysML/INTO-CPS diagrams.

This remainder of this report is as follows:

- Section 2 surveys the literature in the area of SysML, but focussing on semantics. This section is divided in two related but different themes: abstract formal semantics, and semantics for co-simulation.
- Section 3 gives the report's running example, which is used to illustrate the SysML/INTO-CPS profile and its semantics.
- Section 4 presents the diagram types of the SysML/INTO-CPS profile, namely *architecture structure diagrams* (ASDs) and *connection diagrams* (CDs), illustrating them with the report's running example.
- Section 5 describes the metamodels of the diagrams types that make the SysML/INTO-CPS profile.
- Section 6 presents the semantics of the SysML/INTO-CPS profile expressed in the CSP process algebra. It shows how the report's running example can be described in CSP using the profile's CSP semantics.
- Section 7 describes the implementation of the profile in the Modelio tool. It shows how the metamodels describing the syntax of SysML/INTO-CPS have been translated into Modelio.
- Section 8 draws the report's conclusions.
- Appendix A presents the SysML/INTO-CPS diagrams of this deliverable's running example as drawn in the current implementation of the SysML/INTO-CPS profile in the INTO-CPS tool Modelio.

- Appendix B presents further illustrations of the SysML/INTO-CPS profile, highlighting both the visual modelling using the profile's diagrams and the underlying CSP semantics. These examples, together with the report's running example, have been used to design and validate the profile presented here.

2 Related Work

The systems modelling language (SysML) [OMG12] extends a subset of the Unified Modelling language (UML) to support modelling of heterogeneous systems. Typically, such systems are *hybrid* in that they present both *continuous* and *discrete* phenomena that characterises physical and software systems, respectively. SysML emphasises, therefore, *systems engineering* and holistic modelling of cyber-physical systems (CPSs), in contrast to UML's traditional software-centric world and its emphasis on *software engineering*.

CPSs are integrations of computation and physical processes and they are inherently heterogenous [DLV12]. SysML targets CPSs and their heterogeneity by aiming to support many underlying models of computations (MoCs). In this respect, there is a substantial change from UML, which has a core MoC based on the object-oriented (OO) paradigm with its procedural model of component interaction based on interfaces made of methods (or procedures) with type signatures [Lee03]; OO ends up being a MoC that is inclined towards single processors and sequential computations. UML extensions build-up in this core (for example, to support real-time systems). SysML embraces CPSs intrinsic concurrency model where many things occur at the same time and puts a big emphasis on *interoperability*, that is the ability of two or more software components to interact despite differences in language, interface and execution platform [Weg96].

Like UML, SysML is a *semi-formal* language: it has a formal syntax, but no formal semantics. The standards defining both UML and SysML provide well-defined definitions of syntax accompanied by informal explanations of semantics. It is, therefore, not a surprise that SysML follows UML's trend with respects to the *semantics* issue. Like UML, SysML needs to be used in rigorous settings, hence it needs to be a precise language, and it needs, therefore, a formal or mathematical semantic basis. Throughout its development, the UML has been criticised for its semantics problems [EFLR98, Ste02, FGDT06, RF11, MB11]. This inherent semantic com-

plexity, where resolution of semantics is a matter of interpretation in a given usage setting, which is often not always clearly defined, appears to be aggravated in SysML due to SysML's inherent heterogeneity and its support for its many underlying MoCs.

The next sections discuss related work in the area of SysML, emphasising two distinct trends: *abstract formal models of semantics* and *semantics for co-simulation*.

2.1 Abstract Formal Semantics

The SysML formalisation effort, like its UML predecessor, relies mostly on the *denotational* approach to semantics. This is termed *translation*, whereby diagrams are couched in a notation that has a formally defined semantics.

Ding and Tang [DT10] formalise SysML block definition diagrams (BDDs) into a description logic. This approach relies on blocks made up of attributes and operations; the formalisation closely resembles logic-based formalisations of UML class diagrams [AP03]. Similarly, Graves and Bijan [GB11] formalise SysML into a description logic. The authors develop a semantic domain, called *abstract block diagram* logic, to couch block-based SysML diagrams, giving a treatment of SysML blocks that is akin to OO classes.

Bouabana-Tebibel et al [BTRB12] developed a formalisation of internal block definition diagrams (IBDs) in Hierarchical Coloured Petri Nets, focussing on IBD characteristics, such as block-nesting and port-communication.

Unlike the previous approaches, Chouali and Hammad [CH11] take a more holistic approach to semantics by considering a subset of diagrams. Their formalisation into interface automata [dAH01] emphasises component assembly to enable the verification of component interoperability. In this SysML-based approach, component-based system architectures are specified using SysML BDDs, composition links are specified using SysML IBDs, and component protocols are described with sequence diagrams.

The formalisation of Miyazawa et al [MLC13] focusses on SysML blocks formalising them as CSP processes in the formal language CML [WCF+12], a combination of VDM and CSP that is a spin-off of the Circus formal language [WC02, CSW03], a combination of Z and CSP. This work is then taken as the basis for a notion of formal refinement for SysML [MC14]. Similar to [CH11], this work emphasises component assembly described as CSP

parallel composition.

2.2 Semantics for co-simulation

One of the main ingredients of INTO-CPS' approach to the interoperability goal is model co-simulation. Many SysML works look at the semantics question pragmatically and see semantics as a matter of direct support for co-simulation, the ability to execute or simulate a model. INTO-CPS's co-simulation approach is based on the tool-independent Functional Mock-up Interface (FMI) standard [FMI14, BOA+11, BOA+12]. FMI wraps models from different tools in Functional Mock-up units (FMUs), enabling inter-FMU communication and importing into hosting tools. The models are seen as black boxes that need to comply with the FMU interface, ensuring protection of intellectual property and the required interoperability.

Feldman et al [FGP14] developed an approach to generate FMI code from Rhapsody SysML models that wraps statecharts as FMUs. The authors acknowledge problems with FMI co-simulation of statecharts due to the standard's lack of support for instantaneous events and other subtle differences that cause semantic discrepancies between FMI co-simulation and Rhapsody settings.

Several approaches generate SystemC code from SysML with the purpose of model simulation and executability [BJ11, CdSH+13, WCMG15]. This follows the model driven engineering trend, where models are built with the direct aim of code-generation to enable simulation (or executability). SystemC [IEE12] is a C++ based framework that provides an event-based simulation environment being often described as a system-level modelling language. Brasil et al [BJ11] generate SystemC models from SysML descriptions made up of blocks, flow ports and operations. Café et al [CdSH+13] tackles heterogeneity by generating co-simulation models described in *SystemC-AMS* from SysML descriptions of different MoCs, the generated simulation models constituting the semantics of the SysML model. SystemC-AMS provides pre-built MoCs allowing co-simulation of continuous and discrete components. In [CdSH+13], the different SysML diagrams state the MoC setting, resulting in the corresponding MoC encoding in SystemC-AMS. Wawrzik et al [WCMG15] propose a framework that is similar to that of Café et al [CdSH+13]; they translate SysML descriptions into SystemC to enable the simulation of the modelled CPS, using specific dialects of System-C designed to tackle the phenomena being modelled (e.g. hardware/software, network and propagation and analog and physical processes).

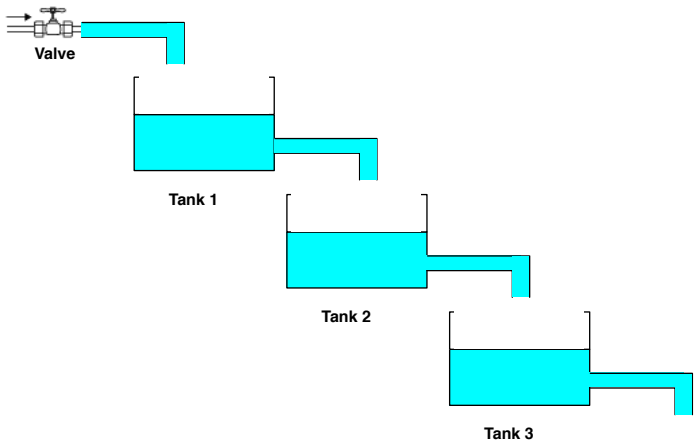


Figure 2: A sketch of the physical layout of the three cascading water tanks system

3 Running Example

This document's running example is the three cascading water tanks sketched in Figure 2. This system controls three physical water tanks through a valve that can turn the inflow of water into the first tank on and off, which results in a chain of flows, with the outflow of one tank constituting the inflow of the next. The valve is turned on and off periodically.

The next sections use this system to illustrate the SysML/INTO-CPS profile presented here.

4 SysML Diagram Types

The INTO-CPS SysML profile embraces the project's interoperability, CPS and holistic modelling themes, emphasising the heterogeneity that exists between the continuous world of physical systems and the discrete world of software systems. At this stage, the profile focusses on multi-modelling and architectural specification.

The diagram types of INTO-CPS in year 1 are as follows:

- **Architecture Structure Diagrams (ASDs).** INTO-CPS ASDs specialise SysML block-definition diagrams [OMG12] to support the specification of a system architecture described in terms of a system's

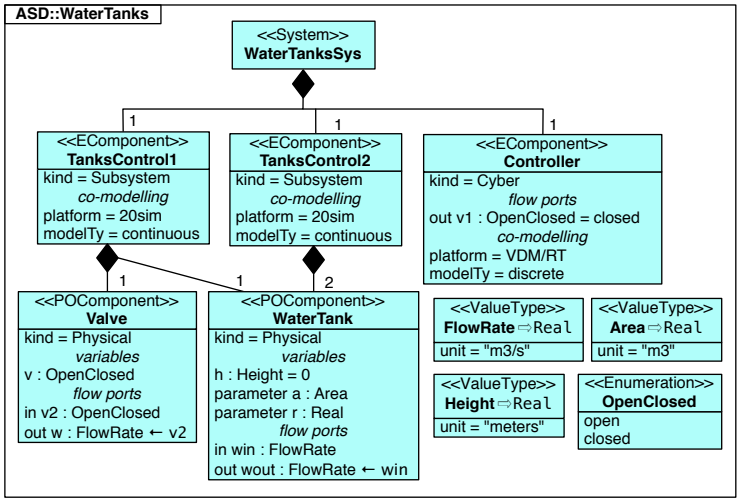
components. A component is a logical or conceptual unit of the system, corresponding to software or a physical entity. ASDs emphasise multi-modelling by outlining that certain components encapsulate a model; not all components have an associated model; some are part of a larger component that has its own model. Components are classified as *cyber*, *physical* and *sub-system*. Cyber components encapsulate some functional logic. A physical component represents an entity of the physical world. A sub-system is an assembly of cyber and physical components and possibly other sub-systems.

- **Connections Diagrams (CDs).** INTO-CPS CDs specialise SysML internal block-definition diagrams [OMG12] to convey the internal configuration of the system's components in terms of the way they are connected.

