DEVSMAP: ON THE PATH OF STANDARDIZED DEVS MODEL REPRESENTATION

Sasisekhar Govind[a], Román Cárdenas[a b] and Gabriel Wainer[a]
[a]Advanced Real-Time Simulation Laboratory, Carleton University, Canada
[b]Integrated Systems Laboratory, Universidad Politécnica de Madrid, Spain

ABSTRACT

Experiment sharing and reproducibility are vital in modeling and simulation, facilitating collaboration and ensuring credibility. However, the nuances of experimentation are often lost, hindering reproducibility and sharing. The DEVS formalism addresses this by formally defining models and their experimental frames. Yet, the diversity of DEVS simulators introduces interoperability challenges, as models remain tool dependent. This research proposes DEVSMap: a language-agnostic representation of DEVS models. DEVSMap introduces a Map data structure using key-value pairs to represent DEVS tuples. It also defines experiments and experimental frames, formally specifying the model creation context. The representation is implemented in JSON, facilitating model repositories for broader sharing. Ultimately, this work aims to establish a comprehensive, language-agnostic DEVS model representation that could serve as a foundation for standardization within the DEVS community.

1 INTRODUCTION

Creating reusable software is of great importance for the reliability, maintainability and reducing the cost of a software project [1]. However, reusability is a complex endeavor [2]: "*In order for Computer Science research to be reproducible, several hurdles have to be cleared: the source code and test case data have to be available, the code has to build, the execution environment has to be replicated, the code itself has to run to completion, and accurate measurements (with respect to performance or other metrics) have to be collected.*" So, combining the experimental description, which involves test data, experimental context, execution environment etc., with the source code, should allow for experiment sharing and reproducibility.

In the area of Modeling and Simulation (M&S), this task is more complex due to the added complexities in system dynamic definition, simulation algorithm implementation, middleware, and experiments to be conducted. The Discrete Event System specification (DEVS) formalism has shown promising results in this area, as it not only rigorously formalizes the model and the experimental frame (EF) under which the models are built and results obtained, but it also provides algorithms that deterministically solve these models within such EF [3]. Furthermore, DEVS supports models across different platforms, and they can be implemented in various execution environments (desktop, RT, parallel, embedded, distributed) without the loss of fidelity in its behavior. This is due to the fact that the DEVS simulation algorithm has been formally verified, ensuring its correctness. This rigorous foundation makes it ideal for safety-critical systems and large engineering projects. A methodology for systems development called Modeling and Simulation Based Engineering (MSBE) proposes DEVS [4] as one of the suitable formalisms for this purpose.

Although they are based on the same foundations, DEVS simulators also suffer from a lack of model and experiment sharing in the M&S community. Despite the simulation algorithm being the same, different simulators implement the algorithm in different languages and for different purposes. One simulator might be in Java to promote platform independence, while the other might be in C++ to enable embedded development. This poses the problem of lack of code/model/ experiment sharing and reproducibility when moving from one simulator to another. Even though the model code may be different, the underlying formalism

Proc. of the 2025 Annual Simulation Conference (ANNSIM'25),
May 26-29, 2025, Universidad Complutense de Madrid, Madrid, Spain
J. L. Risco-Martín, G. Rabadi, D. Cetinkaya, R. Cárdenas, S. Ferrero-Losada, and A. Bany Abdelnabi, eds.

and hence the models themselves can be abstracted away from the code. A standard, language agnostic representation of models that can be parsed into code for any DEVS simulator could foster model sharing and formalize experiment reproducibility, allowing modelers to start from pre-defined, pre-validated models, and reusing them. This would also allow modelers around the globe to use the same model to conduct a variety of experiments (i.e., forest fires, migration of fauna, etc.) allowing for further validation, verification and calibration of the model to be a closer representation of the source system [5]. Another important feature of DEVS is the separation of model from simulator, which allows us to ignore the simulator (and hence the plethora of tools available) and focus solely on the representation of DEVS models.

To deal with these issues, we propose a language-agnostic representation of DEVS models using a structure of a collection of *{key: value}* pairs. Such a collection of key-value pairs (generally referred to as a Map or a Dictionary [6]), was named *DEVSMap*. Further, reflecting the hierarchical and modular nature of DEVS itself, *DEVSMap* is hierarchical in that the data structures maybe nested and modular in that the data structures can be re-arranged as necessary, for example, the *DEVSMap* of a leaf can be included in a *DEVSMap* of a tree in a forest, or the *DEVSMap* of a shrub. Further, to interoperate easily, The JavaScript Object Notation (JSON) file format is used to store and share the *DEVSMap*.

2 DEVS MODEL REPRESENTATION: DEVSMAP

As mentioned earlier *DEVSMap* is a proposed standard for representing Parallel DEVS (PDEVS) models and experiments. At its core, the standard is built around the Map or Dictionary data structure. Unlike traditional maps, *DEVSMap* restricts all the keys and values to strings.

