Towards Graphical Configuration in the INTO-CPS Application

Christian Møldrup Legaard[1], Casper Thule[1], Peter Gorm Larsen[1]

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark,
201408498@post.au.dk, casper.thule@eng.au.dk, pgl@eng.au.dk

Abstract. The INTO-CPS Application is a common interface used to access and create different artefacts in the development of a Cyber-Physical System (CPS) making use of a collection of tools centred around the Functional Mockup Interface (FMI). For the configuration of the composition and adaptation of Functional Mockup Units (FMUs) the application currently imports this information as a multi-model from a SysML model made using the Modelio tool. The contribution presented in this paper is the addition of a unified graphical editor to the INTO-CPS application which eliminates the need for external tools. This is accomplished by using a standard called System Structure and Parameterisation (SSP) complementing FMI, since it is more expressive than the current multi-model representation. SSP enables the description of co-simulation scenarios in a graphical form including semantic adaptation of FMUs using SSP's extension capabilities.

1 Introduction

In Cyber-Physical Systems (CPSs), computing and physical processes interact closely. Their effective design therefore requires methods and tools that bring together the products of diverse engineering disciplines. Without such tools it would be difficult to gain confidence in the system-level consequences of design decisions made in any one domain, and it would be challenging to manage trade-offs between them.

The INTO-CPS project has created a tool chain supporting different disciplines such as software, mechatronic and control engineering that have evolved notations and theories tailored to their specific domains. It would be undesirable to suppress this diversity by enforcing uniform general-purpose models [3,4,11,12,13].

The configuration of co-simulations has, in the past, been carried out using a special CPS profile for the SysML support inside the Modelio tool [2]. The current SysML support also includes diagrams for Design Space Exploration (DSE) where different alternative designs can be explored automatically over different parameters [5]. However, the SysML CPS profile does not yet support hierarchical co-simulations [16] or other semantic adaptations [7]. It would be useful to examine if these concepts could be supported by a single unified editor inside the INTO-CPS application. We take inspi-

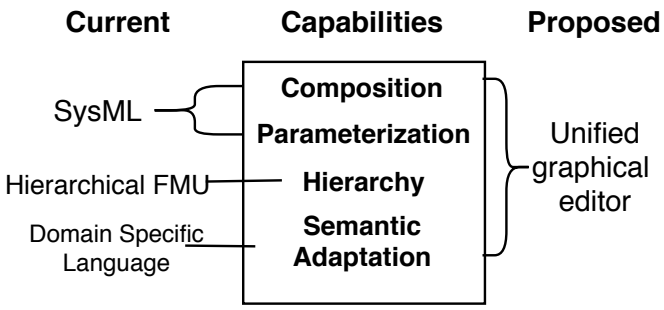

Fig. 1. The migration from current capabilities to the proposed graphical solution/representation.

ration from the modelling tools 20-sim[1], OMEdit[2] and Simulink[3] in order to come up with new ideas for this. Figure 1 depicts the tools currently involved to access different capabilities of the application, as well as the proposed graphical editor which unifies access to these.

The rest of this paper starts with background information for the reader in Section 2. Afterwards Section 3 illustrates how the graphical connections are made directly inside the INTO-CPS Application. Then Section 4 provides an introduction to the overall design of the graphical editor. Finally, Section 5 provides a few concluding remarks while Section 6 provides the future directions expected for this work.

2 Background

It is important to make a distinction between what the FMI standard defines and the information required to form a valid co-simulation scenario as illustrated on Figure 2. The standard is solely concerned with specifying the interface of an individual FMU. A co-simulation scenario describes what instances of FMUs exist, how they are connected, their parameters and their experimental frame [8]. In order to provide the reader with sufficient background about the existing technologies Section 2.1 briefly introduces the existing SysML diagrams to describe the composition of FMUs, Section 2.2 provides a brief introduction to separate existing work on semantic adaptation including hierarchical compositions and Section 2.3 provides a brief introduction to the new SSP standard.