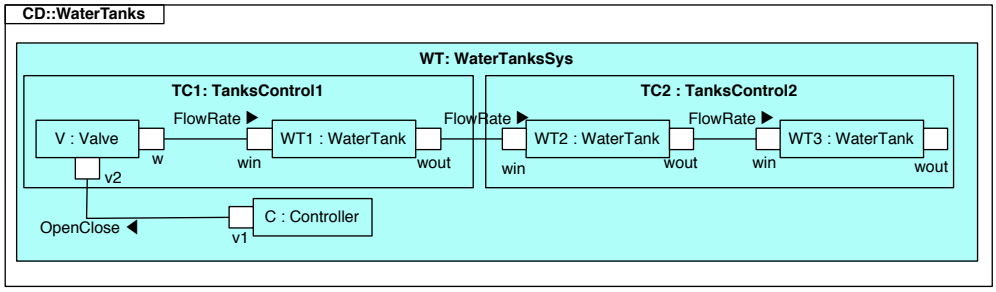
INTO-CPS SysML ASDs and CDs are illustrated with the three cascading water tanks running example of Fig. 3. The diagrams are as follows:

- The ASD (Fig. 3(a)) introduces blocks representing the components of the system. The `system` block represents the system as a whole; it is divided into three components that have stand-alone models (they include a *co-modelling* section in the block's properties compartment): the subsystems `TanksControl1` and `TanksControl2`, both continuous models described in *20-sim*, and the cyber component `Controller`, a discrete model described in *VDM-RT*. The physical blocks `Valve` and `WaterTank` represent the corresponding physical elements of the system. Component `TanksControl1` controls one `Valve` and two `WaterTanks`; component `TanksControl2` controls one `WaterTank`. The section *flow ports* defines the ports of each component. For instance, the `Valve` block includes the `valveI` port, to represent the fact that the valve may be turned `on` or `off`.
- The CD (Fig. 3(b)) describes the internal configuration of the system and complements the description provided by the ASD. It indicates how the different components are connected through ports and the information (types) that flows through those ports.

The diagrams of Fig. 3 drawn in the current implementation of the profile in the INTO-CPS tool Modelio are given in appendix A (Figure 19). Further examples of usage of the INTO-CPS SysML profile presented here are given in appendix B.



(a) Architecture Structure Diagram



(b) Connections Diagram

Figure 3: The SysML/INTO-CPS architectural and connections diagrams of the *three cascading water tanks* system.

5 Metamodels

This section presents the metamodels of the SysML/INTO-CPS profile, which are partitioned into fragments following the FRAGMENTA theory [AdLG15] of model fragmentation. This eases the manipulation of metamodels that are required for the semantics. FRAGMENTA uses a mechanism of proxies to refer to model elements defined in some other fragment.

5.1 Architecture Structure Diagrams

ID	Description	OCL
WF1	At most one system instance	`context System inv:` `System::allInstances()->size() = 1`

WF2	Enumeration literals must be distinct	`context Enumeration inv :`
		`literals->forall(l1, l2 |`
		`  l1 <> l2 implies l1.name <> l2.name)`

WF3	Dependencies of output `FlowPorts`	`context FlowPort inv :`
		`direction = Direction.in`
		`  implies depends.size() = 0`

WF4	Subsystems are instances of `EComponent`	`context Component inv:`
		`kind = ComponentKind::subsystem implies`
		`  self in EComponent::allInstances()`

WF5	`Systems` contain `EComponents` only	`context Composition inv:`
		`src in System::allInstances()`
		`  implies tgt in EComponent::allInstances()`

WF6	`EComponents` contain `POComponents` only	`context Composition inv:`
		`src in EComponent::allInstances()`
		`  implies tgt in POComponent::allInstances()`

WF7	`POComponents` cannot contain other elements	`context Composition inv:` `  not (src in POComponent::allInstances())`

Table 1: OCL Well-formedness constraints of the metamodel of ASDs

The overall metamodel of ASDs is presented in Figure 4; the associated well-formedness constraints, expressed in the object constraint language (OCL) [WK03], are given in table 1. Instances of this metamodel are given in Figure 3(a) and in the example ASDs given in appendix B. The overall metamodel of ASDs is partitioned into the following fragments: *ASD*, *properties and primitive types*, *blocks*, *value types*, *compositions*.

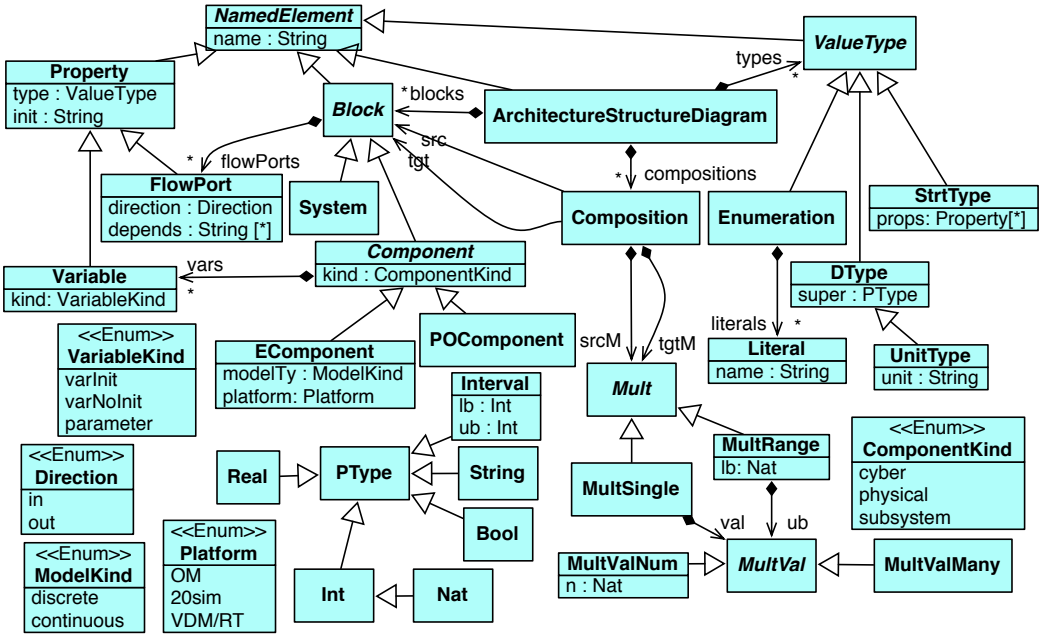


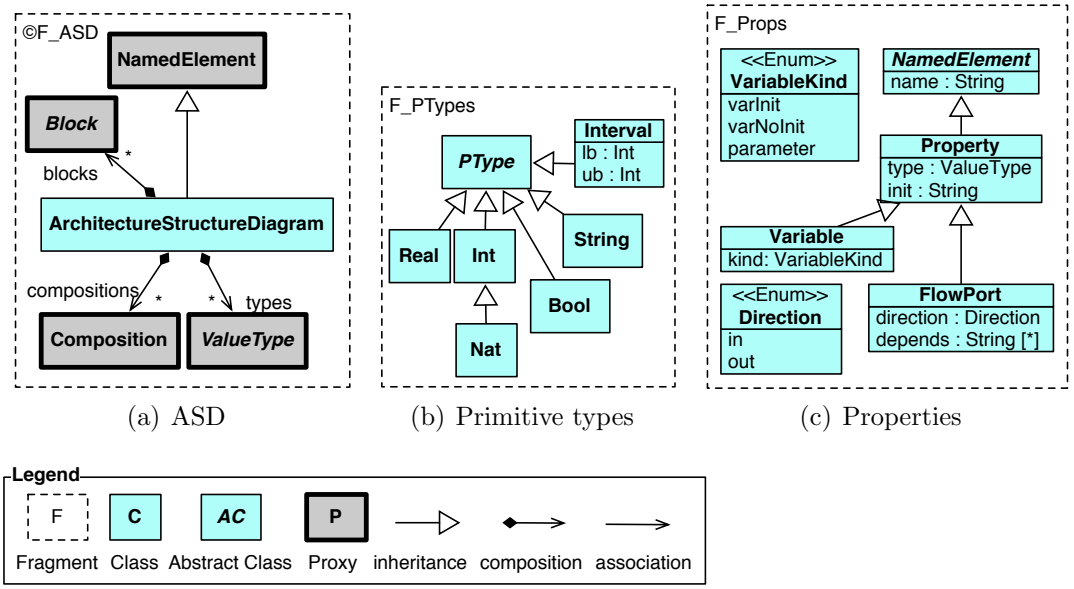
Figure 4: Metamodel of INTO-CPS SysML architectural structure diagrams

The ASD metamodel fragment (`F_ASD`, Figure 5(a)) introduces an `ArchitectureStructureDiagram`, saying that it is a `NamedElement` and comprises a collection of `Blocks`, value types (edges `types` connected to proxy `ValueType`) and `Compositions`. Here, we can see FRAGMENTA's proxy mechanism at play (proxies are represented as tick-lined boxes) to refer to elements defined in other fragments; `F_ASD` is a continuing fragment (symbol © in top-left corner), meaning that it is continued by other fragments, which will define the proxies used in `F_ASD`.

Fragment of Figure 5(b) describes the primitive types of SysML/INTO-CPS. Abstract class `PType` represents a primitive type, subdivided into reals (`Real`), integers (`Int`, which is subclassed by natural numbers, `Nat`), booleans (`Bool`) and `String`.

Figure 5(c) presents the fragment that describes properties:

- At the top of the inheritance hierarchy we have `NamedElement`, an abstract class representing a model element that has a `name`.
- Class `Property` specialises `NamedElement`; it has a `type` and an initial value (`init`). A `Property` is then subclassed by `Variable` and `FlowPort`.



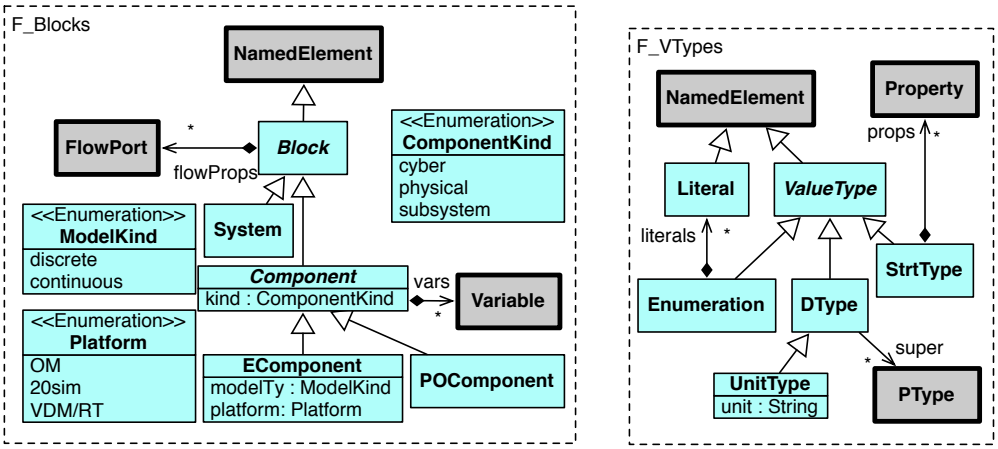
(a) ASD (b) Primitive types (c) Properties

Figure 5: Metamodel fragments of SysML/INTO-CPS

- A `Variable` comprises a `kind` property, which can have values of enumeration `VariableKind` indicating the type of variable, as follows: `parameter` is a variable requiring an explicit initialisation that remains constant, `varInit` is a variable requiring an initialisation, and `varNoInit` is a variable not requiring an initialisation.
- A `FlowPort` represents a communication port and comprises properties `direction` to indicate kind of port — input (`in`) or output (`out`) as defined by enumeration `Direction` — and dependencies on other ports (`depends`), which apply to output ports only according to well-formedness constraint WF3 (Table 1).

