Enhancing SSP creation using sspgen

Lars Ivar Hatledal[1] Eirik Fagerhaug[1]

[1]Department of ICT and Natural Sciences, Norwegian University of Science and Technology, Norway
{laht}@ntnu.no

Abstract

The System Structure and Parameterization (SSP) standard is a tool independent standard to define complete systems consisting of one or more components, including its parameterization, that can be transferred between simulation tools. Thus the SSP standard is a natural extension to the Functional Mock-up Interface (FMI) standard, allowing systems of components, rather than just individual components, to be simulated in a growing number of supported tools. This paper introduces sspgen, a textual Domain Specific Language (DSL) for generating SSP archives. The aim of the DSL is to greatly simplify the creation of SSP compatible simulation systems. sspgen is written in the Kotlin programming language, which provide syntax highlighting and static code analysis in selected tools, full access to Java compatible libraries, and more importantly a scripting context so that sspgen definitions can be easily shared and executed on demand. As the DSL is based on a generic programming language, it enables complex expressions to be evaluated for the purpose of e.g., pre-simulation and initialization of variables. The DSL also performs validation and through integration with the Open Simulation Standard - Interface Specification (OSP-IS) even allows more complex connections to be formed than the single scalar connections that the SSP standard defines, while still retaining compliance. Furthermore, the DSL handles automatic packaging of its referenced content into a ready-to-use SSP archive. As a whole, the introduced package makes it easier to create, modify and share SSP systems.

Keywords: Co-simulation, Domain Specific Language, Functional Mock-up Interface, System Structure and Parameterization

1 Introduction

The System Structure and Parameterization (SSP) standard (Köhler et al. 2016) released in 2019, is a tool-independent format for the description, packaging and exchange of system structures and their parameterization. The SSP is comprised of a set of XML-based formats to describe a network of component models with their signal flow and parametrization, as well as a ZIP-based packaging format for efficient distribution of entire systems, including any referenced models and other resources. The SSP contributes to maximizing re-usability of models and parameters across tools and use cases. An SSP (file extension .ssp) is a zip archive containing one or more XML documents, at least one named *SystemStructure.ssd*, declaring the structure of the simulation. The archive also contains any referenced components, like Functional Mock-up Units (FMUs) and other resources. Thanks to the SSP, and provided that the SSP does not contain non-optional implementation specific annotations, a simulation system can be defined once and later simulated in multiple tools. Currently, the SSP is supported by a number of free and commercial tools. See Table 1 for an overview of tools that support SSP import and subsequent execution. In (Lars I Hatledal et al. 2021), the authors made use of SSP to describe a simulation system that were simulated in a number of compatible open-source importers. More specifically Vico, OMSimulator, libcosim, FMPy, and FMI Go!. The latter two required a slight modification to the SSP description file as they did not support the final publicly released version of the standard (1.0). Thus, they are not present in the provided table of supported tools. In this example, the SSP proved its usefulness by allowing the same definition of a system to be tested and benchmarked in several tools.

Table 1. Tools supporting SSP import and simulation (v1.0).

Name	Vendor	License
SYNECT Model Management	dSPACE	commercial
OMSimulator	OpenModelica	free
Model.CONECT	AVL	commercial
FMI Bench	PMFS	commercial
easySSP	eXXelent solutions	free + commercial
Simcenter System Architect	Siemens	commercial
Simcenter Studio	Siemens	commercial
Dymola	Dassault Systèmes	commercial
libcosim	OSP	free
Vico	NTNU	free
Ecos	NTNU	free

As previously mentioned, the SSP is a collection of XML file(s) declaring the content, connections and parameterization of a simulation as well as any resources, like models, required to run the simulation. Given a set of components, e.g., FMUs, the simulation structure can be formalized in an XML file and zipped together with any

resources required. This can be done using nothing else than an text editor and built-in OS capabilities for zipping a folder. However, this approach is tedious, time-consuming and error prone. The resulting SSP archive may include formal and/or logical errors that will not surface until it is loaded into a simulation tool. Depending on the tool used, the source of any errors reported may be non-obvious. Furthermore, XML is a static text-format, which means that parameters must be provided exactly. I.e., a number like $PI/2$ must be pre-computed and manually typed as e.g., 1.57079632679, which is tedious and may lead to accuracy issues depending on the number of decimal points included. Another issue arises with systems that contain similar components, which expects similar or identical connectors and parameterization options. Declaring such a system in XML leads to excessive copy-pasting and performing changes is error prone as the same logical variable needs to be kept track of during modification throughout the document. Another concern is bit-rot, which occurs when some file is left unused and unmaintained, possibly due to poor understanding of the content. Manually generated SSP archives are prime subjects of bit-rot as maintaining them requires substantial knowledge, which typically dwindles over time and might disappear once the archive is transferred to some other recipient.