2.1 Existing SysML Connection Diagram

A SysML Internal Block Diagram in the context of INTO-CPS is referred to as a *Connection Diagram*. It is used to define the composition and connectivity of several FMUs. The connection diagram contains instances of FMUs (SysML Block instances) which

[1] https://www.20sim.com/

[2] https://openmodelica.org/?id=78:omconnectioneditoromedit&catid=10:main-category

[3] https://se.mathworks.com/products/simulink.html

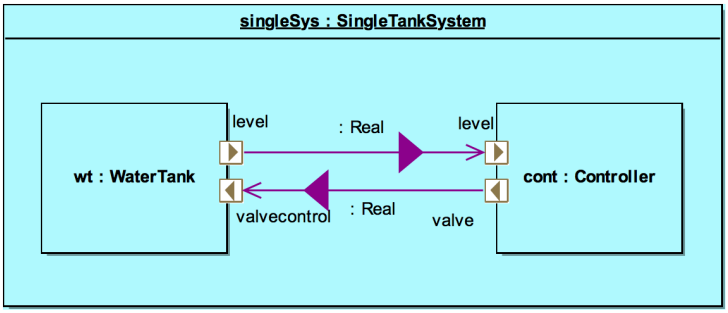

Fig. 2. Connection diagram for the WaterTank example to connect FMUs.

are connected together by means of connectors to the input/output ports of the FMU instances. The connection diagram also enables association of existing FMUs to the block instances for a co-simulation.

Using a general-purpose modeling language such as SysML for defining simulation scenarios, comes with an inherent challenge. The lack of simulation specific concepts in the language makes expression of these more cumbersome and error prone. First, it is required that the users memorise the exact way these concepts must be expressed in SysML. Secondly, it limits the assistance the tool can provide the user, due to its limited understanding of domain specific concepts. These issues are amplified by the fact that errors are only detected when the SysML model is imported in the tool.

In the DSE diagrams inside the SysML profile the exploration capabilities makes use of parameters for the different FMUs, so this is also important for the work we describe here.

2.2 Semantic Adaptations

While the FMI standard defines a common format for exchanging FMUs it does not guarantee the absence of so called *interaction mismatches* when these are composed [7]. The goal of semantic adaptations (SA), is to allow these mismatches to be corrected without the need to involve the original author of the FMU.

A concrete example where this would be if *valve* input of the *WaterTank* accepted a bool instead of a real. In this case it would not be possible to connect the *valvecontrol* and *valve* ports. A SA could be used to apply a threshold to the *valve* output in order to convert it into a bool, as seen in figure 3.

Conceptually, a SA can be thought of as an *external* FMU, which wraps around one or more *internal* FMUs. The external FMU is responsible for managing the execution of the internal FMUs. In the previous example the external FMU would simply pass all signals directly, except for the *valve*, which would be thresholded before being written to the *valve* output of the external FMU. The approach grants a great deal of flexibility, allowing a large number of adaptations to be implemented. In addition, it allows these to be applied in a hierarchical manner, e.g. "on top of" each other.

As described in Section 1 the SysML CPS profile does not yet support semantic adaptations [7]. In general such semantics adaptations are currently established by a

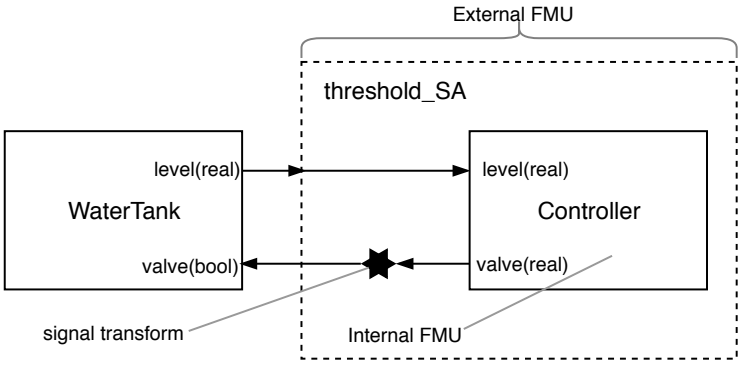

Fig. 3. Example of how semantic adaptation may be applied to correct signal data mismatch.

separate Domain Specific Language (DSL). This is introduced in order to perform different kinds of adaptations to existing FMUs that enable their behaviour to be adjusted in different ways. However, the DSL from [7] also enables support for hierarchical co-simulations. A different approach for hierarchical co-simulations has been introduced in [16].

