AN INTEGRATED MODELING, SIMULATION AND EXPERIMENTATION ENVIRONMENT IN PYTHON BASED ON SES/MB AND DEVS

Hendrik Folkerts
Thorsten Pawletta
Christina Deatcu

Research Group CEA
University of Applied Sciences Wismar
Wismar, D-23966, Germany
hendrik.folkerts@cea-wismar.de
{thorsten.pawletta,christina.deatcu}@hs-wismar.de

Jean-Francois Santucci
Laurent Capocchi

SPE UMR CNRS 6134
University of Corsica
Campus Grimaldi
20250 Corte, France
{santucci, capocchi}@univ-corse.fr

ABSTRACT

This paper proposes a Python-based infrastructure for studying the characteristics and behavior of families of systems. The infrastructure allows automatic execution of simulation experiments with varying system structures as well as with varying parameter sets in different simulators. Possible system structures and parametrizations are defined using an extended System Entity Structure (SES). Based on rules one specific system configuration is derived. In connection with a model base (MB) an executable model for multiple simulation platforms can be created. This paper focuses on the specification and generation of Discrete Event System Specification (DEVS) models for the DEVS simulator DEVSimPy. The derived model is executed in the DEVSimPy simulator. Depending on the results the subsequent derivation and generation of a model is controlled reactively. The proposed infrastructure is illustrated using an example that includes various abstraction hierarchies and time granularity levels.

Keywords: system entity structure, SES, DEVS, model generation, Python.

1 INTRODUCTION

Generally, variability modeling can be seen as an approach to describe more than one system configuration. A system configuration incorporates the structure of the model as well as the parameter settings. The wish to model more than one system configuration can originate from several use cases. Different system configurations arise modeling varying real world systems, such as developing electronic control units for cars. In this case, models are similar for one car type, but still are highly diverse depending on the equipment of the single car. Another use case is using variability modeling to find an optimal design of a system which does not yet exist, as e.g. in the planning phase for new plants. Finally, also modeling real world systems with different levels of detail leads to different system configurations. For all these cases modeling each single system configuration separately and manually would be very time consuming, costly and error prone. This of course calls for automation of the processes necessary for modeling families of systems and for deriving certain system configurations.

In software engineering a commonly used approach for variabilty modeling is the use of Feature Models in combination with 150% models (Capilla, Bosch, and Kang 2013). Transformation methods allow the

synthesis of an executable model with deactivated parts of any domain. 150% models include all variants in just one model and are therefor quite complex and often hard to maintain. Another approach is originated in systems theory. A meta-model of the system configurations represented by a System Entity Structure (SES) is created. Transformation methods linking basic models organized in a model base (MB) support the generation of an optimally taylored executable model. To allow automatic processing of SES as well as automatic model generation and execution, the SES/MB framework as described in Zeigler, Kim, and Praehofer (2000) was extended by new modeling features, methods, and components (Schmidt, Durak, and Pawletta 2016). To bring this powerful approach from theory to practice and make it applicable for engineers, well-founded software tools are needed.

The extended SES/MB (eSES/MB) framework was implemented in a prototype software tool in MATLAB (Pawletta et al. 2014) for modeling and generation of MATLAB/Simulink models. This paper discusses an advanced Python-based prototype. In contrast to the MATLAB prototype, its objective is to support the generation and execution of models for different simulation environments. The Python prototype consists of several tools: SESToPy (**S**ystem **E**ntity **S**tructure **To**ols **Py**thon), SESMoPy (**S**ystem **E**ntity **S**tructure **Mo**delbuilder **Py**thon), and SESEuPy (**S**ystem **E**ntity **S**tructure **E**xecution **u**nit **Py**thon). SESToPy is a graphical editor for creating SES models. It supports all methods of the eSES/MB approach for the management of different SES models and the derivation of concrete model configurations. The modelbuilder SESMoPy implements two basic ways of model generation: (i) native model generation and (ii) model generation using the Functional Mock-up Interface (FMI). SESEuPy implements an execution unit for controlling the execution of a model in a target simulator.

This paper focuses on the collaboration of the several tools, which form the infrastructure. It discusses the integration of the DEVSimPy simulator using the native model generation approach. With the integration of DEVSimPy a full-fledged Python-based framework is achieved for variability modeling and system simulation. The FMI-based approach is not detailed further. Using the example of a watershed, introduced in Santucci, Capocchi, and Zeigler (2016), the usage of the framework is demonstrated. The watershed problem is according to Zeigler, Mittal, and Traore (2018) a typical example of a multi-resolution system. In contrast to the existing watershed model, the solution in this paper is based on the regular elements and does not introduce new SES elements for modeling abstraction hierarchies and time granularities.

The paper is organized as follows. In Section 2 an introduction to the eSES/MB infrastructure is given. Section 3 summarizes main aspects of the watershed example. Section 4 presents the Python-based framework including the DEVSimPy simulator. Moreover, it illustrates the basic modeling steps using the watershed example. Section 5 focuses on the collaboration of the different tools with the goal to automate the generation and execution of DEVSimPy models. Section 6 gives a summary and outlines intended next steps.

2 SHORT INTRODUCTION TO THE EXTENDED SES/MB INFRASTRUCTURE

Figure 1 depicts the eSES/MB infrastructure consisting of the eSES/MB framework, an Execution Unit (EU), and an Experiment Control (EC). The SES describing a set of system configurations has been associated with the idea of model generation of modular, hierarchical systems from the very beginning which led to the SES/MB approach as described in Rozenblit and Zeigler (1993) and Zeigler, Kim, and Praehofer (2000). Each system configuration is defined by its system structure and parameter configuration in the SES. The core assets of all system variants are specified as a set of configurable basic models, which are organized in an MB. The classic SES/MB framework defines a set of transformation methods for generating executable simulation models. Model generation is only provided in an interactive way (Zeigler and Hammonds 2007). To support an automated generation and execution of models, the SES/MB approach has been extended (Schmidt, Durak, and Pawletta 2016; CEA 2017; Pawletta et al. 2018). These extensions make the SES/MB approach more pragmatic and usable in a simulation infrastructure for engineering problems.