The standard promotes the idea of a centralized model repository, allowing modelers with any DEVS simulator to fetch, modify, execute, and update models. In doing so, *DEVSMap* extends the advantages of open-source collaboration to the DEVS domain.

The *DEVSMap* standard defines seven types of JSON files that must be created either by an application or by the user:

- <model name>_atomic.json
- <model name>_coupled.json
- <experiment name>_experiment.json
- <set name>_definition.json
- <identifier>_init_state.json
- <identifier>_param.json
- <name of model>_metadata.json

All placeholders enclosed in angular brackets (<>) are to be replaced by the user or customized by the application producing these files.

All definitions of atomic models in your project must be within the *<model name>_atomic.json*, where the *<model name>* should be appropriately specified. All definitions of coupled models within your project must be within the *<model name>_coupled.json*. The experiments to be carried out must be defined in the *<experiment name>_experiment.json*, where *<experiment name>* helps differentiate between multiple experiments to be run. All the sets used in your model and EF (domains of state variables, domains of ports etc.) are to be defined in *<set name>_definition.json*. Since the standard tries to adhere to the mathematical formalism and remain language agnostic, you have to define sets instead of datatypes. An interpreter of this standard would convert the sets to datatypes. *<identifier>_init_state.json* and *<identifier>_param.json* define the initial conditions of the model (initial states and parameters) under test. And finally, the *<name of model>_metadata.json* provides important information regarding the model itself, like model name, author, date etc. The specific files required depend on whether the goal is to share models, share experiments, or upload content to a model repository.

For the sole purpose of model sharing, each of the atomic model JSON definitions are standalone and can be sent to other modelers along with (if required) the *<set name>_definition.json* files corresponding to the atomic model. For sharing entire experiments, all the different file types are necessary to ensure that the experiment does not fundamentally change between users. The *<name of model>_metadata.json* is most important for indexing in a model repository.

3 ATOMIC MODEL REPRESENTATION

The *<model name>_atomic.json* is a JSON file that represents an atomic DEVS model. The entire DEVS-Map definition in JSON format can be found in the Appendix along with an example. The following paragraphs will walk through the definition in detail. A DEVS atomic model[3] is defined as:

$$AM_p = <X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta>$$

Where AM_p is an atomic model, whose parameters are 'p', X is the set of input ports, Y is the set of output ports, S is the state set, δ_{int} is the internal transition function, δ_{ext} is the external transition function, δ_{con} is the confluent transition function, λ is the output function, and ta is the time advance function. Building on these foundations, we propose a key-value pair-based representation of this model.

Following, we define and explain the various components of this representation. The topmost hierarchy contains three keys. They are: *"<name of the atomic model>"*, *"parameters"*, and *"include_sets"*.

The first key is the name of the atomic model itself. Per the template, it is *"<name of the atomic model>",* and it is to be replaced by the appropriate name. This key will be different for different atomic models. The name will be used to reference the atomic model within the coupled model. For example, if an atomic model is called *counter*, *"<name of the atomic model>"* is to be defined in the *<model name>_atomic.json* as:

```
"counter": {…}
```

The value associated with this key is an object that contains the definition of the atomic model as per the octuple. Which are: *"x"*, *"y"*, *"s"*, *"delta_int"*, *"delta_ext"*, *"delta_con"*, *"lambda"*, and *"ta"*. All these must be defined in the *<model name>_atomic.json* within *"<name of the atomic model>"*.

"x" is used to define the input ports of the atomic model. As its value, *"x"* expects a collection of key-value pairs that list out the various ports and their corresponding domains, as follows:

```
"x": {<name of the input port>: <domain of the input port>, ...}
```

Here, "<domain of the input port>" must be a *mathematical set*. Further, this set must be defined in the *<set name>_defintions.json* (explained further below). For example, if we have an atomic model whose inputs are:

$X = \{cd, inc \mid cd \in \{0, 1\}, inc \in \mathbb{R}^{+,0}\}$

Then:

```
"x": {"cd": "boolean", "inc": "unsigned_R"}
```

Where set *boolean* and set *unsigned_R* have been defined in a JSON file(s) that has been included in *"include_sets"*.