Fragment of Figure 6(a) describes blocks or components. It is as follows:

- A `Block` (an abstract class) is either a `System` or a `Component`. It comprises several instances of `FlowPort` (proxy referring element of fragment of Figure 5(c)) defining specific communication ports owned by the block; such definitions show up under the section `flow ports` of a block's properties compartment. In Figure 3(a), blocks `Valve` and `WaterTank` have port definitions.
- There is at most one `System` instance in any given model (see *WF1* in table 1). In Figure 3(a), `WaterTanks` is the system instance.



(a) Blocks (b) Value Types

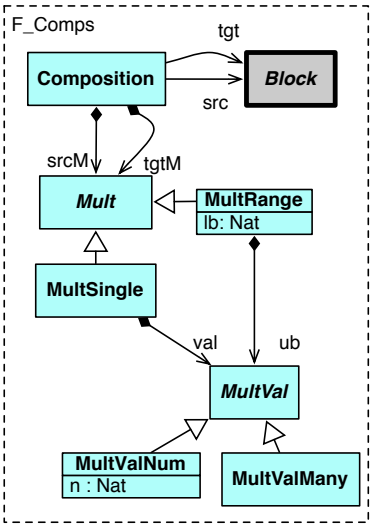
Figure 6: Metamodel fragments of SysML/INTO-CPS

- Enumeration `ComponentKind` captures the three kinds of `Components` supported by the profile: `cyber`, `physical` and `subsystem`. `Component` is an abstract class that is itself divided into encapsulating components (`EComponet`) that define a self-contained model, and part-of components (`POComponet`) that are part of a model of some encapsulating component.[1]
- An encapsulating component (`EComponet`) comprises a type of model (`discrete` or `continuous` as defined by enumeration `ModelKind`) and a `platform` (either `OpenModelica`, `20sim` or `VDM-RT` as defined in enumeration `PlatformKind`). `Components` of `kind subsystem` must be instances of `EComponet` (WF4 in table 1) and they must contain components of `kind cyber` and `physical` only (WF5 in table 1).

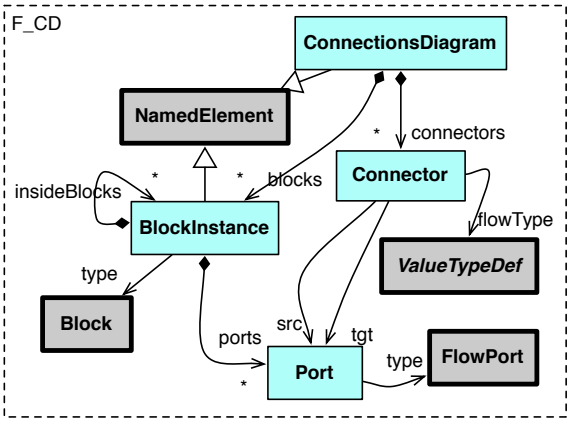
Figure 6(b) presents the fragments describing value types:

- `ValueType`, a subclass of `NamedElement`, is an abstract class that is divided into enumerations, derived types (`DType`) and structural types (`StrtType`).
- An `Enumeration` is defined as a collection of distinct literals (*WF2* in Table 1). A `DType` refers to the base primitive type (proxy `PType`; it is subclassed by `UnitType`, which comprises a measuring unit (such as "kg") – ASD of Fig. 3(a) defines the `FlowRate` derived type from the

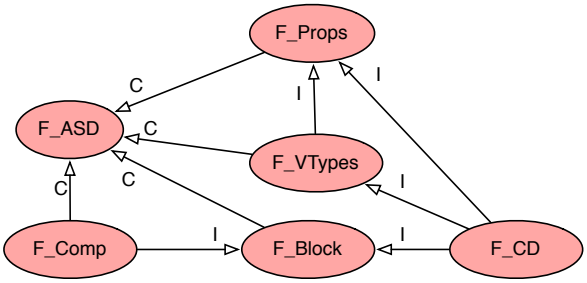
[1] In terms of the FMI, only `EComponents` result in FMUs.



(a) Compositions



(b) Connections Diagram



(c) GFG of SysML/INTO-CPS

Figure 7: Metamodel fragments of SysML/INTO-CPS

SysML `Real` primitive type.

- A `StrtType` comprises several value properties (represented by `Property`) — ASD of Figure 21 defines `StrtType` instances `Date` and `Time`.

The compositions metamodel fragment (Figure 7(a)) is as follows:

- `Compositions` deal with whole-part relationships between `Blocks`; they have source (`src`) and a target (`tgt`) block.
- `Compositions` have multiplicities (class `Mult`), which are subdivided into single multiplicities (`MultSingle` to hold a single value) and range multiplicities (`MultRange`). The actual values of multiplicities attached to composition relations are instances of class `MultVal`.

Figure 7(c) gives FRAGMENTA's global fragment graph (GFG) that depicts the relations between the different fragments that make the metamodel; these relations can be imports (symbol `I`) or continues (symbol `C`).

5.2 Connections Diagrams

The metamodel of INTO-CPS/SysML CDs is presented in Figure 7(b). Instances of this metamodel are given in Figure 3(b) and in the example ASDs of appendix B. The metamodel is as follows:

- A `ConnectionsDiagram` has a name (it is a `NamedElement`) and it comprises several instances of `blocks` connected through `connectors`. A `BlockInstance` is also `NamedElement` and refers to a block defined in the ASD (its `type`). In Figure 3(b), the block instance named `TC1` refers to the ASD type `TanksControl1`.
- A `BlockInstance` can enclose several other block instances (role `insideBlocks`). In Figure 3(b), block instance `TC1` encloses block instances `V` and `WT1`.
- A `BlockInstance` can have several `ports`. A `Port` has a `name`, corresponding to one of the block properties of the corresponding block in the ASD. In Figure 3(b), `V` has ports `valveI` and `flowO`.
- `Connectors` carry a `type` that corresponds to the information that is carried through the connected ports. This type must be consistent with the connected ports. In Figure 3(b), we say that the connector from the port of `V` to the port of `WT1` carries information of type `FlowRate`.

6 Semantics

This section presents the semantics of the SysML/INTO-CPS profile, which is expressed in the formal language CSP [Hoa85].

The semantics uses a version of CSP that is amenable to an analysis with the FDR3 refinement-checker [GRABR14]. A consequence of this is that we loose some precision because we need to represent reals as integers. This constraint is to be removed in the next version of the semantics, which is to be expressed in INTO-CPS, which has support for reals.

We start by giving a brief overview of CSP. Then, we present the structures that define the main constructs of a generic semantics (section 6.2) and then we instantiate these constructs to describe the semantics of our running example of three cascading water tanks of section 3 (section 6.3). Further illustrations of the semantics are provided in appendix B to accompany the SysML/INTO-CPS models of the case studies presented there.

6.1 CSP

Communicating Sequential Processes (CSP) [Hoa85, Sch00, Ros10b], a formal specification language introduced by Hoare [Hoa85] that is part of a class of languages that are known as *process algebras*, aims at describing communicating processes and interaction-driven computations.

CSP's domain of discourse consists of processes, which are self-contained components with particular interfaces through which they interact with their environment. The interface of a process is described as a set of events, which describe atomic, indivisible and instantaneous actions. A process is, therefore, characterised by the events it can engage in and their ordering. CSP is supported by an underlying theory to enable reasoning and model analysis about interaction and communication in this event-based model of interaction.

In CSP, events are transmitted along *communication channels*, which carry messages of particular types. A channel has a set of associated events, corresponding to all messages that may be carried through the channel.

Process expressions are built out of events using a number of operators:

- *Event prefixing*, expressed in CSP as $e \rightarrow P$, describes a process that expects event e and then behaves as process P.
- *Interleaving*, described in CSP as $P_1 \ ||| \ P_2$, defines a composition of two processes that execute in parallel without any synchronisation. The iterated version of interleaving, applies interleaving to any number of indexed processes: $\Big|\Big|\Big| \, i : N \bullet P(i)$.
- *Parallel*, $P_1 \underset{A}{\parallel} P_2$, describes the composition of two processes that execute in parallel synchronising on the set A of events.
- *Sequential*, $P_1; \ P2$, describes a process that executes P_1 until it terminates, and then executes P_2.

- *Hiding*, $P \setminus N$, makes a set N of events internal to a process P.
- *Interrupt*, $P_1 \triangle P_2$, defines a composition that behaves like P_1, but can be interrupted by a synchronisation on one of the initial events of P_2, which then takes over.
- *Throw*, $P_1 \underset{A}{\ominus} P2$, a relatively recent CSP operator [Ros10a], defines a form of interrupt where any occurrence of an event $e \in A$ within P_1 hands control to P_2.

Every CSP process P has an alphabet αP. Its semantics is given using four models: traces, failures, divergences and infinite traces. These are understood as observations of possible executions of the process P, in terms of the events from αP that it can engage in, refuse, or lead to divergence.

6.2 Semantics Structures

This section represents the semantics in terms of generic structures that are to be instantiated for each specific example.

The components or subsystems introduced at the level of SysML/INTO-CPS are represented as CSP processes that are initialised, receive inputs, transmit outputs and respond to simulation steps. The next channels support initialisation and step execution:

$$\textbf{channel}\ initSys, doStepSys$$

We define the external choice of a set of events, which yields $SKIP$ when the set is empty:

$$ExtChoiceEvents(evs) =$$
$$\quad \text{if } empty(evs) \text{ then } SKIP \text{ else } \Box\, e : evs \bullet e \rightarrow SKIP$$

The interleaving of a set of events is defined as:

$$InterleaveEvents(evs) = ||| \ e : evs \bullet e \rightarrow SKIP$$

We define a function that yields a process to initialise a sequence of outputs with the given values.

$$initOfOuts(\langle\rangle) = SKIP$$
$$initOfOuts(\langle o.v\rangle \frown initos) = o.v \rightarrow initOfOuts(initos)$$

The semantics treats components or subsystems as black boxes. It takes into account the fact that the current version of the SysML/INTO-CPS profile identifies the components that make up a design, describing how they are connected, but says very little about their actual behaviour. All the profile's diagrams say is that components receive specific inputs, produce specific outputs, have some state variables and are connected with other components. This is reflected in the definitions of the components as processes, which identify these known pieces of information, but abstracting from the unknown.