2.3 System Structure And Parametrisation

The central goal of the INTO-CPS Application is to enable the composition and co-simulation of a system consisting of one or more FMUs. The use of the FMI standard enables exchange at the level of the individual FMUs. However, it does not describe an exchange format for composing and parametrising a simulation scenario consisting of multiple FMUs. Due to the lack of an established standard, the INTO-CPS Application uses its own ad-hoc format referred to as the *multi-model* format.

Recently a new companion standard to FMI was published: the *System Structure and Parameterization* (SSP) standard [14][4]. The standard uses a zip-based packaging format similar to FMI, where multiple artefacts are bundled into a single package referred to as a *SSP package* – each representing a ready to simulate system. An obvious benefit of adopting this standard is exchangeability of complete systems between tools. Currently only a few such as OpenModelica and *FMPy*[5], however given its close relation to FMI, it seems likely that more tools will adopt it. An example of what the running example may look like when bundled as a package is seen on Figure 4.

The standard covers the functionality present in the existing format, but also adds several capabilities which are useful for the INTO-CPS Application. Additionally, there are also some important semantic differences between the existing format and SSP. Ultimately, some of these influence the interaction in the GUI editor, as such the most relevant differences are highlighted below.

Components and Hierarchy: While the standard is closely aligned with the FMI it is not limited to describing systems purely consisting of FMUs. Instead the standard

[4] `https://ssp-standard.org/`
[5] `https://github.com/CATIA-Systems/FMPy`

```
\---single_tank.ssp          : collection of multi models
    | SystemStructure.ssd    : default multi model
    | VarA.ssd               : alternative multi model
    | ParameterSet.ssv       : set of parameters
    | ParameterMappings.ssm  : binding params to instances
    | SignalDictionary.ssb   : mechanism to group signals
    |
    +---documentation
    | index.html
    |
    +---extra                : extra files extension mechanism
    |
    \---resources            : implementations of components
      WaterTank.fmu
      Controller.fmu
      subsystem.ssp          : potential subsystem
```

Fig. 4. Example of the file structure of the running example stored as a SSP package. The comments on the side describe the role of the file.

establishes the concept of a *component* – an atomic block of the system, which is backed by some underlying implementation. For example, a component may reference a FMU as its implementation. A component may itself be a SSP package, which in this context, is referred to as a *sub-system*. The latter enables the creation of hierarchies of components.

Parameterization: There is an important difference between how parameters are treated in a multi-model and by SSP. In the former all parameters are defined inline, there is no mechanism to reference a "shared" parameter from multiple components. SSP on the other hand provides the *parameter sets* and *parameter mappings* concepts which makes it possible to declare a set of parameters and then bind these to the concrete components[6].

Extensibility: Similar to FMI the standard is largely based on XML and is designed to be extensible through three types of extension mechanisms: Annotations, extra files and MIME type-based format dispatch. This allows for the creation of new extensions to the standard referred to as *layered standards*. A potential use case of this is to extend the standard with support for semantic adaptations.

3 Graphical Editor

Currently, several external tools are required for the creation and composition of co-simulation scenarios. The primary goal of the development of an integrated graphical editor is to integrate the functionality of these tools and provide a more efficient workflow. We refer to this editor as the *Graphical Editor*.

[6] Inside the SSP standard these are referred to as System Structure Parameter Values and System Structure Parameter Mappings.