Although the eSES/MB is usually considered in connection with the generation of simulation models, it is generally applicable to modular-hierarchical structured software systems. In the following a short introduction to the key principles and extensions of the eSES/MB infrastructure is given.

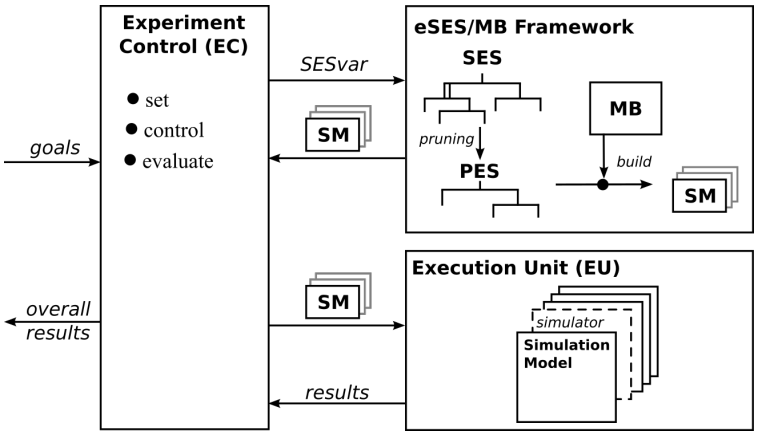

Figure 1: eSES/MB infrastructure based on Pawletta et al. (2018).

2.1 eSES/MB Framework

A set of system designs of any application domain can be coded in an SES. The SES represents all variants of a system in a directed tree structure. Each node in the tree is an entity node or a descriptive node and can have attributes. Entity nodes represent objects of the real or imaginary world, whereas descriptive nodes specify the relationship between entity nodes and can be of three types. Aspect nodes specify the composition of entities and multi-aspect nodes are a special kind of aspect nodes, where all component entities are of the same type. Specialization nodes describe the taxonomy of an entity. The descriptive nodes represent variation points of the system designs coded in the SES. Specifying a composition of entities requires a specification of coupling relations. Couplings can be specified as properties of descriptive nodes of the type aspect or multi-aspect and consist of pairs of entity names and port names, such as (source entity, source port, sink entity, sink port).

For deriving a system variant, the SES needs to be *pruned*. In every variation point a decision needs to be made. The resulting reduced structure is called Pruned Entity Structure (PES), which represents an SES with all variation points resolved. The MB organizes domain and application specific basic models. While the SES describes system configurations independent of a target platform, the basic models in the MB are usually simulator specific. Applying the *build* method, an executable simulation model (SM) is generated using the information in the PES and basic models in the MB.

For automatic generation, execution, and evaluation of system variants the classic SES/MB framework was extended by an an input and output interface as shown in Figure 1. In this context the SES itself has been extended by some new features (Schmidt, Durak, and Pawletta 2016). The term extended SES/MB (eSES/MB) was introduced to distinguish it from the classical approach. Subsequently, the input interface and some essential new features are discussed. The input interface to the eSES/MB framework is established by *SES variables* (SESvar). These variables have a global scope. In order to specify permitted value ranges and dependencies between SESvars *semantic conditions* can be used. They limit the variant diversity and are evaluated during pruning. The introduction of *SES functions* (SESfcn) enables the specification of procedural knowledge. Complex variability, e.g. varying coupling relations, can often be described more easily with SESfcns. At descriptive nodes selection rules can be defined, such as *aspectrules* for aspect and multi-aspect

siblings or *specrules* at specialization nodes. These rules are evaluated during pruning and enable automatic pruning. A special attribute of multi-aspects is the *number of replications* (numRep). The numRep attribute specifies the number of entities to create at a multi-aspect node during pruning. The *mb-attribute* of leaf entity nodes specifies the linkage of the entity node to a basic model in the MB. Attribute values and selection rules can be specified using SESvars or SESfcns. That means that the derivation of system configurations by pruning an SES to a PES can be controlled by the settings of the SESvars.

2.2 Experiment Control and Execution Unit

The Experiment Control (EC) according to Figure 1 is a component introduced for the specification of simulation experiments, automatic experiment execution, and the evaluation of simulation results. The EC controls the generation of a concrete SM or a set of SMs by the eSES/MB framework by assigning values to the SESvars. The EC adds simulation parameters to the returned SM or set of SMs and transmits it to the Execution Unit (EU). The EU links the generated SMs to the simulator, executes simulation runs and, finally, sends the results back to the EC. The results can influence the decision of the EC on how to set the SESvars next.

In the following section a variability problem is introduced exemplary. After that the implementation of the eSES/MB infrastructure in Figure 1 as a modular Python-based framework is discussed.