A domain-specific language (DSL) provide a notation tailored toward some application-domain, and is based on the relevant concepts and features of that domain (Van Deursen and Klint 2002). This paper introduces sspgen, a DSL that aims to ease the creation of SSP archives by providing an accessible and easy to use language construct that is more powerful than manually editing XML or using graphical tools. The solution benefits from existing tooling and provides integration with other standards and tools in order to enhance SSP development.

The rest of the paper is organized as follows; first some related work is given, followed by a presentation of the sspgen software. Finally, a conclusion and notes on future works is given.

2 Related works

Manually creating SSP configurations using basic and readily-available tools, as previously mentioned, is not the only alternative available today.

OMSimulator (Ochel et al. 2019) is a co-simulation framework that allow both import and export of SSP archives. Moreover, it features a simplified Python interface to the underlying C/C++ library for accessibility that can be used to import, define and export SSP configurations. *easySSP* is a graphical, web-based, tool for generating SSP archives. Like graphical tools in general, it favours easy-of-use and accessibility over flexibility. However, large simulations quickly becomes cluttered and editing in a graphical tool has some of the same challenges as editing XML directly.

Additionally, tools using alternative system formats than SSP exists. *Daccosim NG* (Évora Gómez et al. 2019) is a co-simulation framework that features a desktop graphical users interface (GUI) for establishing a simulation graph. The graph may be exported in a custom format only supported by Daccosim, or as an FMU that itself contains other FMUs. The latter allows the system to be loaded into any FMI based simulation tool. While accessible, this solution adds a dependency to an additional solver and the generated FMU is inflexible as it does not allow modifications without re-running the original pipeline. *kopl* is a graphical tool for generating systems compatible with the Open Simulation Platform - System Structure (OSP-SS), which is a format similar to the SSP. By supporting the OSP-IS, connection points are fewer, thus making the system as a whole easier to reason with. The downside is that the format produced is currently only compatible with *libcosim* and eventual tools built on-top of it.

Unlike the graphical tools mentioned, sspgen offers a executable, text-based solution that is more flexible, performs validation, is less verbose than XML, allows arbitrary expressions to be computed as input to the document, and as a novelty, combines the OSP-IS standard with SSP. The DSL defines much of the same concepts and structure as is found in the standard, making the learning curve less steep for users already familiar with the SSP standard. As the solution is text-based, modifications can be easily shared and version-controlled. The solution is further elaborated in the following section.

3 sspgen

This section introduces sspgen, a Kotlin DSL for generating SSP archives that is publicly available as a Maven artifact. Kotlin is a statically typed programming language that runs on the Java Virtual Machine (JVM) that is interoperable and comparable with the more known Java language, but offers additional features and a offer a less verbose syntax. Any and all libraries available for Java are usable by Kotlin and visa versa. Today, Kotlin is mostly known as the main programming language for Android applications, however it is also used as a replacement for JavaScript in web-applications and Java for desktop applications. Thanks to its intuitive type-system and smart use of closures, Kotlin is a very suitable and powerful language for building an embedded DSL. That is, a DSL that is implemented within a host language. While embedded DSLs in general are less flexible than external DSLs, which use an independent interpreter or compiler, embedded DSLs typically benefit from existing tooling. Kotlin for instance, supports a scripting context that allow Kotlin code to be executed without the need for a build-system. While executing a script, any third party dependencies are automatically resolved and the code is compiled on-the-fly. The stand-alone Kotlin compiler that makes this possible is bundled with the IntelliJ integrated development

environment (IDE), but it can also be downloaded directly from the official Kotlin repository on GitHub. Using IntelliJ, however, is encouraged as it adds auto-completion, static-code-analysis, syntax highlighting and enables the script to be executed through a GUI as opposed to the command line.

Listing 1. Kotlin script skeleton demonstrating basic usage of sspgen.

```
@file:DependsOn("info.laht.sspgen:dsl:0.5.2")

import no.ntnu.ihb.sspgen.dsl.*

ssp("...") {
 resources {
  file("path/to/FMU.fmu")
 }
 ssd("...") {
  system("...") {
   elements {
    component("FMU", "resources/FMU.fmu") {
     connectors {
      real("output", output) {
       unit("m/s")
      }
      real("input", input)
      integer("counter", output)
     }
    }
   }
   connections {}
  }
  defaultExperiment(startTime = 1.0)
 }
}.build()
```