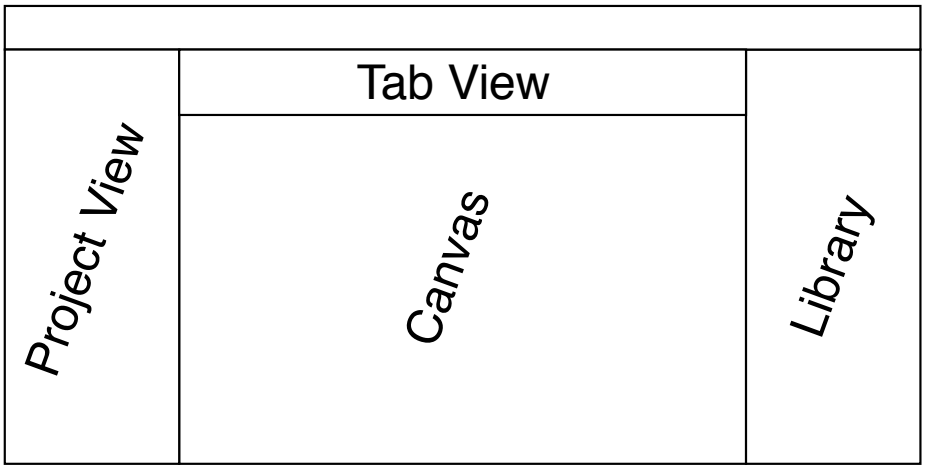

Fig. 5. Mock-up of the envisioned editor annotated with the names of its constituent parts.

The editor aims to provide an interaction that is familiar to users of modelling software such as 20-sim, OMEdit or Simulink. Common for these is that they provide a block-based design flow which allows the user to compose scenarios by dragging components from a library onto a canvas. The scenarios are further elaborated by dragging connections between compatible ports.

An outline of the editor and its different panels can be seen in Figure 5. The functionality of each of these is described in the subsections below.

3.1 Project View

The *project view* provides a tree view of the current project's *artefacts*. An example of these, and the relation between project view and the canvas are is depicted in Figure 6. As shown these are categorised based on their type such as components and connections. Artefacts may themselves be hierarchically composed as is the case of the "cont" component. In these cases they may be expanded in the tree to reveal their internal contents.

3.2 Composition

A central part of defining a simulation scenario is defining instances of components and how these are connected. Instances are created by dragging an item from the *components-group* into the diagram as seen in figure 7. Naturally every instance must be assigned a name. This may be handled by either prompting the user upon creation or automatically resolving the instance name based on the component name as seen in figure 7a.

After having instantiated the needed components the connections between their ports may be established. Ports are represented by symbols reminiscent of a less-than sign with the name of the port next to it. An input is indicated by a symbol pointing

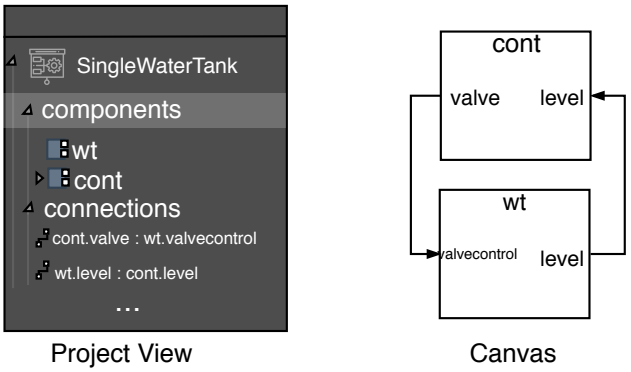

Fig. 6. Project view and canvas relationship

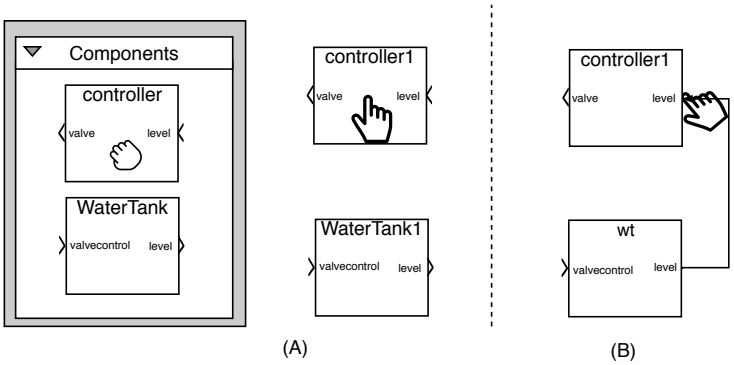

Fig. 7. Illustration depicting how the inputs and outputs of FMUs may be connected graphically.

inwards whereas an output is indicated by a symbol pointing outwards. This representation is proposed as it is compact and shows the name and direction of a port.