"y" is used to define the output ports of the atomic model, using a collection of key-value pairs that list out the various ports and their corresponding domains, as follows:

```
"y": {<name of the output port>: <domain of the output port>, ...}
```

For example, if you have an atomic model whose inputs are $Y = \{count\_value \mid count\_value \in \mathbb{R}\}$, then:

```
"y": {"count_value": "R"}
```

"s" is used to define the state variables of the atomic model. As its value, *"s"* expects a collection of key-value pairs that list out the various state variables that make up the state set, as follows:

```
"s": {<name of the state variable>: <domain of the state variable>, ...}
```

For example, if you have an atomic model whose state set is defined mathematically as:

$$S = \{(count, delta, inc\_dec, \sigma) \mid count \in \mathbb{R}, delta \in \mathbb{R}^{+,0}, inc\_dec \in \{0, 1\}, \sigma \in \mathbb{R}^{+,0}\}$$

Then:

```
"s": {"count": "R", "delta": "unsigned_R", "inc_dec": "boolean", "sigma": "unsigned_R"}
```

"delta_int" is used to define the internal transition function. It expects a collection of key-value pairs. The keys within *"delta_int"* are required to be expressions of state variables that compute to either true or false, and the values are again a collection of key-value pairs that represent a new state set. For any given state, one and only one of the keys should evaluate to True, all the others must evaluate to False. Furthermore, *"otherwise"* is a valid key that evaluates to True if all the other keys evaluate to False. *"otherwise"* is mandatory to promote model completeness.

```
"delta_int": {
  "<expression of state variables that evaluate to true or false>": {
    "<state variable name>": "<expression to compute new value>",...  },
  ⋮
  "otherwise": {
    "<state variable name>": "<expression to compute new value>",...  }
}
```

For example, if we have an internal transition function defined as:

$$\delta_{int}(count, delta, inc\_dec, \sigma) = \begin{cases} (count + delta, delta, inc\_dec, \sigma), & inc\_dec = 0 \\ (count - delta, delta, inc\_dec, \sigma), & inc\_dec = 1 \end{cases}$$

Then:

```
"delta_int": {
  "inc_dec == 0": { "count": "count + delta"},
  "inc_dec == 1": { "count": "count - delta"},
  "otherwise":    {"inc_dec": "0"}
}
```

Note that within the new state set, if state variables are omitted, their values remain unchanged through the transition. A list of valid operators is given in the Appendix. Further, if required, the conditions are allowed to be nested as such:

```
"delta_int": {
  "condition_1": { ...
      "condition_n": {
        "<state variable>": "<expression to compute new value>",... }, ...
    },
  },
  "otherwise": {
    "<state variable>": "<expression to compute new value>",...     }
}
```

The *"delta_ext"* follows the same pattern and is used to define the external transition function. It expects a collection of key-value pairs, where the keys are an expression of state variables, elapsed time and inputs, and the values are the new state set, as follows:

```
"delta_ext": {
  "<expression of s, e and x that evaluates to true or false>": {
    "<state variable name>": "<expression to compute new value>",...  },
  ⋮
  "otherwise": {
    "<state variable name>": "<expression to compute new value>",...  }
```

```
}
```

For example, if you had an external transition function defined as:

$$\delta_{ext}\big((count, delta, inc\_dec, \sigma), e, (inc, cd)\big) = \begin{cases} (count, delta, cd, \sigma - e), & inc = \emptyset, cd \neq \emptyset \\ (count, inc, inc\_dec, \sigma - e), & inc \neq \emptyset, cd = \emptyset \\ (count, inc, cd, \sigma - e), & otherwise \end{cases}$$

Then:

```
"delta_ext": {
  "inc_value.bagSize() != 0": {"delta": "inc_value.bag(-1)"},
  "change_dir.bagSize() != 0": {"inc_dec": "change_dir.bag(-1)"},
  "otherwise": {
    "inc_dec": "change_dir.bag(-1)",
     "delta": "inc_value.bag(-1)"
  }
}
```

Note that *.bagSize()* and *.bag(<index>)* are operators used to manipulate a message bag. The rules of indexing a bag are described further in the Appendix, but they are similar to the indexing rules of *lists* in the Python language.

"delta_con" is similar to both *"delta_ext"* and *"delta_int"*, and is used to define the confluent transition function. It expects, as its value, a collection of key-value pairs, where the keys are an expression of state variables and inputs, and the values are the new state set. Per the PDEVS formalism, it is mandatory to define the confluent transition function. In DEVSMap, the confluent transition function is defined as:

```
"delta_con": {
    "<expression of s and x that evaluates to true or false>": {
        "<state variable name>": "<expression to compute new value>",... },
    "otherwise": {
    "<state variable name>": "<expression to compute new value>",...  }
}
```

"lambda" is used to represent the output function. The keys are expressions of state variables that compute to True or False. However, unlike the transition functions, the values are themselves a collection of key-value pairs that map output ports to expressions of state variables used to compute the output at that port. Like the transition functions, *"otherwise"* is a valid keyword:

```
"lambda": {
  "<expression of s that evaluates to true or false>": {
    "<output port name>": "<expression of s to compute output>",...  },
  "otherwise": {
    "<output port name>": "<expression of s to compute output>",...  }
}
```

For example, if you had an output function that could be defined as *λ(s) = count*

Then:

```
"lambda": {
  "otherwise": {
    "count_value": "count" }
}
```

Note that here, since no conditions are imposed on the output, the *"otherwise"* keyword is used to always put the output *count* at port *count_value*.