The CSP definition of a component is split into *local* and *composite*: local specifies the internal processing of the component in terms of the inputs, outputs and step, and composite says how the parts are joined to make the overall component behaviour.

The local portion of a component is described by the following process:

$$
\begin{array}{l}
System0(ins, outs, initOuts) = \\
\quad \text{let} \\
\qquad Init = initSys \rightarrow initOfOuts(initOuts);\ Inputs \\
\qquad Inputs = ExtChoiceEvents(ins);\ Step \\
\qquad Step = doStepSys \rightarrow Outputs \\
\qquad Outputs = ExtChoiceEvents(outs);\ Inputs \\
\quad \text{within} \\
\qquad Init
\end{array}
$$

A local component receives as parameters a set of input (ins) and output ports ($outs$), and a sequence of output initialisations ($initOuts$). The process description is subdivided into: *initialisation*, *inputs*, *step* and *outputs*. The initialisation ($Init$) expects the event $initSys$, then it initialises the outputs that require an initialisation ($initOfOuts$), and proceeds with the inputs. The processing of inputs ($Inputs$) expects values in the input ports (parameter ins) described as an external choice of all events and then proceeds with a step. The step ($Step$) expects a $doStepSys$ event and proceeds with the outputs. The processing of outputs ($Outputs$) does an external choice of all the outputs and then proceeds with the inputs. The local behaviour of a system ($System_0$) starts with the initialisation.

A composite component needs to put together local and composite behaviours. It takes a set of input ports (ins), a set of output ports ($outs$), a sequence of output initialisations ($initOuts$), a set of initialisation events of the component's parts ($initEvs$), a set of step events of the component's parts ($doStepEvs$)

and a process describing the internal composition of the component (*IntComp*):

$$
\begin{array}{l}
SystemComposite(ins, outs, initOuts, initEvs, doStepEvs, IntComp) = \\
\quad \text{let} \\
\quad\quad EvLinker(initEvs, doStepEvs) = \\
\quad\quad\quad \text{let} \\
\quad\quad\quad\quad Init = initSys \rightarrow InterleaveEvents(initEvs);\ Step \\
\quad\quad\quad\quad Step = doStepSys \rightarrow InterleaveEvents(doStepEvs);\ Step \\
\quad\quad\quad \text{within} \\
\quad\quad\quad\quad Init \\
\quad\quad SysLinked = System0(ins, outs, initOuts) \\
\quad\quad\quad\quad\quad \underset{\{|initSys, doStepSys|\}}{\|} EvLinker(initEvs, doStepEvs) \\
\quad \text{within} \\
\quad\quad (SysLinked \underset{initEvs \cup doStepEvs \cup ins}{\|} IntComp) \setminus (initEvs \cup doStepEvs)
\end{array}
$$

The internal process *EvLinker* states the dependencies between initialisation and step in the component and its parts. An initialisation in the component (*initSys*), is followed by an interleaving of initialisations in its parts; likewise for the step (*Step*). Internal process *SysLinked* does the parallel composition of the local portion of the component ($System_0$) with *EvLinker*, synchronising on the events of the component, so that initialisation and step on the component is followed by initialisation and step on the parts. The actual composite process is defined as the parallel composition of *SysLinked* and process *IntComp* with a synchronisation on the internal initialisation and step events, and input ports. The internal initialisation and step events are then hidden.

6.3 Running Example

We illustrate the CSP semantics with the SysML/INTO-CPS ASD and CD for the cascading water tanks example introduced in section 3 and which are given in Figure 3.

The first CSP snippet defines the types of the ASD (Fig. 3(a)). Enumeration `OpenClosed` is represented as a CSP datatype. All the remaining value types, derived from the reals, are represented in this version of CSP as inte-

gers.

$$\textbf{nametype}\ \mathit{Real} = \mathit{Int}$$
$$\textbf{datatype}\ \mathit{OpenClosed} = \mathit{open} \mid \mathit{closed}$$
$$\textbf{nametype}\ \mathit{FlowRate} = \mathit{Real}$$
$$\textbf{nametype}\ \mathit{Area} = \mathit{Real}$$
$$\textbf{nametype}\ \mathit{Height} = \mathit{Real}$$

We introduce a type to represent the indexing of `WaterTank` instances, reflecting the CD (Fig. 3(b)), which states that there are three such instances:

$$\textbf{nametype}\ \mathit{WaterTanksIx} = \{1 \mathinner{..} 3\}$$

The CSP specification turns to the definition of channels that describe the ports of the ASD; each flow port of each ASD component (Fig. 3(a)) has a corresponding channel:

$$\textbf{channel}\ w : \mathit{FlowRate}$$
$$\textbf{channel}\ \mathit{win}, \mathit{wout} : \mathit{WaterTanksIx}.\mathit{FlowRate}$$
$$\textbf{channel}\ v1, v2 : \mathit{OpenClosed}$$

Above, the definitions specify the types of the values transmitted through the channel. For example, channel `w` carries values of type `FlowRate`. The channels of `WaterTank` carry two values: one from `WaterTanksIx` indicating the water tank instance and another from `FlowRate`.

We now define the channels corresponding to meaningful events in the execution or simulation of a system's components. We consider that there are two such events: *initialisation* and *step execution*.

$$\textbf{channel}\ \mathit{initWaterTank} : \mathit{WaterTanksIx}$$
$$\textbf{channel}\ \mathit{doStepWaterTank} : \mathit{WaterTanksIx}$$
$$\textbf{channel}\ \mathit{initValve}, \mathit{doStepValve}$$
$$\textbf{channel}\ \mathit{initTanksControl}1, \mathit{doStepTanksControl}1$$
$$\textbf{channel}\ \mathit{initTanksControl}2, \mathit{doStepTanksControl}2$$
$$\textbf{channel}\ \mathit{initController}, \mathit{doStepController}$$
$$\textbf{channel}\ \mathit{initWaterTanksSys}, \mathit{doStepWaterTanksSys}$$

Above, the channel definitions of components with more than one instance (`WaterTank`) include the indexing type to identify the appropriate instance.

The different components are specified as CSP processes by instantiating the appropriate semantic structures. The CSP process defining the `Valve` takes into account that it is an atomic component:

$$Valve = System0(\{|v2|\}, \{|w|\}, \langle\rangle)[\![initSys, doStepSys / initValve, doStepValve]\!]$$

This customises the general $System0$ for `Valve` using renaming.

The `WaterTank` is defined similarly. The process's parameter represents the fact that it has several instances:

$$WaterTank(i) = System0(\{|win.i|\}, \{|wout.i|\}, \langle\rangle)[\![initSys, doStepSys / initWaterTank.i, doStepWaterTank.i]\!]$$

To represent the composite `TanksControl1`, we start by representing the internal configuration of this component as described in the CD (Fig. 3(b)), stating how the sub-components are wired through the flow ports:

$$TanksControl1Comp = Valve \underset{\{|w|\}}{\|} WaterTank(1)[\![win.1 / w]\!]$$

Above, the wiring of the connected ports is specified using the CSP renaming operator; $win.1$ of process $WaterTank(1)$ is renamed to w of $Valve$.

The process $TanksControl1$ is defined as a composite component:

$$TanksControl1 = SystemComposite(\{||\}, \{||\}, \langle\rangle, \{|initValve, initWaterTank.1|\}, \{|doStepValve, doStepWaterTank.1|\}, TanksControl1sComp)[\![initSys, doStepSys / initTanksControl1, doStepTanksControl1]\!] \setminus \{|w|\}$$

The renaming customises the general $SystemComposite$ to the purpose of $TanksControl1$; once $Valve$ and $WaterTank$ are wired the channel w is hidden so that it becomes an internal channel not visible to the outside world.

The internal configuration of `TanksControl2`, as described in the CD (Fig. 3(b)), is described as the following CSP process:

$$TanksControl2Comp = WaterTank(2) \underset{\{|wout.2|\}}{\|} WaterTank(3)[\![win.3 / wout.2]\!]$$

The composite component `TanksControl2` is defined as:

$$TanksControl2_0 = SystemComposite(\{||\}, \{||\}, \langle\rangle, \{|initWaterTank.2, initWaterTank.3|\}, \{|doStepWaterTank.2, doStepWaterTank.3|\}, TanksControl2Comp)[\![initSys, doStepSys / initTanksControl2, doStepTanksControl2]\!] \setminus \{|wout.2|\}$$

`Controller` is an atomic component:

$$Controller = System0(\{|\}, \{|v1|\}, \langle v1.closed \rangle) \\ [\![{}^{initSys,\ doStepSys}/_{initController,\ doStepController}]\!]$$

Above, the initial value of output port $v1$ is set in the initialisation, as described in the ASD (Fig. 3(a))

Finally, the overall system is specified as a composite component:

$$WaterTanksSysComp = Controller[\![{}^{v1}/_{v2}]\!] \\ \underset{\{|v2|\}}{\|} (TanksControl1 \underset{\{|wout.1|\}}{\|} TanksControl2[\![{}^{win.2}/_{wout.1}]\!])$$

$$WaterTanksSys = SystemComposite(\{|\}, \{|\}, \langle\rangle, \\ \{|initTanksControl1, initTanksControl2, initController|\}, \\ \{|doStepTanksControl1, doStepTanksControl2, doStepController|\}, \\ WaterTanksSysComp) \\ [\![{}^{initSys,\ doStepSys}/_{initWaterTanksSys,\ doStepWaterTanksSys}]\!] \{|wout.1, wout.3, v2|\}$$

7 The INTO-CPS profile in the Modelio Tool

This section gives some details on Modelio's implementation of the profile presented here. The SysML/INTO-CPS diagrams of this deliverable's running example drawn using the current version of Modelio's implementation are given in appendix A.

The following starts by describing Modelio's extension mechanisms, and then shows how Modelio has been extended to accommodate SysML/INTO-CPS.

7.1 Modelio

Modelio is an Open Source MDE workbench tool, which supports UML2.x and BPMN 2.0 standards. It provides an implementation of the UML2.x profile mechanism, which makes Modelio extendable and capable of accommodating other profiles by customising the constructs provided by UML2.x or BPMN2.0. Figure 8 presents part of the metamodel of UML2.x that underpins Modelio's UML2.x-based extensions.