3 INTRODUCTION OF THE WATERSHED EXAMPLE

The considered example concerns a watershed belonging to a mountainous part of France Alpes (Coron et al. 2014). It involves the rain and snow precipitations over a period of one year. Precipitation can be classified according to its nature into solid or liquid. Depending on the time of year and the geographical area concerned, the amount of solid or liquid precipitation will be different. The proportion moves between the two extremes: only snow or only rain. When precipitation falls as snow, the hydrological response of the watershed is not the same as observed in case of rain. There is a lack of response in the short term watershed due to the accumulation of solid precipitation in the form of a snow cover on the soil surface. When the conditions of melting are met, which can occur several days to several months after the occurrence of precipitation, this water is remobilized. It causes a delayed reaction of the watershed. The dynamic behavior of the watershed is mainly influenced by precipitation and temperature conditions. Usually, the precipitation data are day related and the temperature data are hour or day related. Based on these facts, the dynamic behavior of a watershed can be considered on the basis of different levels of detail. According to Santucci, Capocchi, and Zeigler (2016) the level of detail can be considered regarding the decomposition of influences, called *abstraction hierarchies*, and the *temporal granularity* of data. They introduced the following levels of detail:

- In the simplest case, called *watershed abstraction hierarchy level 0*, the behavior is modeled just considering the percentage of rainfall on a daily basis;
- At the *watershed abstraction hierarchy level 1*, the behavior is expressed in more detail by taking into account the altitude and the flow associated with the three hydrogeological layers: (i) soil, (ii) surface and (iii) aquifer. At this hierarchy level, the altitude influence is broken down in two different levels of detail, called *altitude abstraction hierarchy level 0* and *level 1*.
- Starting from the altitude abstraction hierarchy level 1, snow effects are considered on a daily or hourly basis, which are differentiated as *basin time granularity level 0* and *level 1*.

From the above considerations follows a set of model variants with different levels of detail. They are used as a case study in the following sections.

4 VARIABILITY MODELING USING THE PYTHON-BASED FRAMEWORK

In this section the basic implementation of the previously introduced eSES/MB infrastructure as a modular Python-based software environment is presented. Then, the modeling capabilities are demonstrated using the watershed example. The different model variants of the watershed are specified with an SES and the organization of basic models in an MB is discussed. The other components of the Python software environment and their interaction are discussed in Section 5.

4.1 The Python-Based Framework

Figure 2 shows the structure of the Python-based framework according to the general eSES/MB infrastructure in Figure 1. The eSES/MB infrastructure is realized by several software tools. The software tools

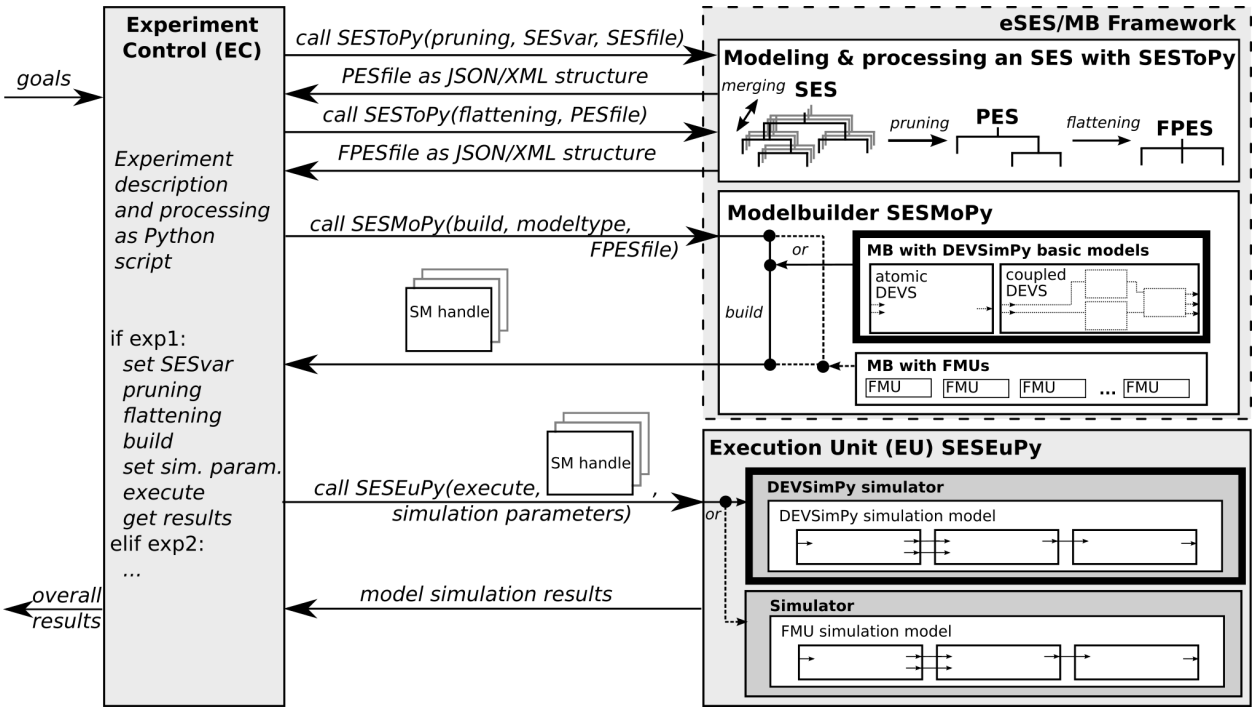

Figure 2: Python-based implementation of the eSES/MB infrastructure for multiple EUs.

discussed in this paper are called *SESToPy, SESMoPy, SESEuPy* and *DEVSimPy* (RG CEA 2019; SPE 2019).

The specification of an SES is done with the tool SESToPy (**S**ystem **E**ntity **S**tructure **To**ols **Py**thon). SESToPy also provides methods for processing an SES up to the derivation of a specific system configuration. The methods can be used interactively or as API methods. SES trees and all kinds of properties including SESvar and SESfcn can be defined via a graphical user interface. The link to basic models in the MB is defined with the special attribute *mb* as described in Section 2. During modeling an SES with SESToPy, checks on the SES and plausibility tests are executed indicating model errors. SESToPy supports the export of SES, PES and FPES structures in the XML or JSON format.

The *build* method of the general eSES/MB infrastructure in Figure 1 is realized with a second tool called SESMoPy (**S**ystem **E**ntity **S**tructure **Mo**delbuilder **Py**thon). SESMoPy implements a modelbuilder that supports two ways of generating executable models. For both approaches, corresponding basic models must be organized in an MB, as shown in Figure 2.