"ta" is used to represent the time advance function of the atomic model. *"ta"* like *"lambda"* expects key-value pairs as its value, where the keys are expressions of s that compute to true or false. But unlike the output functions, the values are expressions that compute the time advance of the state. Because of this the expression to compute the time advance value must result in a value that belongs in $\mathbb{R}^{+,0}$:

```
"ta": {
  "<expression of s that evaluates to true or false>": "<expression of s>"
```

```
  ⋮
  "otherwise": "<expression of s>"
}
```

For example, if you had a time advance function that can be defined as $ta(s) = \sigma$

Then:

```
"ta": {
  "otherwise": "sigma"  }
```

Coming out of *"<name of the atomic model>"*, the following keys are again in the topmost hierarchy.

The *include_sets* key uses an array of either local or remote *<set name>_definitions.json* files. These files define all the sets that are used within the atomic model. For example: if you have an integer state variable, you are required to include *"integer_definitions.json"* in your *<model name>_atomic.json* where *"integer_definitions.json"* contains the definition for set $\mathbb{Z}$:

```
"include_sets": [<set repository file>, <set repository file>, …]
```

It is essential to include all set definitions required for the atomic model, ensuring that the domain sets of each state variable, parameter, input port, and output port are included in this array.

The *parameters* key has, as its value, an object that contains all the parameters of the atomic model along with the corresponding domains. If the model is not parameterized, this object can be left blank:

```
"parameters": {<parameter name>: <parameter domain>, …}
```

For example, if your atomic model had a parameter *delay_time* $\in \mathbb{R}$, and *num_outputs* $\in \mathbb{N}$:

```
"parameters": {"delay_time": "R", "num_outputs": "N"}
```

3 COUPLED MODEL REPRESENTATION

The *<model name>_coupled.json* is a JSON file that represents a coupled DEVS model. Again, the entire DEVSMap definition in JSON format can be found in the Appendix along with an example. The following paragraphs will walk through the definition in detail. A DEVS coupled model is defined as:

$$CM = < X, Y, D, M_d, EIC, EOC, IC >$$

Where *CM* is the coupled model, *X* is the set of input ports, *Y* is the set of output ports, *D* is the set of component names, M_d for $d \in D$ are DEVS models, *EIC* is external input coupling, *EOC* is external output coupling, and *IC* is internal couplings. A coupled model representation has two topmost keys: *"<name of the coupled model>"*, and *"include_sets"*. The first key is the name of the coupled model itself. For example, if your coupled model is called *counter_display*, then this key would be defined as:

```
"counter_display": {…}
```

The value associated with this key is an object that contains the definition of the coupled model as per the septuple. Which are: *"x"*, *"y"*, *"components"*, *"eic"*, *"eoc"*, *"ic"*. Like the atomic model definition, *"x"*, and *"y"* are keys that define the port names, and the domain associated with the values coming into or going out of the port.

```
"x": {<name of the input port>: <domain of the input port>, ...},

"y": {<name of the output port>: <domain of the output port>, ...}
```

The *"components"* key defines the various components that make up the coupled model. As its value it expects a collection of key-value pairs that associate a DEVS model with a unique alphanumeric component identifier. This identifier is required to be unique to allow for hierarchy and modularity. The *"components"* key satisfies the role of *D* and M_d in the coupled model definition:

```
"components": {
 "<name of the DEVS model as defined in its JSON file>": "<unique component identifier>",...}
```

The *"eic"* key is used to represent the external input couplings of the DEVS model. As its value, *"eic"* expects an array of collections of key-value pairs:

```
"eic": [
  {
    "port_from": "<an input port of this coupled model>",
    "port_to": "<an input port of a component defined in "components">",
    "component_to": "<identifier of the component whose input port is in "port_to">"
  },
    ⋮
],
```

The *"eoc"* key is used to represent the external input couplings of the DEVS model. As its value, *"eoc"* expects an array of collections of key-value pairs:

```
"eoc": [
  {
    "port_from": "<output port of a component defined in "components">",
    "port_to": "<an output port of this coupled model>",
    "component_from": "<identifier of the component whose output port is in "port_from">",
  },
    ⋮
],
```

The *"ic"* key is used to represent the input couplings of the DEVS model. As its value, *"ic"* expects an array of collections of key-value pairs. *"ic"* is to be defined in the *<model name>_coupled.json* as:

```
"ic": [
  {
    "port_from": "<output port of a component defined in "components">",
    "port_to": "<an input port of a component defined in "components">",
    "component_from": "<identifier of the component whose output port is in "port_from">",
    "component_to": "<identifier of the component whose input port is in "port_to">"
  },
    ⋮
],
```

4 DEFINING DOMAIN SETS

All the sets used in the model definitions (for example, as domains of the state set) have to be defined in the *<set name>_defintions.json* file. There are a few basic sets that we do not have to specifically define, these are: *integer* ($\mathbb{Z}$), *unsigned_integer* ($\mathbb{Z}^{+,\mathbf{0}}$), *floating_point* ($\mathbb{R}$), *boolean* ($\{0, 1\}$), *char* (Set of character ASCII values) etc., the exhaustive list can be found in the Appendix.