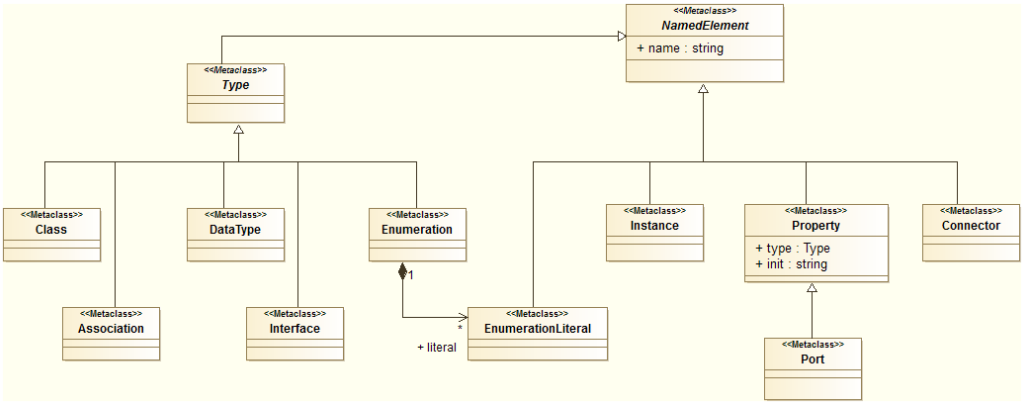


Figure 8: Some UML2.x Metaclasses

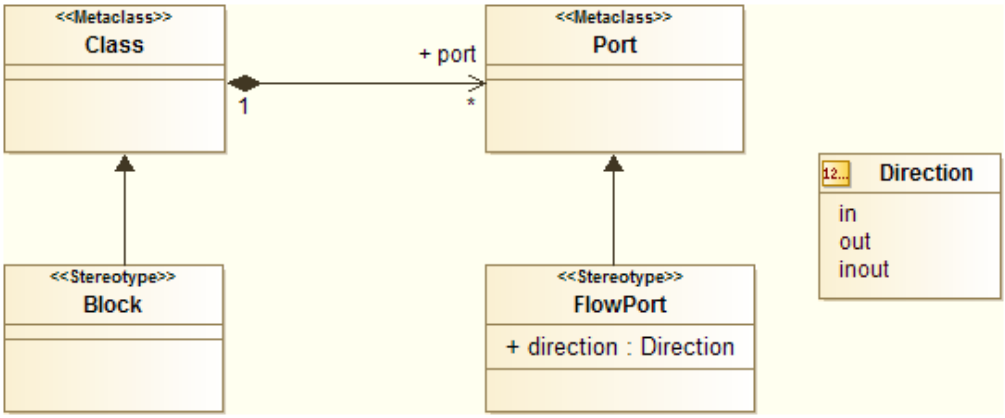


Figure 9: SysML Block and FlowPort stereotypes. (Black-pointed arrows denote stereotype extension.)

Two extensions supported by Modelio, SysML and MARTE, dedicated, respectively, to systems and real time modelling, are the closest to the INTO-CPS modelling domain. Figure 9 illustrates Modelio's extension approach: UML metaclasses `Class` and `Port` are extended by SysML `Block` and `FlowPort` stereotypes, respectively.

7.2 SysML/INTO-CPS Modelio profile

The modelio SysML/INTO-CPS extension is organised around the following logical groups: *block, type, instance, library and diagram.*

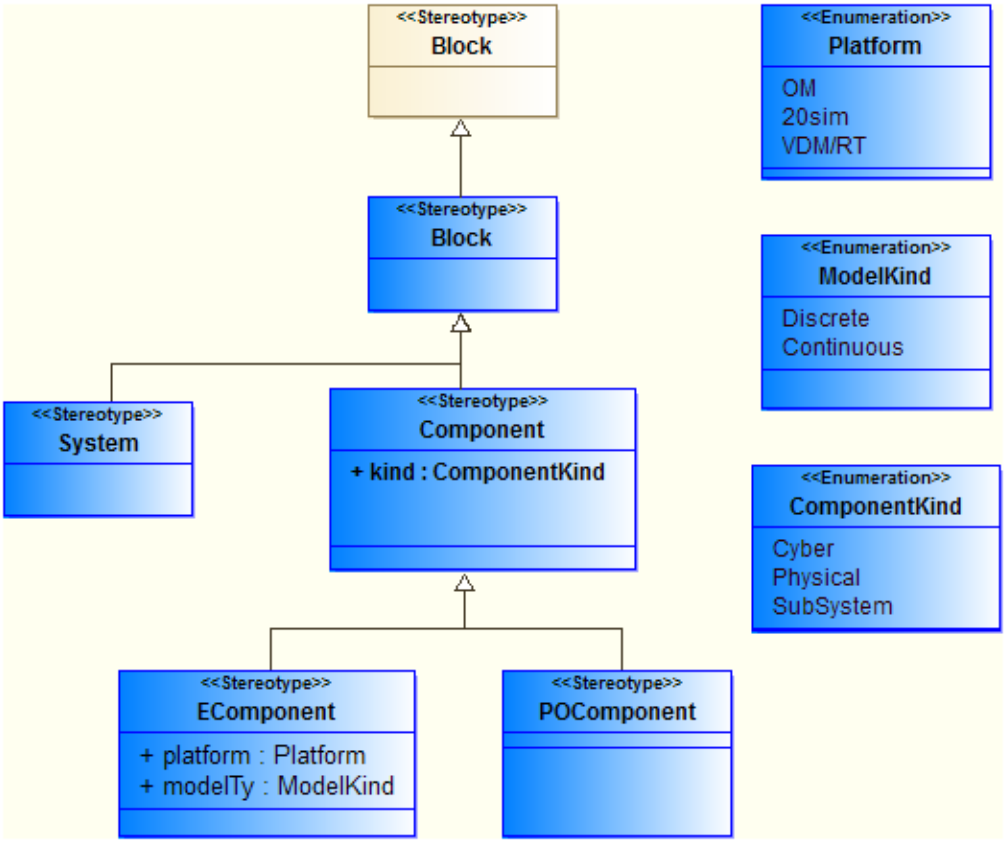


Figure 10: The different kind of INTO-CPS blocks

Figure 10 presents the block group, which realises the `Blocks` metamodel fragment of Figure 6(a). This is based on specialising the SysML `Block` stereotype, which enables us to incorporate the metamodel elements of the fragment that is given Figure 6(a).

Figure 11 presents a realisation of the metamodel fragments of Figures 6(a), 5(c) and 7(a). It describes the inheritance relations that exist between `Block`, `System`, `Component`, `EComponent` and `POComponent`. As defined in the INTO-CPS metamodel, a Block can be composed of several `FlowPorts` and `Variables` (Figure 6(a)). It can also be the source (src) and the target (trt) of `Composition`. An INTO-CPS Composition extends the UML2.x Association metaclass, and then inherits its multiplicities mechanism, which is represented in the metamodel (Figure 7(a)) but not in Modelio's profile definition.

The *types* group of Figure 12 realises the metamodel fragment of Figure 6(b).

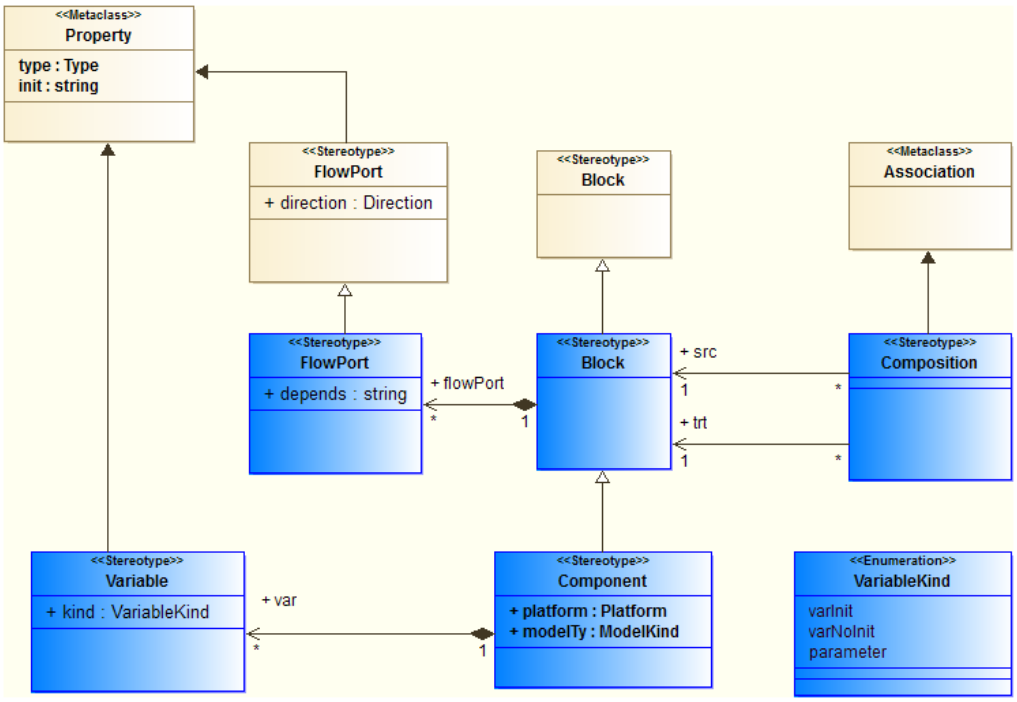


Figure 11: The INTO-CPS block structure

It comprises several stereotypes that are mainly used to type FlowPorts and Variables presented in Figure 11 but also the Connections defined at the instance level (Figure 14). Figure 13 shows how the different type metaclasses of the profile extend the UML2.x metaclasses.