A pair of ports are connected by hovering the mouse over an unconnected output port, initiating a drag and finally releasing the drag over an unconnected input port. To limit the probability of connecting ports incorrectly the GUI will not accept a connection from the source port to an incompatible target. An simple rule is to check if the units of the two ports matches, but it is possible to imagine more elaborate validation procedures.

3.3 Parameterisation

A component may expose several parameters allowing its behaviour to be adjusted without the need to modify its implementation. This process is referred to as parameterisation. In the context of the running example a parameter of the water tank component would be its area or the gravitational constant. Double clicking on a component instance will open an *configuration-window*, as seen in figure 8. The window contains a section which list all available parameters of the component. A concrete value may be entered for a given parameter, or it may be bound to a parameter of the parameter set.

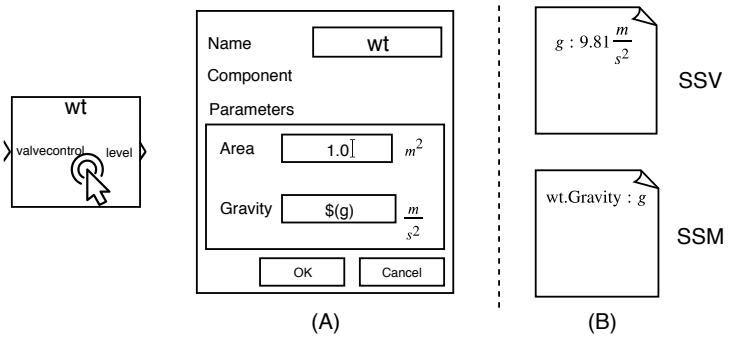

Fig. 8. Parameterization of a component. A *configuration-window* is opened when a component is double clicked.

The editor allows such bindings to be defined directly from the configuration-window. In the context of the editor these are referred to as *parameter-binding*. The bindings are defined using a special notation: *$(parameter)*, as seen on 8a.

The bound parameters of the components now refer to the value of the corresponding parameters of the parameter set. An example of how this may be used would be if multiple water tanks existed, in this case they would all depend on the gravitational acceleration, g, experienced in their environment. Using the binding mechanism the value of g may adjusted in the parameter set to examine its impact on the whole system's performance.

From a users standpoint it would be useful to know the valid ranges and constraints of a parameter's values. This would allow the editor to inform the user of any inconsistencies during configuration rather than resulting in error during simulation. The SSP standard itself does not define any mechanism for specifying these. However, due to its extensibility it would be possible to implement a layered standard for this purpose.

3.4 Representing Hierarchy

A central feature of the SSP standard is its ability to describe hierarchical systems. Specifically it allows for the composition of a system from multiple subsystems, which may themselves be SSP-packages.

More related to the editor itself is how the hierarchy of components is represented. 20-sim and Simulink both allow the creation and navigation of this hierarchy graphically. Both use a very similar approach, the primary differences being notation. An an example of what this may look like in the editor is shown in Figure 9.

Navigation is straightforward, double clicking a subsystem changes the view such that it shows the subsystems internals. We refer to this as *opening* a subsystem. This process can be repeated to reach deeper into the hierarchy. Given the lack of a parent component to click on, navigating upwards is done using a toolbar button.

The process of creating subsystems from existing components may be done in the following way. First one or more components are selected, then the "Subsystem from Selection" item is chosen from the context menu. This results is a new subsystem in place of the selected components. The subsystem contains the now replaced components.