The topmost key in this file is: *"<name of the set>"* which is to be replaced by the appropriate name. As its value, it expects a collection of key value pairs. The keys are the elements of the set, and as its value, it expects four parameters: *"domain"* which is the domain of the element, *"min"* and *"max"* which is the range of the element and *"dimension"* which defines the algebraic dimension of the element as an array. The contents of the file are as follows:

```
{
  "<name of the set>": {
        "<element of the set>": {
            "domain": "<domain of this element>",
            "min": "<smallest value in the set>",
            "max": "<Largest value in the set>",
            "dimension": ["integer value or n", …]
         }, …
   }
}
```

For example, if you had a set defined as: $A = \{(w, t) \mid w \in \mathbb{R}^{n \times n}, t \in \mathbb{R}^{+,0}\}$ then:

```
{
```

```
  "A": {
    "w": {
      "domain": "floating_point",
      "min": "-inf",
      "max": "+inf",
      "dimension": ["n", "n"]},
    "t": {
      "domain": "floating_point",
      "min": "0",
      "max": "+inf",
      "dimension": ["1"]}
  }
}
```

Note that in the example, the dimension of element *w* is set to be *n*x*n*. This would signify a *dynamic array* (for example, *std::vector<>* in C++, or *ArrayList* in Java).

5 EXPERIMENTAL FRAMES AND EXPERIMENT REPRESENTATION

We have gone through a normalized notation that includes all that is required for model sharing. However, we need a few more file types if we want to share experiments. The first aspect to take into consideration when building a DEVS model is to provide context and considering the model's EF, which can be defined as a specification of the conditions under which the system is observed or experimented with [3].

An EF can be formalized as: $<T, I, C, O, \Omega_I, \Omega_C, SU>$, where T is a time base, I is a set of input variables that influence the system, C is a set of run control variables, O is a set of output variables that the system of interest influences, Ω_I is a set of admissible input segments (acceptable input values for a given time), Ω_C is a set of admissible control segments, and SU is the summary mappings [5].

Further, the realization of such a frame can be done using three main DEVS models: *generator*, *acceptor*, *transducer*. The *generator* is an active DEVS model that generates values y at time t, such that $(y, t) \in \Omega_I$. The *acceptor* is a passive DEVS model that accepts input values x at time t, such that $(x, t) \in \Omega_C$. The *transducer* is a passive DEVS model that realizes a summary mapping $\in$ SU.

Traoré and Muzy [7] extended this and provided a generalization of EFs and follows the DEVS specification hierarchy to introduce frame interface, frame behavior, and frame system. The final level of the hierarchy proposed in [7] is the frame system defined as: $<T, I_M, I_E, O_M, O_E, \Omega_M, \Omega_E, \Omega_C, D, C_d$, *CPIC, EICC, POCC, CEOC, CCC*$>$, where T is a time base, I_M is a set of frame-to-model input variables, I_E is a set of input variables to the EF, O_M is a set of model-to-frame output variables, O_E is a set of output variables of the EF, Ω_M is a set of admissible input segments of the model, Ω_E is a set of admissible input segments to the EF, Ω_C is a set of admissible output segments expected of the model, D is the set of indices of frame component models, C_d is the set of models itself, *CPIC* is the Control-to-Plug-in-Input coupling, *EICC* is the External-Input-to-Control coupling, *POCC* is the Plug-in Output-to-Control coupling, *CEOC* is the Control-to-External-Output coupling, and *CCC* is the Control-to-Control coupling. The realization of this is no different from that mentioned earlier (since it is an extension). Additionally, the *generator*, *acceptor*, and *transducer* would be the components defined in D and C_d.

Finally, Denil et.al. in [8] extended Traore and Muzy's work to add a few more components as seen in Figure 1. It includes a set of Solver(s) for the source system and the EF, with P_S being the set of parameters required by the solvers. Next, the Experimental Setup is the frame itself with O_E and I_E per Traore and Muzy's definition, S_E is the set of signals that facilitate white box testing, P_E is the set of parameters of the EF, and IC_E is a set of initial conditions of the EF. Next, we see the Model/System with input and output ports I_M and O_E, model parameter set P_M, set of initial conditions IC_M, and S_M is a set of signals that facilitate white-box testing. Finally, the Observation Collector is an element that does white-box testing, with C_M and C_E being the ports through which the signals from the model and EF are sent through.