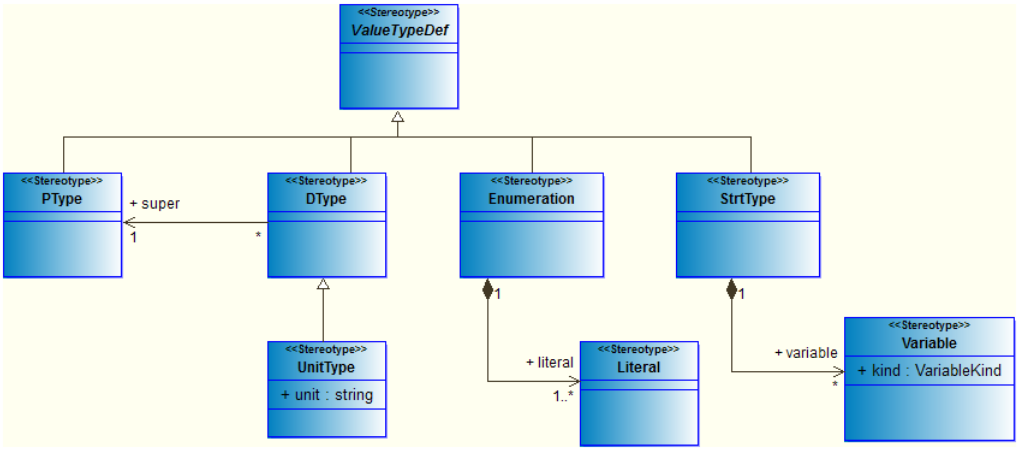


Figure 12: The INTO-CPS Types specification

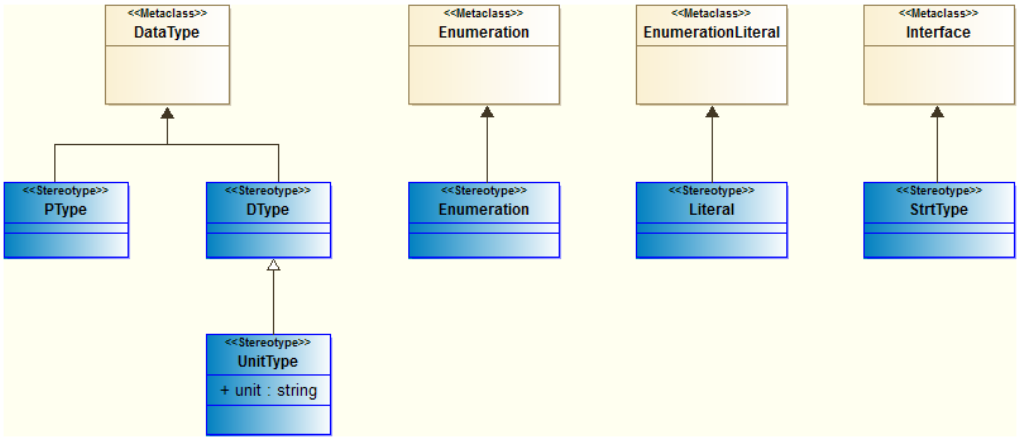


Figure 13: The INTO-CPS Types extensions

UML distinguishes between the class or type level and the instance or object level. This distinction also takes place in the INTO-CPS profile; with the block and type stereotype groups on one hand, and instances group on the other hand, which is presented in Figure 14. The instances group realises the metamodel fragment corresponding to connections diagrams (Figure 7(b)). Here, `Block` and `FlowPort` concepts are instantiated by `BlockInstance` and `Port`, respectively. A connection between ports can be set thanks to the `Connector` stereotype which can be typed with a reference to a ValueType.

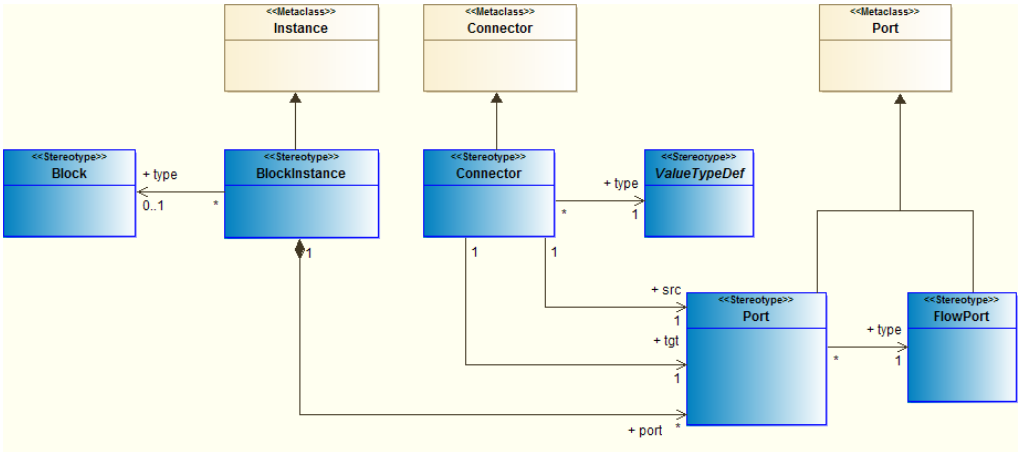


Figure 14: The INTO-CPS instance relationship

The INTO-CPS metamodel also define new primitive types in the meta-

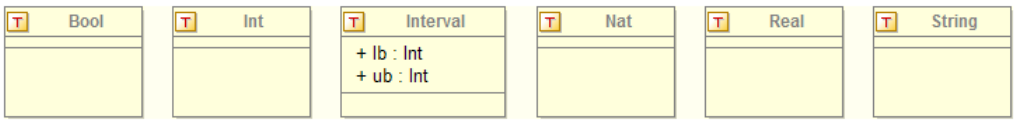


Figure 15: INTO-CPS primitive type library

model fragment of Figure 5(b). UML has four primitive types, `Integer`, `Boolean`, `String`, and `UnlimitedNatural`; INTO-CPS adds a `Real` and `Interval` primitive types as shown in Figure 15.

The INTO-CPS profile defines two kinds of diagrams ASDs and CDs. As shows in Figure 16, these diagram types extend UML Class and Object diagrams. To support the creation of blocks and types of an ASD Figure 17, the ASD editor in Modelio includes two palette groups. Instance concepts have been grouped in the CD editor (an example is given in Figure 18).

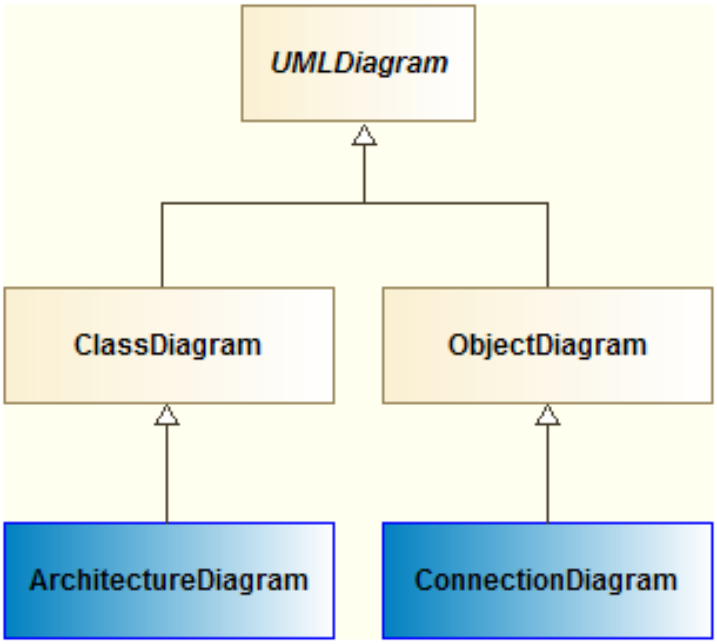


Figure 16: INTO-CPS diagrams

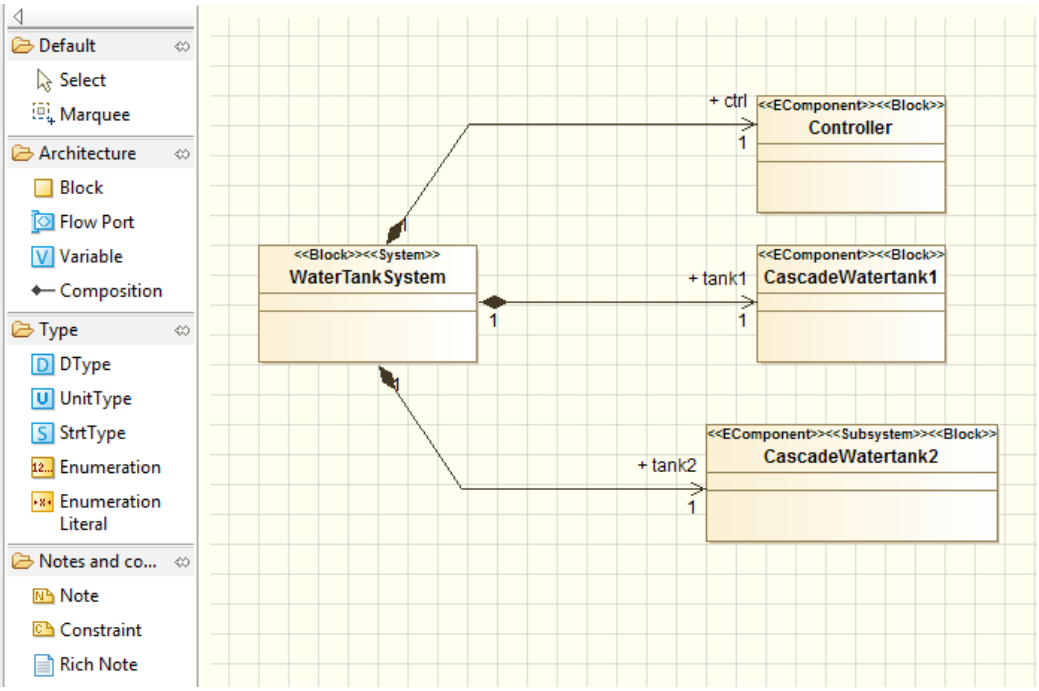


Figure 17: A Block definition inside an INTO-CPS architecture diagram

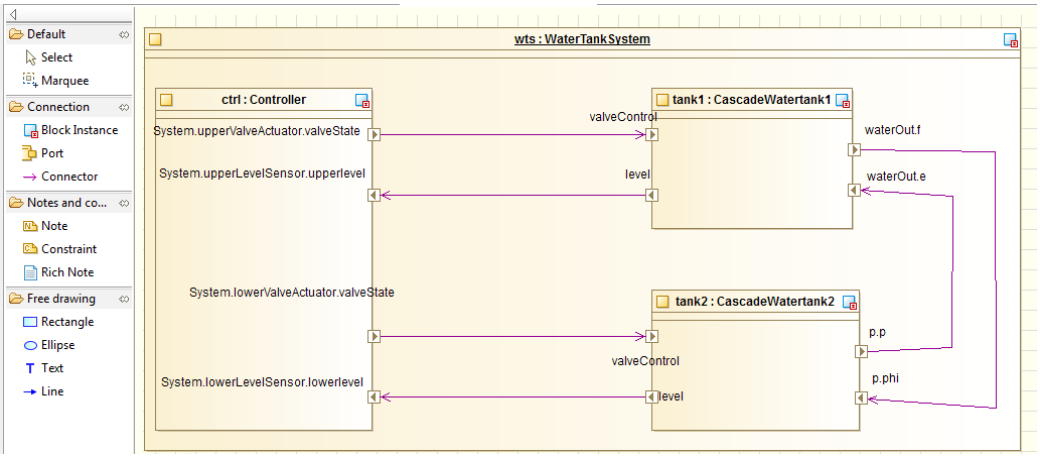


Figure 18: Connection diagram example

8 Conclusions

This reports describes the efforts of INTO-CPS's WP2 on the foundations of SysML for cyber-physical systems (CPS). The report surveys the literature in the area of SysML for CPS with an emphasis on semantic foundations, and

presents the SysML/INTO-CPS profile, to be further developed and used in the context of the INTO-CPS project, by providing the profile's syntactic definitions based on UML class metamodels and the profile's semantics expressed in the CSP process algebra.