The first approach, called *native model generation*, is the generation of executables for a specific Execution Unit (EU). This approach is discussed in this paper for the DEVSimPy simulator as presented in Figure 2. DEVSimPy (Capocchi et al. 2011) is a Parallel DEVS (PDEVS) based modeling and simulation environment that is implemented in Python. Due to the modular structure and well-defined interfaces, native modelbuilders for any other simulation tools as well as modules for generating component-based software for non-simulation-specific environments can be implemented in the same way. Currently, SESMoPy supports the native model generation for MATLAB/Simulink/Simscape/Simevents, Dymola, OpenModelica, the PDEVS for MATLAB toolbox, and for DEVSimPy.

The second approach is the model generation based on the Functional Mock-up Interface (FMI). Depending on the way of model generation, the processing in the corresponding EU is different, as depicted in Figure 2. In this paper the native model generation using DEVSimPy is presented in detail, the FMI approach is not discussed here.

The Experiment Control (EC) is implemented as a Python script, which defines all experimentation steps, necessary parameters and controls the entire model generation and experiment execution. During modeling, one or more SES are specified interactively with SESToPy. The dynamic behavior is specified using basic models for DEVSimPy that are organized in an MB.

Model generation starts by deriving a specific model configuration from an SES, which is influenced by the current settings of the SESvar. As shown in Figure 2, the EC calls SESToPy's API method *pruning* and passes the SESvar and an SES file. SESToPy returns the model configuration in form of a PES as JSON or XML file structure. Since SESMoPy only supports the model generation for flat PES structures, the inner nodes of the PES need to be removed. The transformation method to achieve a Flattened Pruned Entity Structure (FPES) is called *flattening*. An FPES can be returned to the EC calling SESToPy's API method flattening and passing a PES file. Next, the EC calls the modelbuilder SESMoPy. Depending on the previously described ways of model generation SESMoPy is called with the target simulator, which is DEVSimPy here, and the FPES file as arguments. An executable DEVSimPy SM is saved and a file handle is passed back to the EC. For execution of the SM, the EC sets all necessary parameters and methods. For DEVSimPy simulation parameters such as start time and end time need to be defined. After that the EC calls the tool SESEuPy (**S**ystem **E**ntity **S**tructure **E**xecution **u**nit **Py**thon) which in turn starts a simulation run with DEVSimPy. SESEuPy acts as a kind of wrapper and because of its well-defined interfaces the framework is easily extendable to work with other simulation environments. After simulation execution, simulation results are returned from SESEuPy to the EC and analyzed there. Based on the experiment specification, the EC detects the next steps to take. In case the experiment goals are met, the overall results are calculated and returned. Otherwise more simulation runs with the generated SM might need to be executed or a new model configuration needs to be derived by new settings of the SESvar.

4.2 Variant Modeling with SESToPy

Variant modeling using the tool SESToPy is demonstrated on the example of the watershed introduced in Section 3. In addition to the functionality of the tool SESToPy, the example shows how a model family with different abstraction hierarchies and different time granularity can be clearly modeled using the extensions of the eSES/MB approach.

The SES of the watershed and its representation in SESToPy are depicted in Figure 3. The basic properties to consider when modeling a watershed are the amount of rain and the ambient temperature. Thus, the root of the SES is composed of the leaf nodes *Rain*, *Temp*, the inner node *WS* describing the watershed itself, and an auxiliary leaf node *ToDisk*. The composition is expressed with the aspect node *rootDEC*, which describes a *has-a* relationship. The coupling relations are specified as attribute *cplg1* of the aspect node as described in Subsection 2.1 and are discussed in detail for variable coupling relations at the end of this section. According to the eSES/MB approach in Subsection 2.1, each leaf node defines an mb-attribute referring to a basic model in the DEVSimPy MB. The watershed *WS* can be either of the type *SimpleLayer* or of the type *WS1*, where *WS1* is more detailed. According to Section 3, *SimpleLayer* constitutes the watershed abstraction hierarchy level 0, whereas *WS1* constitutes watershed abstraction hierarchy level 1. For selection during pruning, the watershed *WS* has a descriptive node of type specialization *WSSPEC* as child expressing an *is-a* relationship. Specrules defined as attribute of *WSSPEC* are evaluated, allow the selection of one child node depending on the SESvar *WSLevel*.