The Solver(s) and Observation Collector definitions are intended for future consideration, but P_E, IC_E, P_M, and IC_M are important to be considered in the DEVSMap representation. DEVSMap considers the latest EF definition as provided by Denil et. al. We have decided to split the EF into two: a coupled model definition of the EF (with appropriate atomic model files) and an experiment file.

To draw parallels between a DEVS coupled model and the frame system, consider a coupled model E = <X, Y, D, C_d, IC, EIC, EOC>, where X equivalent to I_E, Y is equivalent to O_E, D and C_d are atomic models *generator*, *acceptor* and *transducer*, CCC is IC, EICC is EIC, CEOC is EOC. This model E is then to be represented as a DEVSMap coupled JSON file. The remaining attributes: T (which is real since we are dealing with DEVS) CPIC, POCC, I_M, O_M, but P_E, IC_E, P_M, and IC_M are defined in the <*experiment name*>*_experiment.json*. [8] also points out that the definitions by Zeigler, Traore and Muzy neglect time span of the experiment, hence, this is added to the <*experiment name*>*_experiment.json* as well.

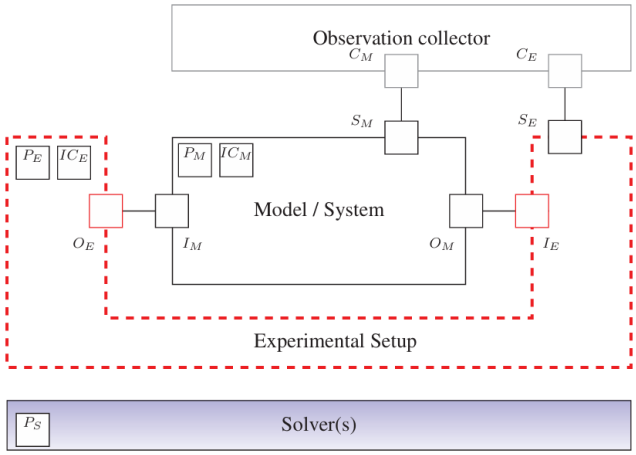

Figure 1: Experimental Frame per [8].

<*experiment name*>*_experiment.json* has five keys in its topmost hierarchy: *"model_under_test", "experimental_frame","cpic","pocc","time_span"*.

The *"model_under_test"* key is used to represent the model which can be experimented on. It expects a collection of key-value pairs that represent the top model files, the initial condition file, and the parameters file. It is defined as such:

```
"model_under_test": {
  "model": "<filename of top coupled model>",
  "initial_state": "<filename of initial state of model>",
  "parameters": "<filename of parameters of model>"
}
```

The *"experimental_frame"* key is used to define the EF. It expects a coupled model file, an initial condition file, and parameters. It is defined as:

```
"experimental_frame": {
  "model": "<filename>",
  "initial_state": "<filename>",
  "parameters": "<filename>"
}
```

The *"cpic"* key is used to represent the couplings between the output of the EF and the input to the top coupled model. It is defined as:

```
"cpic": [
    {   "port_from": "<an output port of the EF model>",
        "port_to": "<an input port of the model under test>"    }, …
]
```

Similarly, the *"pocc"* key is used to represent the couplings between the output of the top coupled model and the input of the EF. It is defined as:

```
"pocc": [
    {   "port_from": "<an output port of the model under test>",
        "port_to": "<an input port of the EF model>"  },…
]
```

Finally, *"time_span"* is used to represent the length of the experiment. As its value, it either a rational value or *inf*.

```
"time_span": "a rational number or inf"
```