The report illustrated visual modelling based on the profile's metamodels and the semantics with several examples. The semantic foundations of the SysML/INTO-CPS profile presented here denote an underlying UTP model based on the CSP's UTP semantics.

The work presented here paves the way to an integration of the profile with the functional mock-up interface (FMI) [FMI14] based on formal semantic foundations, as both the SysML/INTO-CPS profile and FMI [ACWK15] have been given a CSP semantics with an underlying UTP model. This allows us to establish a refinement relation between the more abstract SysML phenomena and the more concrete FMI co-simulations, allowing us to reason about properties at the more abstract SysML level that are preserved at the level of the FMI.

In the year 2 of INTO-CPS, we expect to introduce more diagram types and to exploit the formal semantics presented here for the purpose of verification and validation of SysML/INTO-CPS models.

9 References

[Abr96] Jean-Raymond Abrial. Steam-boiler control specification problem. In *Formal Methods for Industrial Applications*, 1996.

[ACWK15] Nuno Amálio, Ana Cavalcanti, Jim Woodcock, and Christian König. Foundations for FMI co-modelling. Technical Report D2.1d, INTO-CPS project, 2015.

[AdLG15] Nuno Amálio, Juan de Lara, and Esther Guerra. FRAGMENTA: A theory of fragmentation for MDE. In *MODELS 2015*. IEEE, 2015.

[AP03] Nuno Amálio and Fiona Polack. Comparison of formalisation approaches of UML class constructs in Z and object-Z. In *ZB 2003*, LNCS. Springer, 2003.

[BJ11] Keyla Brasil and Diógenes Silva Junior. Automatic translation of SysML models to systemc executable specification. In *WCAS*, 2011.

[BOA+11] Torsten Blochwitz, Martin Otter, M. Arnold, C. Bausch, C. Clauß, Hilding Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Modelica Conference*, 2011.

[BOA+12] 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. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Modelica Conference*, 2012.

[BTRB12] Thouraya Bouabana-Tebibel, Stuart H. Rubin, and Miloud Bennama. Formal modeling with SysML. In *Int Conf Reuse Integration*. IEEE, 2012.

[CdSH+13] Daniel Chaves Café, Filipe Vinci dos Santos, Cécile Hardebolle, Christophe Jacuet, and Frédéric Boulanger. Multi-paradigm semantics for simulating SysML models using SystemC-AMS. In *FDL 2013*. IEEE, 2013.

[CH11] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on SysML and interface automata.

Innovations in Systems and Software Engineering, 7(4):265–274, 2011.

[CSW03] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A refinement strategy for Circus. *Formal aspects of computing*, 15(2–3):146–181, 2003.

[dAH01] Luca de Alfaro and Thomas Henzinger. Interface automata. In *ESEC/FSE 2001*. ACM, 2001.

[DLV12] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber-physical systems. *Proceedings of IEEE: special issue on CPS*, 100(1):13–28, 2012.

[DT10] Song Ding and Sheng-Qun Tang. An approach for formal representation of SysML block diagram with description logic SHIOQ(D). In *Industrial and Information systems*. IEEE, 2010.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In *UML'98*, volume 1618 of *LNCS*, pages 336–348. Springer, 1998.

[FGDT06] Robert France, Sudipto Ghosh, and Trung Dinh-Trong. Model-driven development using UML 2.0: Promises and pitfalls. *IEEE Computer*, 39(2):59–66, 2006.

[FGP14] Yishai A. Feldman, Lev Greenberg, and Eldad Palachi. Simulating rhapsody SysML blocks in hybrid models with FMI. In *Modelica Conference*, 2014.

[FMI14] FMI development group. Functional mock-up interface for model exchange and co-simulation, 2.0. `https://www.fmi-standard.org`, 2014.

[GB11] Henson Graves and Yvonne Bijan. Using formal methods with SysML in aerospace design and engineering. *Annals of Mathematics and Artificial Intelligence*, 63(1):53–102, 2011.

[GRABR14] 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.

[Hen96] Thomas Henzinger. The theory of hybrid automata. In *LICS'96*, 1996.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IEE12] IEEE. IEEE 1666 standard SystemC language reference manual. `http://dx.doi.org/10.1109/IEEESTD.2012.6134619`, 2012.

[Jif94] He Jifeng. From CSP to hybrid systems. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, 1994.

[Lee03] Edward A. Lee. Model-driven development – from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded systems*, 2003.

[MB11] María Victoria Cengarle Manfred Broy. UML formal semantics: lessons learned. *Software and Systems Modeling*, 10(4):441–446, 2011.

[MC14] Alvaro Miyazawa and Ana Cavalcanti. Formal refinement in SysML. In *IFM 2014*, LNCS. Springer, 2014.

[MLC13] Alvaro Miyazawa, Lucas Lima, and Ana Cavalcanti. Formal models of SysML blocks. In *ICFEM*, LNCS. Springer, 2013.

[OMG12] OMG. OMG systems modeling language, version 1.3. Technical report, OMG, 2012.

[RF11] Bernhard Rumpe and Robert France. Variability in UML language and semantics. *Software and Systems Modeling*, 10(4):439–440, 2011.

[Ros10a] A. W. Roscoe. CSP is expressive enough for π. In *Reflections on the Work of C.A.R. Hoare*. Springer, 2010.

[Ros10b] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[Sch00] Steve Schneider. *Concurrent and Real-Time Systems*. Wiley, 2000.

[Ste02] Perdita Stevens. On the interpretation of binary associations in the unified modelling language. *Software and Systems Modeling*, 1(1):68–79, 2002.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *ZB 2002*. Springer, 2002.

[WCF+12] Jim Woodcock, Ana Cavalcanti, John Fitzgerald, Peter Larsen, Alvaro Miyazawa, and Simon Perry. Features of CML: A formal modelling language for systems of systems. In *SoSE*. IEEE, 2012.

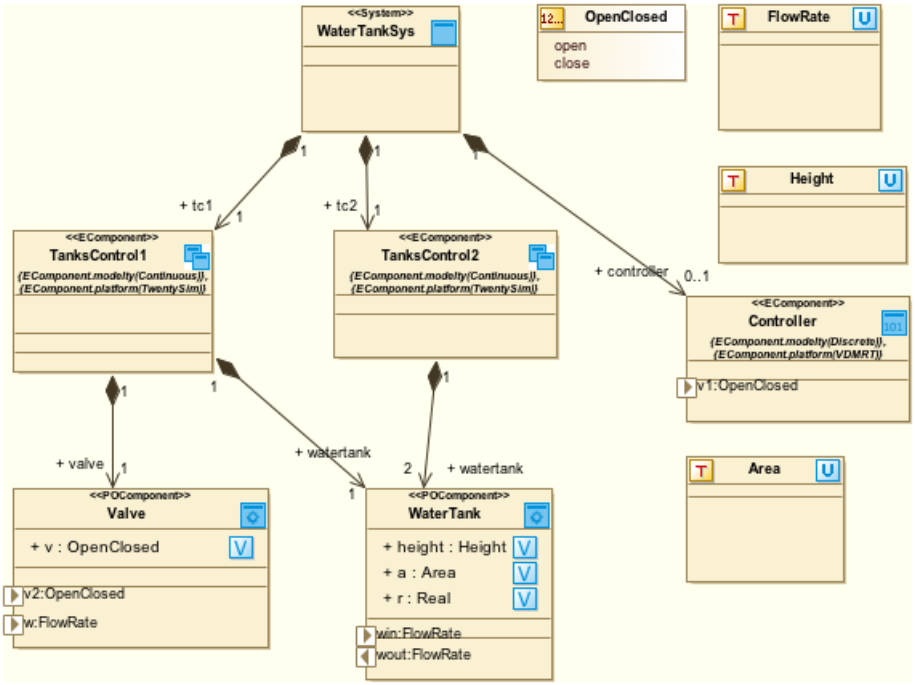
[WCMG15] Frank Wawrzik, William Chipman, Javier Moreno Molina, and Christoph Grimm. Modeling and simulation of cyber-physical systems with SICYPHOS. In *DTIS*, 2015.

[Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, 1996.

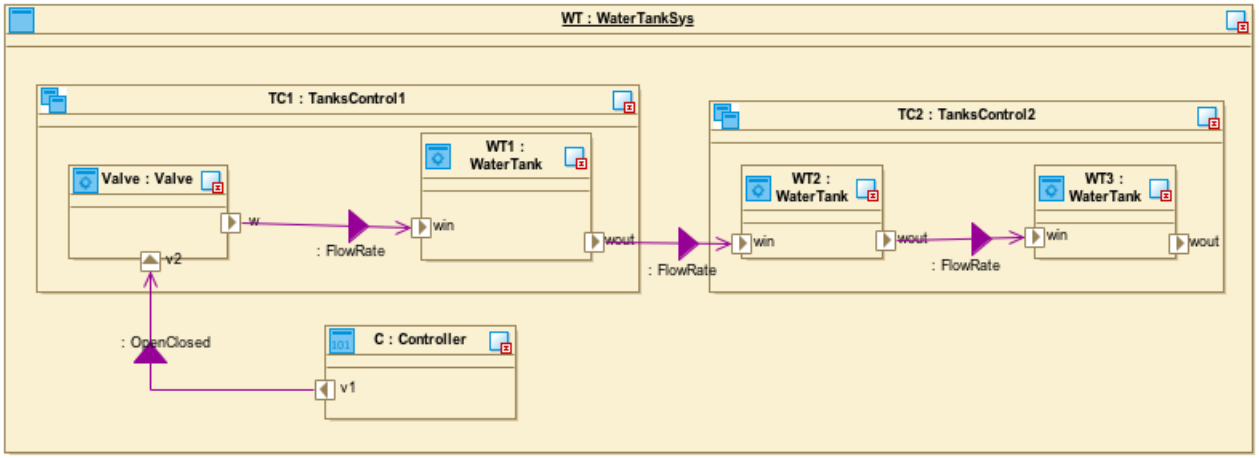
[WK03] Jos Warner and Anneke Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley, 2003.

A Modelio Diagrams

This appendix presents the SysML/INTO-CPS diagrams of this deliverable's running example (given section 4) as drawn in the current version of Modelio's implementation of the SysML/INTO-CPS profile.