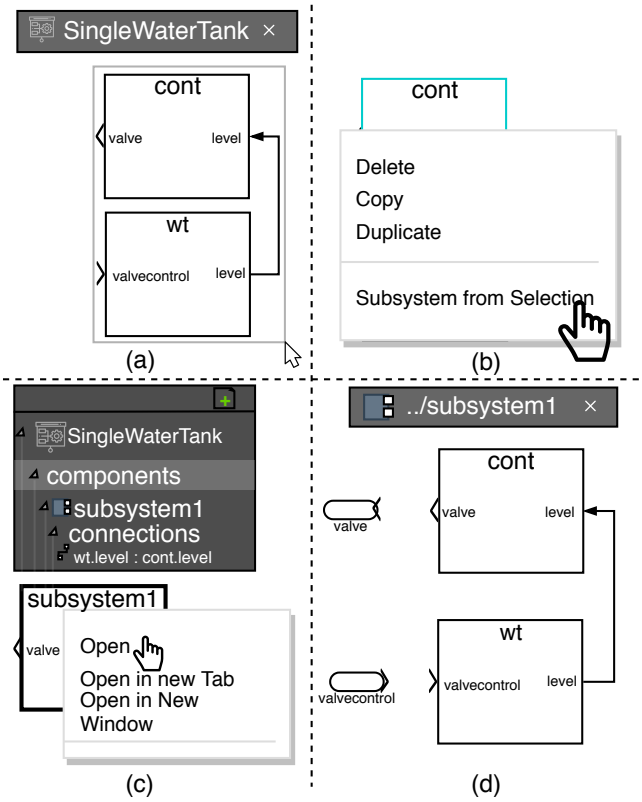

Fig. 9. Proposed method to create subsystem from a selection of components. (a) components are selected. (b) "Subsystem from Selection" is selected in the context menu. (c) the two components have been converted into a single subsystem, which is double clicked to access the internal view. (d) internal view of the subsystem.

3.5 Semantic Adaptations

A solution is proposed that allows different types of semantic adaptations graphically using the familiar drag and drop system, as shown in Figure 10. A group containing entries corresponding to commonly used types of adaptations is added to the library. An adaptation can be dragged onto a component thereby "decorating" it with the adaptation. The parameters of an adaptation may be configured in a similar fashion to how a component is parameterised.

The suitability of this approach relies on the available adaptations being sufficiently modular, such that they can be composed to cover most needs. Alternatively, its possible to imagine an "DSL"-adaptation which allows for arbitrary adaptations made in a domain specific language.

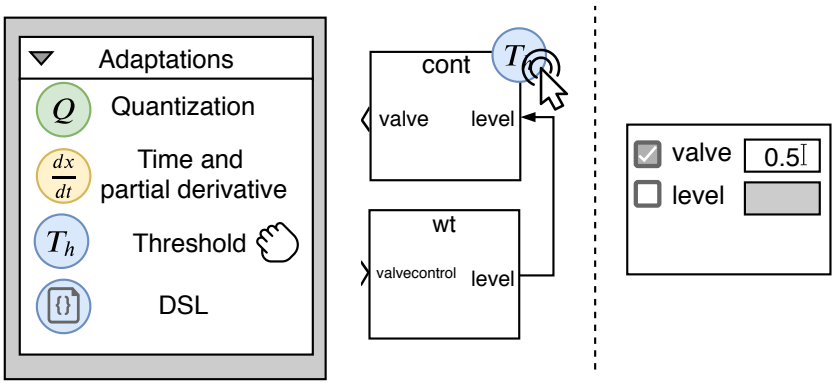

Fig. 10. applying and configuring of semantic adaptations.

4 Graphical Editor Design

Similar to the existing application the graphical editor is written in the Typescript[7] language on top of the Angular framework [1]. Angular is a front end framework consisting of several libraries and utility programs that aid in the development of web applications.

The editor implements the *model-view-controller* (MVC) design pattern [6]. This emphasizes the separation of data of the application from how its represented graphically. It allows the data of the model to be represented by multiple views. A specific example of two views that share the same data are the project and canvas views, shown earlier in Figure 5.