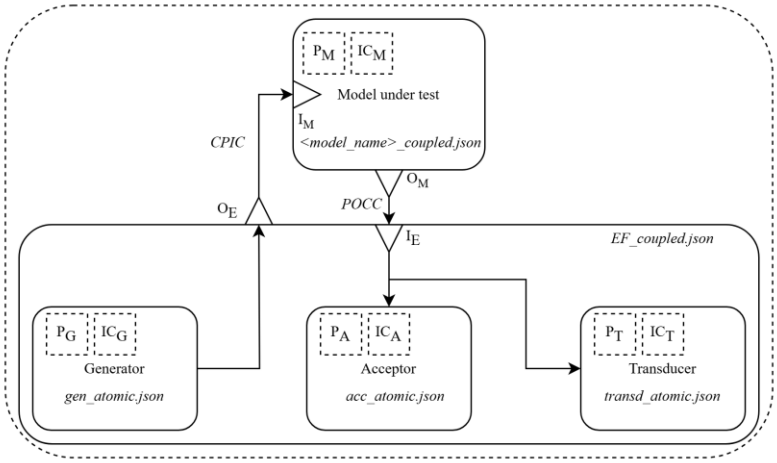

Figure 2: DEVSMap Experimentation setup

Figure 2 shows the complete experimental setup per the *DEVSMap* specification. Here, we see the *EF_coupled.json*, which is the *DEVSMap* that defines the EF coupled model. Within the coupled model, we see three atomic (could also be coupled) models representing a *generator*, *acceptor* and *transducer*. Along with the atomic models, we can also see P_G, IC_G, P_A, IC_A, P_T, IC_T, which are the parameters and initial conditions of the EF. The inputs and outputs of the EF depicted as O_E and I_E. Above the EF coupled model, we see the model of the source system which is being experimented on. The inputs and outputs of this model are depicted using I_M and O_M. P_M and IC_M represent the parameters and initial conditions of the model under test. Finally, the couplings between the model and the EF is shown using *CPIC* and *POCC*.

Drawing parallels between the diagram given in Figure 2 and *DEVSMap*, we see that the EF is defined in its own coupled model file *EF_coupled.json*, the model under test is also its own coupled model file *<model_name>_coupled.json* and the interface between the model under test and the EF, done using CPIC and POCC is defined in the *<experiment_name>_experiment.json* file. The file also defines P_G, IC_G, P_A, IC_A, P_T, IC_T, P_M, and IC_M.

6 PARAMETER AND STATE INITIALIZATION

The initial conditions (states) and the parameter values of the model are defined in the *<identifier>_init_state.json* and *<identifier>_param.json* files. The *<identifier>* is to be replaced with an appropriate name to help one differentiate between multiple files. The structure of both these files are identical. Starting off with initializing the parameters, the topmost key is *"parameters"* within which, as its value, are a collection of key value pairs that represent, hierarchically, the different models that constitute the top coupled model. Here is a prototype representation:

```
{
  "parameters": {
    "<name of the top coupled model>": {
      "<component_id of an atomic model>" : {"<name of parameter>": "<value of parameter>", …},
```

```
      "<component_id of an atomic model>" : {"<name of parameter>": "<value of parameter>", …},
      "<component_id of a coupled model>": {…}, …
    },
    "<component_id of a coupled model>": {…}, …
  }
}
```

Here, *component_id* refers to the id given to the model within the coupled model definition file. It is important to use the *component_id* and not the model's name to differentiate between multiple "instances" of the same atomic or coupled model. Another thing to note is that the *component_id* of the top coupled model is never defined, and so the name of the top coupled model is used in the topmost level. Moving down the hierarchy, when an atomic model is reached, the parameter and its value is defined.

Similarly, *<identifier>_init_state.json* represents the initial states of the models as follows:

```
{
  "init_states": {
    "<name of the top coupled model>": {
      "<component_id of an atomic model>" : {"<state variable>": "<value>", …},
      "<component_id of an atomic model>" : {"<state variable>": "<value>", …},
      "<component_id of a coupled model>": {…}, …
    },
    "<component_id of a coupled model>": {…}, …
  }
}
```

We see that the topmost key is *"init_states"*, and as its value it expects a collection of key-value pairs representing the hierarchical structure of a DEVS model.

7 META DATA AND INDEXING

In addition to formally representing models and experiments, it is equally important to provide a human readable description for documentation purposes and to facilitate indexing in a model or experiment repository. Research in various fields[9] have also shown that such meta-data description is important for reproducibility of experiments, and to document small, extremely critical details that would otherwise be lost.

This descriptive model data is represented in *<model_name>_metadata.json* as such:

```
{
  "model": "<name of the model>",
  "model_description": "<description of the model>",
  "experiment": "<name of the model>",
  "experiment_description": "<description of the experiment>",
  "developer_name": "<name of the developer>",
  "developed_date": "<date of model creation>",
  "organization": "<organization/ affiliations of the model developer>",
  "keywords": ["<keyword that can be used to index the model>", ...],
  "top_model_hierarchy": ["<component_id of atomic/ coupled model in the top model>", ...]
}
```

8 CONCLUSION

In this research, we have addressed the issues of reusability and reproducibility of DEVS models, by proposing the DEVSMap representation. Researchers [11] convey the importance of such platform independent representations and point out two main types of reusability and interoperability: simulator based interoperability and model based interoperability.

Various researchers have proposed numerous representations of DEVS models for model based interoperability, chief among them being DEVSML [10]. DEVSML proposed an eXtensible Markup Language (XML) based representation of DEVS atomic and coupled models, and rides on the JAVAML canonical representation of Java source code. This not only allows for platform independence, but it also allows for

integration with Service Oriented Architecture (SOA) framework, which then promoted simulator interoperability. DEVSMap, being model representation as key-value pairs, also can be represented as XML files, hence making it compatible with the DEVSML framework with minor modifications. However, it extends on DEVSML by defining and proposing a representation of experimental frames along with their models. This allows for true reproduction of the experiment as the modeler intended, since the modeler can provide the context in which their model is valid.