(a) Architecture Structure Diagram



(b) Connections Diagram

Figure 19: The SysML/INTO-CPS architectural and connections diagrams of the *three cascading water tanks* system as drawn in Modelio

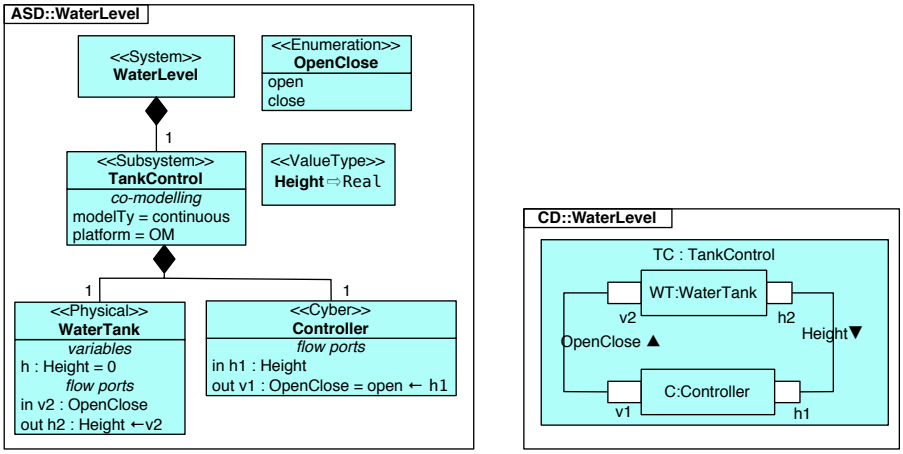
B Further Examples

B.1 Water level

This example is taken from He Jifeng's Hybrid-CSP paper [Jif94], a part of the larger steam-boiler case study [Abr96], the *Water Level* system controls the level of water in a water tank through a control valve that can be switched `on` or `off`.

B.1.1 SysML

Figure 20 presents the system's ASD and CD; the subsystem `TanksControl` governs the interaction between the physical `WaterTank` and the cyber-component `Controller`.



(a) Architectural Structure Diagram (b) Connections Diagram

Figure 20: The INTO-CPS SysML ASD and CD of the *Water Level* system.

B.1.2 CSP Semantics

The CSP definitions of the types defined in Fig. 20(a) are as follows:

$$\textbf{nametype}\ \mathit{Real} = \mathit{Int}$$
$$\textbf{nametype}\ \mathit{Height} = \mathit{Real}$$
$$\textbf{datatype}\ \mathit{OpenClosed} = \mathit{open} \mid \mathit{closed}$$

The CSP definitions of flow ports is as follows:

$$\textbf{channel}\ vi : OpenClosed$$
$$\textbf{channel}\ ho : Height$$
$$\textbf{channel}\ hi : Height$$
$$\textbf{channel}\ vo : OpenClosed$$

The CSP definitions of event channels is as follows:

$$\textbf{channel}\ initWaterTank, doStepWaterTank$$
$$\textbf{channel}\ initController, doStepController$$
$$\textbf{channel}\ initTanksControl, doStepTanksControl$$
$$\textbf{channel}\ initWaterLevel, doStepWaterLevel$$

The CSP definition of component `WaterTank` is as follows:

$$WaterTank = System0(\{|vi|\}, \{|ho|\}, \langle\rangle)$$
$$[\![initSys, doStepSys / initWaterTank, doStepWaterTank]\!]$$

The CSP definition of component `Controller` is as follows:

$$Controller = System0(\{|hi|\}, \{|vo|\}, \langle vo.open\rangle)$$
$$[\![initSys, doStepSys / initController, doStepController]\!]$$

The CSP definition of the internal composition of subsystem `TanksControl` is:

$$TanksControlComp = WaterTank[\![v2/v1]\!] \underset{\{|v1,h2|\}}{\|} Controller[\![h1/h2]\!]$$

The CSP definition of `TanksControl` is:

$$TanksControl = SystemComposite(\{||\}, \{||\}, \langle\rangle,$$
$$\{|initWaterTank, initController|\},$$
$$\{|doStepWaterTank, doStepController|\}, TanksControlComp)$$
$$[\![initSys, doStepSys / initTanksControl, doStepTanksControl]\!] \setminus \{|v1, h2|\}$$

The internal composition of `WaterLevel` is:

$$WaterLevelComp = TanksControl$$

The CSP definition of overall `WaterLevel` system is as follows:

$$
\begin{aligned}
&WaterLevel0 = SystemComposite(\{|\}, \{|\}, \langle\rangle,\\
&\quad \{|initTanksControl|\},\\
&\quad \{|doStepTanksControl|\}, WaterLevelComp)\\
&\quad [\![^{initSys,\ doStepSys}/_{initWaterLevel,\ doStepWaterLevel}]\!]
\end{aligned}
$$

B.2 Thermostat

This example is a variation of the *temperature control* case study that is given in [Hen96]. A user interacts with the system using a software controlled interface that enables switching the heating on and off, setting the current date and time, and desired room temperature.

B.2.1 SysML

A design of this system, highlighting two subsystems, `Heating` and `Controls`, is given in Fig. 21. `Controls` takes the thermostat's user settings, which are channeled to the `Heating` subsystem to purvey the desired room temperature.

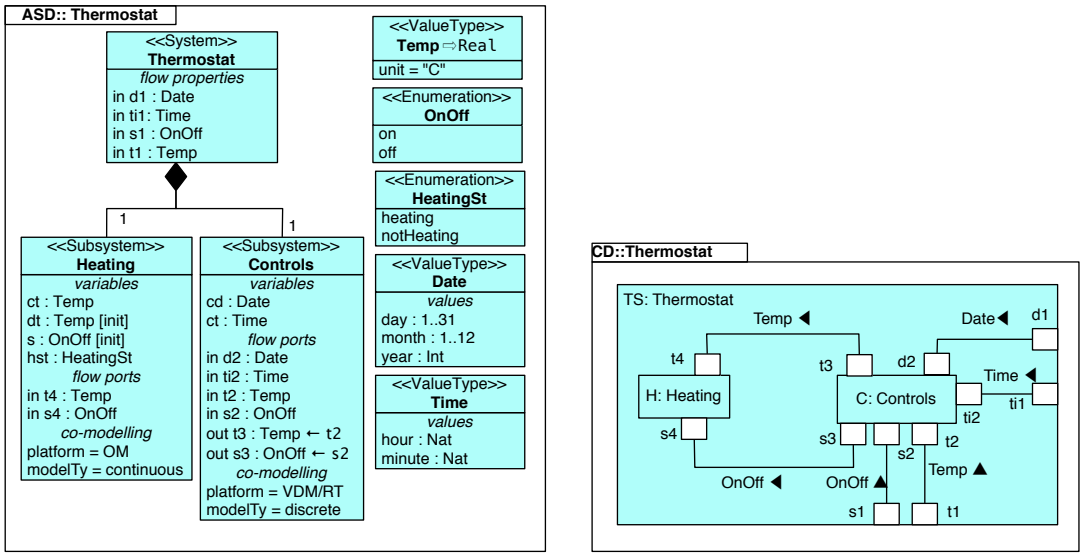
B.2.2 CSP Semantics

The CSP definition of the types of Fig. 21(a) are as follows:

$$
\begin{aligned}
&\textbf{nametype}\ Real = Int\\
&\textbf{nametype}\ Temp = Int\\
&\textbf{datatype}\ OnOff = on \mid off\\
&\textbf{datatype}\ HeatingSt = heating \mid notHeating\\
&\textbf{nametype}\ Date = (\{1\,..\,31\}, \{1\,..\,12\}, Int)\\
&\textbf{nametype}\ Time = (\{0\,..\,24\}, \{0\,..\,60\})
\end{aligned}
$$

The CSP definitions of flow ports is as follows:

$$
\begin{aligned}
&\textbf{channel}\ d1, d2 : Date\\
&\textbf{channel}\ ti1, ti2 : Time\\
&\textbf{channel}\ t1, t2, t3, t4 : Temp\\
&\textbf{channel}\ s1, s2, s3, s4 : OnOff
\end{aligned}
$$



(a) Architecture Structure Diagram (b) Connections Diagram

Figure 21: The INTO-CPS SysML ASD and CD of the *temperature control* system

The CSP definitions of sub-system events is as follows:

$$\textbf{channel}\ initHeating : Temp.OnOff$$
$$\textbf{channel}\ doStepHeating$$
$$\textbf{channel}\ initControls, doStepControls$$
$$\textbf{channel}\ initThermostat, doStepThermostat$$

Subsystem `Heating` is as follows:

$$Heating = System0(\{|t4, s4|\}, \{||\}, \langle\rangle)$$
$$[\![initSys, doStepSys / initHeating, doStepHeating]\!]$$

Likewise, for system `Controls`:

$$Controls = System0(\{|d2, ti2, t2, s2|\}, \{|t3, s3|\}, \langle\rangle)$$
$$[\![initSys, doStepSys / initControls, doStepControls]\!]$$

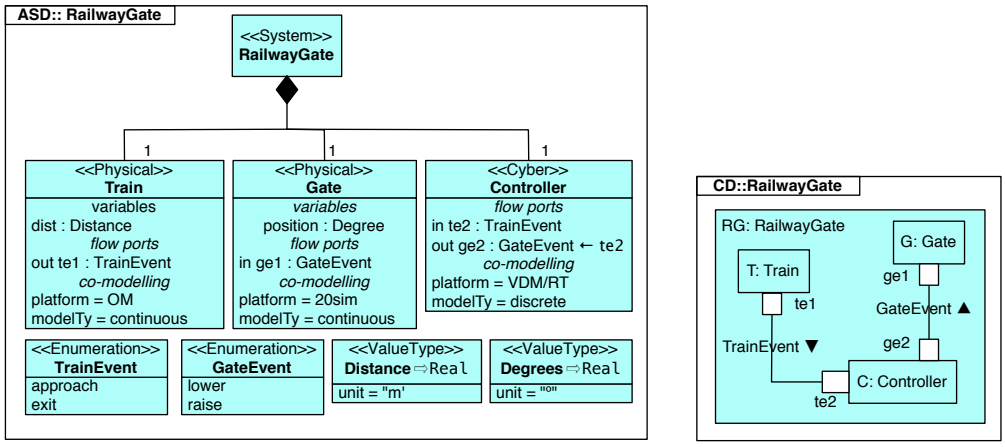
The internal composition of `Thermostat` is as follows:

$$ThermostatComp = (Heating[[\![t4, s4 / t3, s3]\!]\ \underset{\{|t3,s3|\}}{\|}\ Controls)$$
$$[\![ti2, d2, t2, s2 / ti1, d1, t1, s1]\!]$$

The overall system `Thermostat` is defined as follows:

$$Thermostat = SystemComposite(\{|d1, ti1, s1, t1|\}, \{||\}, \langle\rangle,$$
$$\{|initHeating, initControls|\},$$
$$\{|doStepHeating, doStepControls|\}, ThermostatComp)$$
$$[\![initSys, doStepSys / initThermostat, doStepThermostat]\!] \setminus \{|t3, s3|\}$$

B.3 Railway Gate



(a) Architectural Diagram (b) Connections Diagram

Figure 22: The INTO-CPS SysML ASD and CD of the *railway gate* system.

This example is taken from [Hen96]. The INTO-CPS/SysML model is given in Fig. 22.