As mentioned, sspgen is powered by Kotlin, and makes use of closures in such a way that it acts like a *DSL*. Thus offering a DSL context within a generic programming language. While the package is compatible with Java, it can only be intuitively used in the context of Kotlin. As shown in Listing 1, the idea is that users should create a generic Kotlin script, which then adds sspgen as a dependency. The script context allows the code to be easily modified, executed, shared and version-controlled. The actual SSP archive required for simulation is generated on demand by running the script, hopefully reducing the likelihood of bit-rot as maintenance becomes easier. As sspgen runs in a scripting context, generic expressions can be evaluated, which is immensely powerful. For one, loops can be used to declare multiple similar connectors as shown in Listing 2. Furthermore, scripts can make use of any third party library compatible with Java in order to compute e.g., parameterization options. Component references are included either a through a file handle, an URL or as the path to a PythonFMU script. Using the URL option, the script definitions can be shared as a single executable file and easily version-controlled. Another benefit from using URLs is that the referenced components can be updated automatically, as re-running the script can be configured to download the latest version. If this behaviour is not desired, one could naturally point the URL to a fixed version. E.g., if the artifact version-controlled using Git, one could point the URL to a fixed tag rather than an evolving branch.

Listing 2. Using loops to declare similarly named connectors.

```
connectors {
 for (i in 0..10) {
  real("transform[i].position.x", input)
  real("transform[i].position.y", input)
  real("transform[i].position.z", input)
 }
}
```

Listing 3. Declaring connections using sspgen.

```
// SSP type connections
connections {
    "wheel.p1.f" to "chassis.p.f"
    "chassis.p.e" to "wheel.p1.e"
    "ground.p.f" to "wheel.p.f"
    "wheel.p.e" to "ground.p.e"
}

// OSP-IS type connections
ospConnections {
    "chassis.linear mechanical port" to
        "wheel.chassis port"
    "wheel.ground port" to
        "ground.linear mechanical port"
}
```

Listing 4. Declaring annotations using sspgen.

```
val stepSize = 1.0/100
...
defaultExperiment {
  annotations {
    annotation("org.openmodelica") {
      """
        <oms:SimulationInformation resultFile=
            "results.mat"/>
      """
    }
    annotation("com.opensimulationplatform") {
      """
        <osp:Algorithm>
          <osp:FixedStepAlgorithm baseStepSize=
            "$stepSize"/>
        </osp:Algorithm>
      """
    }
  }
}

namespaces {
  namespace("oms",
    "http://openmodelica.org/oms")
  namespace("osp",
    "http://opensimulationplatform.com/SSP/
        OSPAnnotations")
}
```

3.1 Connections

The *connections* closure in Listing 3 shows how regular connections between components are made. By default, outputs are declared on the left hand side of the infix function *to*. It is also possible to swap the ordering for all connections by specifying a boolean flag. Declaring optional

linear transformations on the signals formed are achieved by invoking an instance method on the object returned by the individual connection objects.

3.2 Annotations

Several SSP importers, like OMSimulator and libcosim, require tool-specific annotations to be present in the imported XML. This requires users that want to support multiple tools, and use some tool for creating the SSP, to edit the generated XML manually. sspgen allows annotations to be added as plain-text, allowing multiple tools to be supported without further editing. Listing 4 shows how annotations are declared using sspgen.

3.3 Validation

sspgen performs several types of validation of the declared content. Firstly, it checks that the connectors refers to actual variables and that the declared type matches. Secondly it checks that the connections are valid and that a connector has been declared for a given variable. FMI4j (Lars Ivar Hatledal, Zhang, et al. 2018), a JVM library for importing FMUs, is used to validate FMU components based on their *modelDescription.xml*. sspgen is also able to recognise proxy-fmu [1] components (a solution for remote execution of FMUs developed under the umbrella of the Open Simulation Platform), so that their *modelDescription.xml* files can be validated in the same way as regular FMUs. Furthermore, sspgen can perform additional checks as part of the integration with OSP-IS and/or FMI-VDM-Model as explained in more detail below. Currently, FMI version 1.0 and 2.0 for co-simulation is supported. When working with non-compliant models or unsupported FMI versions, it is possible to turn of validation.

3.4 Integration with OSP-IS

The OSP interface specification (OSP-IS) is an addition to the FMI 2.0 standard for co-simulation which provides a method for adding semantic meaning to model interface variables. OSP-IS aims to simplify the model connection process, and enables validation of semantically correct simulations (Open Simulation Platform 2020b). In short, an XML document adhering to the OSP-IS, which declares more complex input and output variable relationships, can be used by tools that support it to form more complex and semantically correct connections between models. Currently, the only tool that natively supports the OSP-IS is libcosim (Open Simulation Platform 2020a) from the OSP foundation. sspgen allows OSP-IS connections to be formed within the DSL, which are later transpiled to single scalar connections that the SSP supports, while retaining the static type checking during the build process. Thus, sspgen enables the OSP-IS to be used by any SSP compatible tool. For example the *ospConnections* shown in Listing 3 are transpiled to the SSP compatible XML as shown in Listing 5