DEVSMap proposes a representation for atomic and coupled models, experimental frames and experiments, along with model metadata allowing it to be indexed within a model repository, as proposed in [12]. It also promotes model reuse, as DEVSMap allows flexibility of experiment definition. That is, DEVSMap inherently allows the modeler to 'mix and match' models and experiments. This modularity does come with the cost of increased files per model or experiment, but with modern advancements in computing technology, this is no longer an issue.

In the future, we would like to extend our work to implement simulator interoperability, analyzing its compatibility with the DEVS/SOA [13] simulation platform. Hence, we would like to propose this representation as a standard to unify the DEVS experimentation process.

A APPENDIX

An example model representation is described in this (https://github.com/Sasisekhar/DEVSMap_parser.git) GitHub repository. The code in the repository is a work-in-progress (as of March 2025) C++ Parser library to parse DEVSMap into DEVS models. The repository also contains an example of the implementation of the Parser for the Cadmium V2 DEVS simulation environment.

REFERENCES

[1] Ted Biggerstaff and Alan Perlis (Eds). 1989. Software Reusability, Volume 1 & 2, ACM Press, NY.

[2] Collberg, C., Proebsting, T. and Warren, A.M., 2015. Repeatability and benefaction in computer systems research. *University of Arizona TR*, *14*(4), pp.1-68.

[3] Zeigler, B.P., Praehofer, H. and Kim, T.G., 2000. Theory of modeling and simulation. Academic press.

[4] Gianni, D., D'Ambrogio, A. and Tolk, A. eds., 2014. Modeling and simulation-based systems engineering handbook. CRC Press.

[5] Zeigler, B.P., 1984. Multifacetted modelling and discrete event simulation. Academic Press Professional, Inc..

[6] Brass, P., 2008. Advanced data structures (Vol. 193). Cambridge: Cambridge university press.

[7] Traoré, M.K. and Muzy, A., 2006. Capturing the dual relationship between simulation models and their context. Simulation Modelling Practice and Theory, 14(2), pp.126-142.

[8] Denil, J., Klikovits, S., Mosterman, P.J., Vallecillo, A. and Vangheluwe, H., 2017, April. The experiment model and validity frame in M&S. In *Proceedings of the Symposium on Theory of Modeling & Simulation* (pp. 1-12).

[9] Shpilker, P., Freeman, J., McKelvie, H., Ashey, J., Fonticella, J.M., Putnam, H., Greenberg, J., Cowen, L., Couch, A. and Daniels, N.M., 2022. MEDFORD: A human-and machine-readable metadata markup language. Database, 2022, p.baac065.

[10] Mittal, S., Martin, J.L.R. and Zeigler, B.P., 2007. DEVSML: Automating DEVS simulation over SOA using transparent simulators. *DEVS Symposium*.

[11] Wainer, G.A., Al-Zoubi, K., Hill, D.R., Mittal, S., Martín, J.L.R., Sarjoughian, H., Touraille, L., Traoré, M.K. and Zeigler, B.P., 2018. Standardizing DEVS model representation. In *Discrete-Event Modeling and Simulation* (pp. 427-458). CRC press.

[12] Chreyh, R. and Wainer, G., 2009, March. Cd++ repository: an internet based searchable database of devs models and their experimental frames. In Proceedings of the 2009 Spring Simulation Multiconference (pp. 1-8).

[13] Mittal, S., Risco-Martín, J.L. and Zeigler, B.P., 2009. DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVS unified process. Simulation, 85(7), pp.419-450.

AUTHOR BIOGRAPHIES

SASISEKHAR GOVIND is a Ph.D. student in Computer Engineering at Carleton University under the supervision of Dr. Gabriel Wainer. His research interests include modeling and simulation, embedded systems, and distributed simulations. His email address is: sasisekharmangalamgo@cunet.carleton.ca.

ROMAN CARDENAS is an Assistant Professor in the Department of Electronic Systems at Universidad Politécnica de Madrid (UPM), Spain, where he obtained a Ph.D. in Electronic Systems Engineering in Cotutelle with Carleton University (CU). His research interests include modeling and simulation with applications in the IoT domain. His email address is r.cardenas@upm.es.

GABRIEL WAINER is a Professor in the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada). His current research interests are related to modeling methodologies and tools, parallel/distributed simulation, and real-time systems. He is a Fellow of SCS. His e-mail is gwainer@sce.carleton.ca. His website is www.sce.carleton.ca/faculty/wainer.