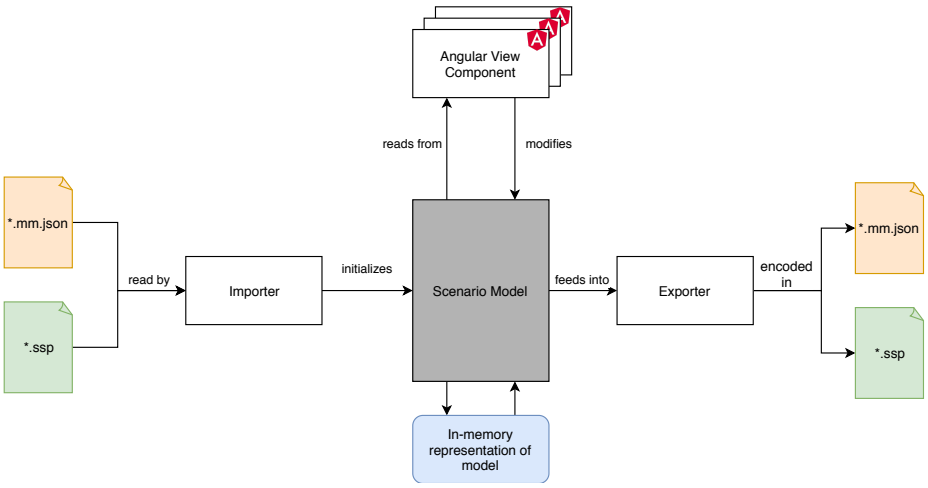

Fig. 11. Flow importing and exporting multi-models and SSPs

[7] https://www.typescriptlang.org/

A conceptual view of the architecture can be seen on Figure 11. Starting from the left we see that a co-simulation scenario can be imported from a file encoded as either the current multi model format or the SSP format. The file is parsed and used to instantiate an in-memory representation of the scenario referred to as the *scenario model*. The model has several uses. It provides an abstraction of the underlying model to components such as the editor. The editor reads and generates a graphical representation of the model. Conversely, changes to the scenario made through the editor are propagated back to the model. Finally, the model may be exported to a format suitable for exchange, such as multi-model or SSP.

4.1 Rendering of components

The canvas-view is built on top of the javascript library mxGraph[8]. It provides the functionality of creating interactive graphs, creating connectors, undo-redo, automatic layout and much more.

4.2 Modules

Specifically the implementations consists of several components, referred to as a *Modules* within the context of Angular[9]. A module is a collection of components and services which provide a coherent set of functionality to the application. In this setting the work presented in this paper naturally needs to fit into the recently developed cloud version of the INTO-CPS Application [15].

To facilitate version management the modules are published on NPM under the namespace *@into-cps*. The concrete packages are listed below:

@into-cps/graphical-editor
@into-cps/scenario-model
@into-cps/model-serialization

5 Concluding Remarks

This paper has demonstrated our work in progress towards supporting SSP in order to provide graphical editing support for the composition (and adaptation) of FMUs directly inside the INTO-CPS Application. We envisage that this will increase the user-friendliness of the INTO-CPS Application since it requires one external tool less and it will be able to support more than what is possible in the SysML CPS profile. Furthermore, there are additional ideas (see below) for how additional improvements can be made for this technology.

6 Future Work

The work presented in this paper is at an early stage and as a consequence there are many extension possibilities that we envisage in the future. Below the most obvious directions are listed.

[8] `https://github.com/jgraph/mxgraph`
[9] `https://angular.io/guide/architecture-modules`

6.1 Generating template of a component

In the existing SysML CPS profile there is a concept of Architecture Structure Diagrams. Inside these it is possible to include all the FMUs included in a particular project. The interface for each of these can be described here and as a consequence it is possible to export model descriptions for each FMU. Such model descriptions can then subsequently be imported by individual modelling and simulation tools such as Overture [10]. We envisage that it also will be easy to include this capability inside the graphical editor. Such model descriptions can also be included as new libraries that others will be able to make use of in their co-simulation composition.

6.2 Implement SSP support in Maestro

The added value of the editor relies on the co-simulation Maestro supporting the SSP standard. A central question is how to support the simulation of hierarchical systems. One potential approach is turning each subsystem into a FMUs. This approach is described in Hierarchical FMU [16] and the DSL for hierarchical co-simulation [7].