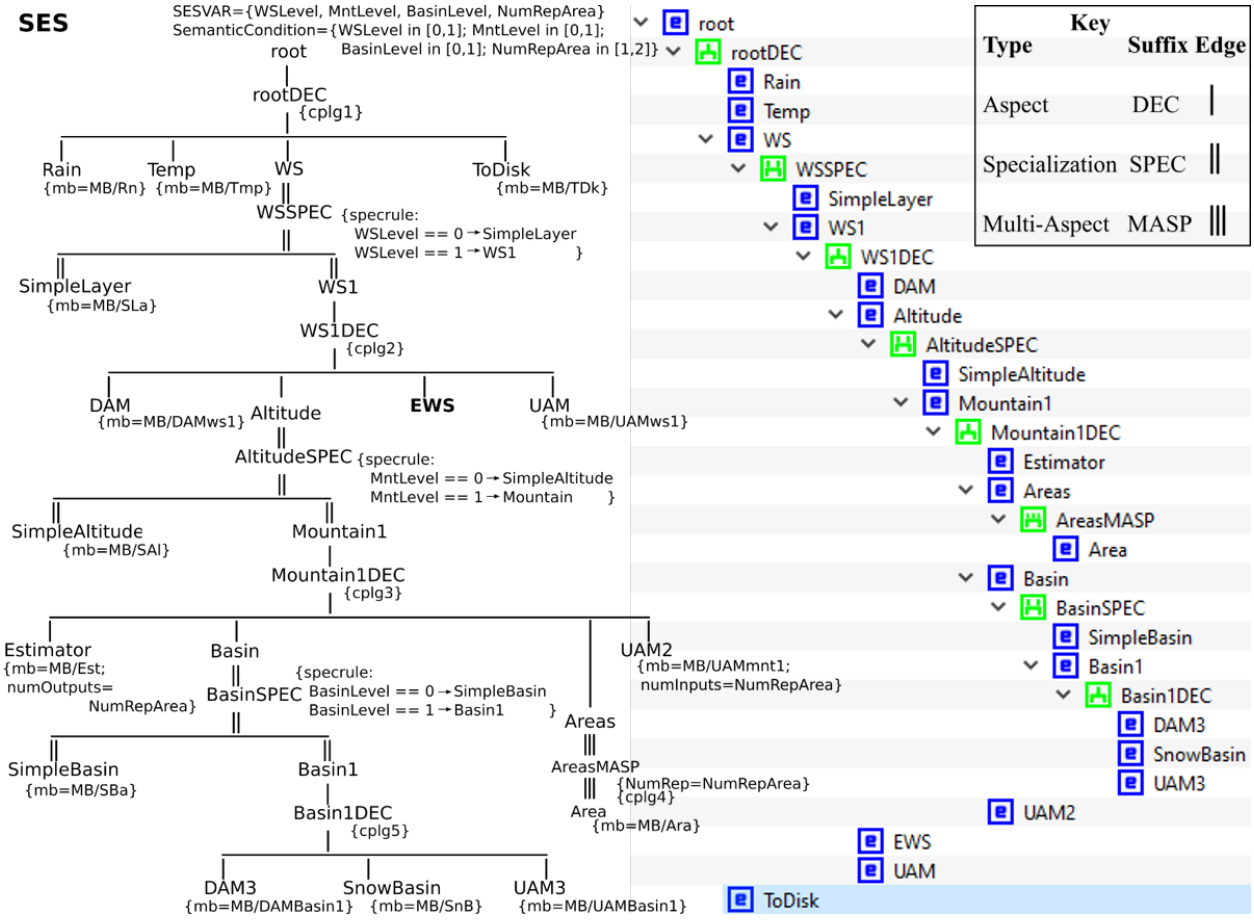

Figure 3: The SES tree of the watershed and its representation in SESToPy.

WS1 is composed of the four siblings: *DAM, Altitude, EWS* and *UAM*. The leaf nodes *DAM* and *UAM* refer to basic models needed for defining abstraction hierarchies in DEVSimPy. The entity *WS1* that represents a coupled model in the context of modular-hierarchical modeling needs to have the same interface as its sibling entity node *SimpleLayer*, which refers to a basic model in the MB. In order to tackle the differences and create unique interfaces independent of the level of abstraction, an *Upward Atomic Model* (UAM) and a *Downward Atomic Model* (DAM) are introduced. *EWS* is an entity node at which another SES defining the

system configuration of an *Elementary watershed* could be merged (Santucci, Capocchi, and Zeigler 2016). It will be considered as a leaf node for the purpose of this example. The fourth sibling of the composition of *WS1* describes the *Altitude* of the watershed. Depending on the level of detail the *Altitude* can be of the type *SimpleAltitude* or of the type *Mountain1*, according to the two altitude abstraction hierarchy levels as discussed in Section 3. Which specialization of the node *Altitude* is selected during pruning is defined by specrules in *AltitudeSPEC* evaluating the SESvar *MntLevel*.

Mountain1 describing the rainfall or snow depending on the height is composed of the four siblings: *Estimator, Basin, Areas* and *UAM2*. The composition structure is specified with the aspect node *Mountain1DEC*. The leaf node *Estimator* refers to a basic model calculating the rainfall in each point of the mesh. The node *Areas* is refined by a variable number of nodes of type *Area*, each of which represents the rainfall on a specific considered area and refers to a basic model in the MB. Hence, *Areas* is followed by a multi-aspect node *AreasMASP*. In *AreasMASP* the variable number of children is specified with the SESvar *NumRepArea* and the coupling relations are defined in the attribute *cplg4*. *UAM2* is another leaf node referring to a basic model necessary for defining an abstraction hierarchy analogously to *UAM*. According to Section 3, on this abstraction hierarchy level a time granularity is specified by different expressions of the inner entity *Basin*. It takes into account the storage effect of water or snow on the mountain. Thus, there is a specialization *BasinSPEC* in which dependent of the SESvar *BasinLevel* a selection is taken during pruning. The resolution of the storage effect of the *Mountain* can be on a daily or hourly basis. If the time granularity shall be coarser the leaf entity *SimpleBasin* is chosen. In case the watershed shall be studied with a finer time resolution, the entity *Basin1* is selected. *Basin1* is decomposed into the leaf entity node *SnowBasin*, which has the siblings *DAM3* and *UAM3* providing the same functionality as *DAM* and *UAM*.

Coupling relations between entity nodes are described using a special attribute at aspect or multi-aspect nodes. In Figure 3, these attributes have the name prefix *cplg*. While the attributes *cplg1, cplg2* and *cplg5* are specifying static relations, *cplg3* and *cplg4* are affected by the varying number of replicated *Area* entities that is specified by the multi-aspect *AreasMASP*. The current number of replications depends on the setting of the SESvar *NumRepArea*. Additionally, this SESvar is used to specify the varying coupling relations. The attributes *cplg3* and *cplg4* specify an SESfcn call and pass the SESvar *NumRepArea* as current input argument. Listing 1 presents an excerpt of the definition of the SESfcn that is called in attribute *cplg3* of aspect *Mountain1DEC*.

```
def cplfcn1(children,parent,NUM):
    #parent is Mountain1, children[0] is Estimator, children[1] is ...
    cplg = []
    cplg.append([parent,"in1",children[0],"in1",""])            #fixed couplings
    cplg.append([parent,"in2",...                               #fixed couplings
    if NUM==2:
        cplg.append([children[0],"out2",children[1],"in2",""])  #variable couplings
        cplg.append([children[1],"out2",children[3],"in2",""])  #variable couplings
    return cplg                                                 #return
```

Listing 1: Excerpt of the Python code defining the SESfcn of the attribute cplg3.