Listing 5. OSP-IS connections transpiled to SSP.

```
<ssd:Connections>
    <ssd:Connection startElement="wheel"
        startConnector="p1.f" endElement=
        "chassis" endConnector="p.f"/>
    <ssd:Connection startElement="chassis"
        startConnector="p.e" endElement=
        "wheel" endConnector="p1.e"/>
    <ssd:Connection startElement="ground"
        startConnector="p.f" endElement=
        "wheel" endConnector="p.f"/>
    <ssd:Connection startElement="wheel"
        startConnector="p.e" endElement=
        "ground" endConnector="p.e"/>
</ssd:Connections>
```

3.5 Integration with FMI-VDM-Model

sspgen optionally integrates with the FMI-VDM-Model (Battle et al. 2019) tool created by the INTO-CPS project. The FMI-VDM-Model is a formal model of the FMI standard using the VDM Specification Language. The integration allows optional static analysis of the included FMUs for informative purposes. To use, simply provide the path to the FMI-VDM-Model executable when invoking the sspgen functions *validate* or *build*.

3.6 Integration with PythonFMU

PythonFMU (Lars Ivar Hatledal, Collonval, and Zhang 2020) is a Python framework for developing FMUs using the Python programming language. sspgen allows components written using PythonFMU to be declared in its source form. sspgen then calls the PythonFMU packaging tool, which must be pre-installed on the system, during the build process. This makes it easier to prototype SSP systems by shortening the development loop. This command is featured in the DSL, but thanks to the underlying scripting context, it is possible for users to write generic code that produced components from other sources on demand.

4 Conclusion and future work

This paper presents sspgen, a high-level DSL aimed at easing and enhancing the creation of SSP archives. More specifically, the DSL enables evaluation of complex expressions, is less verbose than XML and introduces concepts that makes authoring easier, such as the ability to copy data between components. Furthermore, the DSL is written in Kotlin, a statically typed language that offers auto-completion and syntax highlighting. Moreover, Kotlin features a standard library that further expand the already well-established standard library provided by Java. Additionally, a vast eco-system of third party libraries are readily-available. In the context of sspgen, such libraries can be used to e.g., compute parameterization options for components prior to export. More importantly, sspgen performs validation of its content so that the user can address potential issues before actually loading

[1] https://github.com/open-simulation-platform/proxy-fmu

the SSP into a simulation tool. Furthermore, the DSL supports the OSP-IS standard, allowing more complex connections to be formed, which can be further validated for semantic correctness. All while retaining compliance with the SSP standard. The tool is not feature complete according to the standard, but covers the necessary features in order to be effective by the current users and has seen wide usage internally by researchers and master students at NTNU campus Aalesund for the purpose of modelling maritime systems. The tool is largely stable and future work includes maintenance, documentation and responding to user requests. Additionally, support for FMI 3.0 and further versions of the SSP standard will be considered. sspgen is open-source and available from `https://github.com/Ecos-platform/sspgen` under a permissive license.

Acknowledgements

This work was supported in part by the Project "SFI Offshore Mechatronics", under Grant 237896 from Research Council of Norway.

References

Battle, Nick et al. (2019). "Towards a Static Check of FMUs in VDM-SL". In: *International Symposium on Formal Methods*. Springer, pp. 272–288.

Évora Gómez, José et al. (2019). "Daccosim NG: co-simulation made simpler and faster". In: *Linköping electronic conference proceedings*.

Hatledal, Lars I et al. (2021). "Vico: An entity-component-system based co-simulation framework". In: *Simulation Modelling Practice and Theory* 108, p. 102243.

Hatledal, Lars Ivar, Frédéric Collonval, and Houxiang Zhang (2020). "Enabling python driven co-simulation models with pythonfmu". In: *Proceedings of the 34th International ECMS-Conference on Modelling and Simulation-ECMS 2020*. ECMS European Council for Modelling and Simulation.

Hatledal, Lars Ivar, Houxiang Zhang, et al. (2018). "Fmi4j: A software package for working with functional mock-up units on the java virtual machine". In: *The 59th Conference on Simulation and Modelling (SIMS 59)*. Linköping University Electronic Press.

Köhler, Jochen et al. (2016). "Modelica-association-project "system structure and parameterization"–early insights". In: *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*. 124. Linköping University Electronic Press, pp. 35–42.

Ochel, Lennart et al. (2019). "Omsimulator–integrated fmi and tlm-based co-simulation with composite model editing and ssp". In: *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*. 157. Linköping University Electronic Press.

Open Simulation Platform (2020a). *libcosim*. (Date accessed 11-May-2022). URL: https://github.com/open-simulation-platform/libcosim.

Open Simulation Platform (2020b). *OSP-IS*. (Date accessed 11-May-2022). URL: https://opensimulationplatform.com/specification/.

Van Deursen, Arie and Paul Klint (2002). "Domain-specific language design requires feature descriptions". In: *Journal of computing and information technology* 10.1, pp. 1–17.