6.3 Runtime validation

One of the uses of co-simulation is to validate a system's behaviour during runtime to detect and correct any potential design flaws, as early in the development cycle as possible. Currently there is no mechanism built in to INTO-CPS to support this task. It is useful to investigate whether a suitable method exists for providing runtime validation during simulation. Potential inspiration may be drawn from [9].

6.4 Simplify design space exploration and adapt to SSP

In the context of INTO-CPS design space exploration is a mechanism that allows a scenario to be run multiple times with different combinations of parameters. The performance of each run is evaluated by means one or more cost function, each defined in its own Python script. The scripts takes as input the traces resulting from the INTO-CPS co-simulation, its parameters and global information such as the step size.

Currently this information is passed to the scripts by the means of positional arguments and therefore requires the user to manually define the ordering of the individual arguments. This obviously does not scale well with the number of parameters to the scripts.

To eliminate the need for manually ordering a reference to an object containing the relevant information may be passed instead. This would allow a script to access variables using semantically meaningful names such as *data.traces.wt.level* instead of *sys.args[3]*.

6.5 Package manager for components

The tools 20-sim, OMEdit and Simulink all come with an extensive library of components which provide much of the functionality a user would otherwise have to implement themselves. The same benefits are also realised in INTO-CPS application where open-source libraries and framework are used extensively.

While the SSP standard provides the format for exchange it does not describe the infrastructure to share these. It would be interesting to investigate effective mechanisms for publishing and managing components. Potentially inspiration can be drawn from a package manager such as the one used in the application, npm[10].

Acknowledgements

We are grateful to the Poul Due Jensen Foundation, which has supported the establishment of a new Centre for Digital Twin Technology at Aarhus University, which will take forward the principles, tools and applications of the engineering of digital twins. We also acknowledge EU for funding the INTO-CPS project (grant agreement number 644047) which was the original source of funding for the INTO-CPS Application. Finally, we thank the reviewers for their throughout feedback.

References

1. Ang, K.H., Chong, G., Li, Y.: Pid control system analysis, design, and technology. IEEE Transactions on Control Systems Technology 13(4), 559–576 (July 2005)
2. Bagnato, A., Brosse, E., Quadri, I., Sadovykh, A.: SysML for Modeling Co-simulation Orchestration over FMI: the INTO-CPS Approach. Ada User Journa 37(4), 215–218 (December 2016)
3. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)
4. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)
5. Foldager, F., Balling, O., Gamble, C., Larsen, P.G., Boel, M., Green, O.: Design Space Exploration in the Development of Agricultural Robots. In: AgEng conference. Wageningen, The Netherlands (July 2018)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
7. Gomes, C., Meyers, B., Denil, J., Thule, C., Lausdahl, K., Vangheluwe, H., Meulenaere, P.D.: Semantic adaptation for FMI co-simulation with hierarchical simulators. SIMULATION 0(0), 0037549718759775 (2018), https://doi.org/10.1177/0037549718759775
8. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. 51(3), 49:1–49:33 (May 2018)
9. Havelund, K., Roşu, G.: An overview of the runtime verification tool java pathexplorer. Form. Methods Syst. Des. 24(2), 189–215 (Mar 2004), https://doi.org/10.1023/B:FORM.0000017721.39909.4b

[10] https://www.npmjs.com/

10. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
11. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
12. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), ISBN 9781498742450
13. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards Semantically Integrated Models and Tools for Cyber-Physical Systems Design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
14. Modelica Association Project SSP: System Structure and Parameterization Specification Document, Version 1.0. Modelica Association, Linköping, Sweden (2019), http://www.ssp-standard.org
15. Rasmussen, M.B., Thule, C., Macedo, H.D., Larsen, P.G.: Moving the INTO-CPS Application to the Cloud. In: The 17th Overture workshop. Porto, Portugal (October 2019)
16. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), http://www.sciencedirect.com/science/article/pii/S1569190X1830193X