The SESvar *NumRepArea* is passed to the input variable *NUM*. The other two input variables are defined implicitly and encode the names of the parent node and the subnodes of the currently viewed node. The watershed SES defines the four SESvars *WSLevel*, *MntLevel*, *BasinLevel*, and *NumRepArea* as input interface. *Semantic conditions* can be used to specify permitted value ranges and dependencies between SESvar. They are checked before each pruning operation. The following conditions apply to the defined SESvar: $WSLevel \in [0,1]$, $MntLevel \in [0,1]$, $BasinLevel \in [0,1]$, and $NumRepArea \in [1,2]$. Depending on the current SESvar settings different model configurations are derived.

4.3 Creating Basic Models with DEVSimPy

Generating a dynamic model, an MB with basic models needs to be at hand. In the context of DEVS-based modeling, basic models can be atomic or coupled models. The DEVSimPy modeling and simulation environment (Capocchi et al. 2011) provides different tools for specifying such models. DEVSimPy acts as an user-friendly interface for collaborative M&S of DEVS systems implemented in the Python language. It provides a GUI for the PyDEVS (Bolduc and Vangheluwe 2001) and PyPDEVS (Tendeloo and Vangheluwe 2015) simulation kernels. Atomic or coupled DEVSimPy models can be specified in Python language using a text editor and can be stored and structured in libraries. Complex simulation models are created manually by using drag and drop from model libraries. For the watershed example, the DEVSimPy *Watershed* library (Santucci, Capocchi, and Zeigler 2016) has been defined. The SES in Figure 3, modeling the watershed, specifies references to 14 different basic models with the mb-attribute at its leaf nodes. Furthermore, to simplify matters, the node EWS marked in bold is also to be regarded as a leaf node with a reference to a basic model EWS. Hence, the MB contains the following basic models created with DEVSimPy: *Rn*, *Tmp*, *SLa*, *TDk*, *DAMws1*, *SAl*, *EWS*, *UAMws1*, *Est*, *SBa*, *Ara*, *UAMmnt1*, *DAMBasin1*, *SnB*, and *UAMBasin1*. In Figure 4 an excerpt of the MB organizing the basic models for this example is presented.

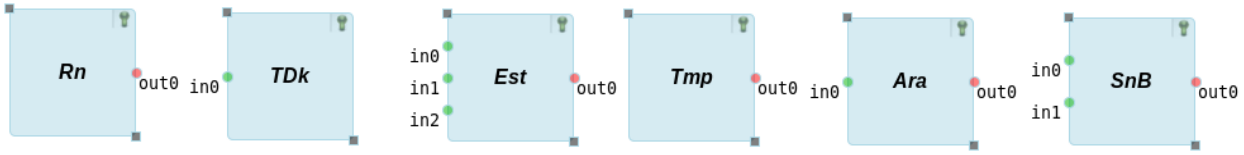

Figure 4: Excerpt of the MB in DEVSimPy.

At this point only the basic models with the name prefix *UAM* and *DAM* should be considered briefly, as introduced in Santucci, Capocchi, and Zeigler (2016). As discussed in Section 3, a set of models with different levels of detail should be investigated. In the SES the necessary variation points are specified using specialization nodes. During pruning, these nodes are resolved by evaluating their selection rules and applying the inheritance axiom. The parent node and one selected child node are merged to a new node. This means, for example, that the node *WS* in the SES in Figure 3 is either replaced by a newly formed node *SimpleLayer_WS* or the node *WS1_WS*. Both nodes have to specify identical input and output interfaces in accordance with the replaced node *WS*. The interface compatibility is managed using basic models of the type *upward atomic model* (UAM) and *downward atomic model* (DAM). Additionally, these models manage different time granularities. The functionality of these systems as interface managers can be seen in the model structure of the executable DEVSimPy model presented in the next section.

5 SIMULATION MODEL GENERATION AND EXECUTION

As depicted in Figure 2, the model generation and execution is managed by an EC in the form of a Python script that calls API methods of the other framework tools and analyzes the returned results. Subsequently, the basic steps of model generation and execution for the DEVSimPy environment are discussed in more detail by the example of the watershed. Listing 2 shows a snippet of a Python script implementing an EC for investigating the watershed example.

We assume that an SES and the corresponding MB with the DEVSimPy basic models were created, as discussed in Section 4.2 and Section 4.3. Then, model generation starts in line 4 by setting the SESvars for deriving a system configuration and calling the pruning method of SESToPy in line 5, which returns the PES. The correctness of the SESvar settings is checked by SESToPy by evaluating the semantic conditions before starting the actual pruning operation. The resulting PES is illustrated in Figure 5. Since

SESMoPy can only process FPES, an FPES is created by calling SESToPy's flattening method in line 6. Then SESMoPy is called in line 7, the type of the model and the FPES are passed as arguments and a handle to the generated DEVSimPy model is returned. The results of these operations are given in Figure 6.

```
1  ...
2  SESfile = ...
3  if conditions_for_experiment:       #model variant with most details
4      SESvar = [WSLevel=1,MntLevel=1,BasinLevel=1,NumRepArea=2]       #set SESvars
5      PESfile = SESToPy("prune",SESvar,SESfile)                      #prune
6      FPESfile = SESToPy("flatten",PESfile)                          #flatten
7      h_model = SESMoPy("build","DEVSimPy",FPESfile)                 #build
8      tstart, tfinal = 0, 24*60
9      results = SESEuPy("simulate",h_model,"DEVSimPy",tstart,tfinal)  #execute
10 elif conditions_for_experiment:     #model variant with least details
11     SESvar = [WSLevel=0,MntLevel=0,BasinLevel=0,NumRepArea=1]       #set SESvars
12     PESfile = ...                                                  #prune
13 ...
```

Listing 2: Snippet of a Python-Script implementing an EC for the watershed example.

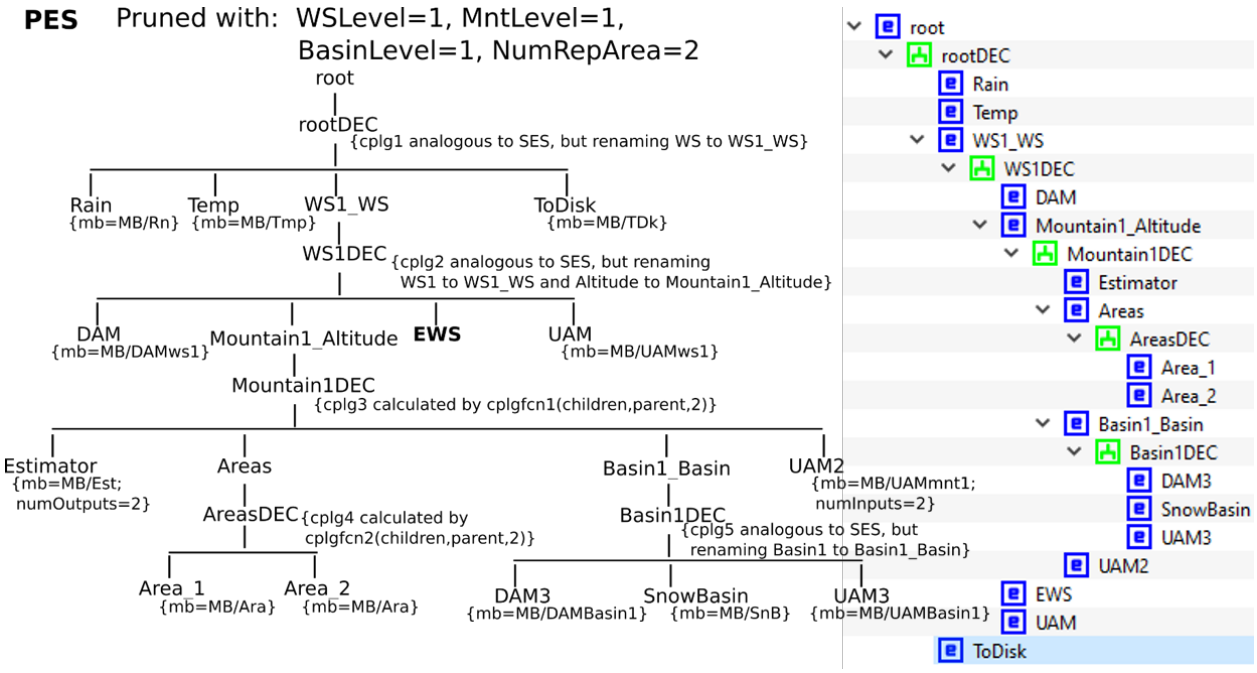

Figure 5: A PES tree of the watershed and its representation in SESToPy.

The generated DEVSimPy model maps the highest level of detail for the watershed referring to Section 3. After the model generation, the model execution parameters are set in line 8. In line 9 SESEuPy is called with the handle to the generated model, the type of the model and the model execution parameters. SESEuPy executes the simulation with DEVSimPy and the simulation results are returned.

The results can be analyzed using any number of other tools before a completely new model configuration is generated and executed. Beginning with line 11 the first lines of an example of the generation and execution of a second model variant are shown, which maps the lowest level of detail of the watershed example. Based on the different simulation results, the script in Listing 2 can implement any number of further calculations to solve a problem under investigation. In Santucci, Capocchi, and Zeigler (2016) numerical simulation results of differently detailed models of the watershed example are presented.

6 CONCLUSION AND FURTHER WORK

In the paper a Python-based framework for modeling and simulation of families of systems was introduced based on an extended SES/MB-based approach. In addition, the framework introduces an experiment control unit and a simulation execution unit supporting different target simulators. Using the DEVSimPy simulation environment, the integration with a concrete target environment was demonstrated. The usage of the FMI-based approach for integrating different simulators was not considered. The presented Python-based framework provides the simulation community an open source toolset for the investigation of variability systems, based on the SES/MB and DEVS theory. The tools are provided at RG CEA (2019) and SPE (2019). Moreover, this work opens several perspectives: (i) introduce learning algorithms to guide the pruning phase by improving the correlation between the selected structure and the system under study, and (ii) apply the proposed approach in the context of reinforcement learning to address the problems of large state space, which is an aggregation of states and actions, using the concept of probabilistic hierarchy (Kessler et al. 2017).

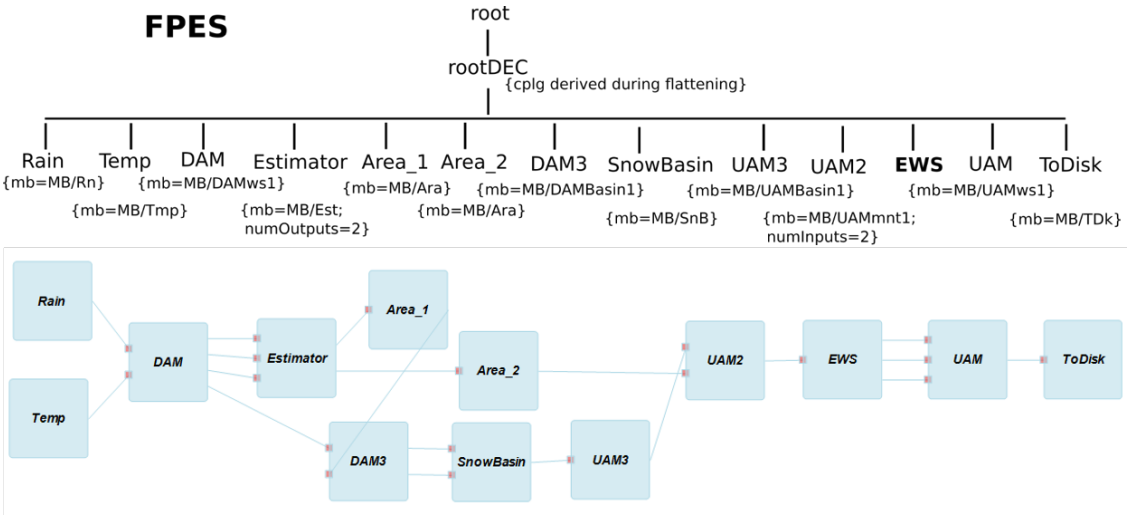

Figure 6: The FPES tree of the watershed and the generated DEVSimPy model.

REFERENCES

Bolduc, J.-S., and H. Vangheluwe. 2001. "The modelling and simulation package PythonDEVS for classical hierarchical DEVS". *McGill University, MSDL Technical Report MSDL-TR-2001–01*.

Capilla, R., J. Bosch, and K. C. Kang. (Eds.) 2013. *Systems and Software Variability Management, Concepts, Tools and Experiences*. Springer.

Capocchi, L., J. F. Santucci, B. Poggi, and C. Nicolai. 2011, June. "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems". In *Proc. of 20th IEEE International Workshops on Enabling Technologies*, pp. 170–175.

RG CEA 2017. "The SES Toolbox for MATLAB / Simulink Website". http://www.cea-wismar.de/tbx/SES_Tbx/sesToolboxMain.html. Accessed Sep. 09, 2018.

RG CEA 2019. "eSES/MB Infrastr.". http://www.cea-wismar.de/tbx/Py_eSESMB/. Accessed Mar. 11, 2019.

Coron, L., V. Andréassian, C. Perrin, M. Bourqui, and F. Hendrickx. 2014. "On the lack of robustness of hydrologic models regarding water balance simulation: a diagnostic approach applied to three models of increasing complexity on 20 mountainous catchments". *Hydrology and Earth System Sciences* vol. 18 (2), pp. 727–746.

Kessler, C., L. Capocchi, J.-F. Santucci, and B. Zeigler. 2017. "Hierarchical Markov Decision Process Based on Devs Formalism". In *Proceedings of the 2017 Winter Simulation Conference*, WSC '17, pp. 73:1–73:12. Piscataway, NJ, USA, IEEE Press.

Pawletta, T., D. Pascheka, A. Schmidt, and S. Pawletta. 2014, 08. "Ontology-Assisted System Modeling and Simulation within MATLAB/Simulink". *SNE Simulation Notes Europe* vol. 24, pp. 59–68.

Pawletta, T., A. Schmidt, U. Durak, and B. P. Zeigler. 2018, December. "A Framework for the Metamodeling of Multivariant Systems and Reactive Simulation Model Generation and Execution". *SNE Simulation Notes Europe* vol. 28 (1), pp. 11–18.

Rozenblit, J. W., and B. P. Zeigler. 1993. "Representing and constructing system specifications using the system entity structure concepts". In *Proceedings of the 25th Winter Simulation Conference, Los Angeles, California, USA, December 12-15, 1993*, pp. 604–611.

Santucci, J.-F., L. Capocchi, and B. P. Zeigler. 2016. "System entity structure extension to integrate abstraction hierarchies and time granularity into DEVS modeling and simulation". *SIMULATION* vol. 92 (8), pp. 747–769.

Schmidt, A., U. Durak, and T. Pawletta. 2016. "Model-based Testing Methodology Using System Entity Structures for MATLAB/Simulink Models". *SIMULATION* vol. 92 (8), pp. 729–746.

SPE 2019. "DEVSimPy". https://github.com/capocchi/DEVSimPy. Accessed Mar. 11, 2019.

Tendeloo, Y. V., and H. Vangheluwe. 2015. "PythonPDEVS: a distributed parallel DEVS simulator". In *Proceedings of the Symposium on Theory of Modeling & Simulation*, pp. 91–98. Alexandria, VA, USA, Society for Computer Simulation International.

Zeigler, B., T. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation*. Elsevier Science.

Zeigler, B. P., and P. E. Hammonds. 2007. *Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange*. Orlando, FL, USA, Academic Press.

Zeigler, B. P., S. Mittal, and M. K. Traore. 2018. "MBSE with/out Simulation: State of the Art and Way Forward". *Systems* vol. 6 (4).

AUTHOR BIOGRAPHIES

HENDRIK FOLKERTS is a PhD student at Wismar University, Germany. He is developing a tool for creating an SES including pruning and model generation for different simulation tools. His email address is hendrik.folkerts@cea-wismar.de.

THORSTEN PAWLETTA is a Professor for Applied Computer Science at Wismar University, Germany. His research interests lie in M&S theory and application in engineering. His email address is thorsten.pawletta@hs-wismar.de.

CHRISTINA DEATCU is a research engineer at Wismar University, Germany. Her research interests lie in theory of modeling and simulation, especially of discrete event and hybrid systems. Her email address is christina.deatcu@hs-wismar.de.

JEAN-FRANCOIS SANTUCCI has been a full professor in computer sciences at the University of Corsica (France), SPE CNRS 6134 Research Lab since 1995. He was assistant professor at the EERIE School, Nimes, France from 1987 to 1995. His email address is santucci@univ-corse.fr.

LAURENT CAPOCCHI has been an assistant professor in computer sciences at the University of Corsica (France), SPE CNRS 6134 Research Lab since 2007. His email address is capocchi@univ-corse.fr.