Externalizing Patterns for Simulations in Software Engineering of Systems-of-Systems

Valdemar V. Graciano Neto, Wallace Manzano*
University of São Paulo, São Carlos, Brazil
{valdemarneto,wallace.manzano}@usp.br

Adair José Rohling, Mauricio Gonçalves Vieira Ferreira
INPE - Instituto Nacional de Pesquisas Espaciais
São José dos Campos, Brazil
adairrohling@gmail.com,mauricio@ccs.inpe.br

Tiago Volpato, Elisa Yumi Nakagawa
University of São Paulo, São Carlos, Brazil
{tiagovolpato}@usp.br

ABSTRACT

Systems-of-Systems (SoS) often support critical domains. They must be trustworthy, i.e., they must keep their operation in progress, being not subject to failures, as they can cause potential damages and hazards to human integrity. Simulations are a recurrent approach in SoS development, as they can anticipate potential failures, consequently increasing the level of trustworthiness and quality exhibited by a SoS. Nevertheless, simulation is still software and demands engineering. Moreover, many simulation formalisms are not trivial of specifying, sometimes tangling software an hardware details to program an executable simulation. Thus, the aim of this paper is contributing for software engineering of SoS by externalizing two patterns for the conception of SoS simulations. We evaluated our patterns by applying them in a case study in two different domains. For both, patterns were successfully applied during automatic generation of functional code, supporting the execution of SoS simulations and prediction of SoS behavior at design-time.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; **Simulator / interpreter**; *Source code generation*;

KEYWORDS

System-of-Systems, Simulation, Software Architecture, Pattern

ACM Reference format:
Valdemar V. Graciano Neto, Wallace Manzano, Adair José Rohling, Mauricio Gonçalves Vieira Ferreira, and Tiago Volpato, Elisa Yumi Nakagawa. 2018. Externalizing Patterns for Simulations in Software Engineering of Systems-of-Systems. In *Proceedings of SAC 2018: Symposium on Applied Computing, Pau, France, April 9–13, 2018 (SAC'18),* 8 pages.
https://doi.org/10.1145/3167132.3167313

*First author is also with Universidade Federal de Goiás, Goiânia, Brazil; and Université de Bretagne-Sud, Vannes, France

SAC'18, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167313

1 INTRODUCTION

Systems-of-Systems[1] (SoS) are an alliance of pre-existing systems termed as constituents that cooperate to achieve global missions [23]. They often support critical domains, such as response to disaster events, e.g., forest fires and floods [29]. Thus, they must be conceived to be trustworthy, exhibiting a high-degree of reliability in its operation, and following well-established principles of engineering and validation [13, 24]. Simulations are a recurrent practice within the SoS development life cycle [12], as (i) they imitate the surrounding environment and the operation of a system in the real-world, (ii) support the observation of the effects to draw inferences concerning the operational characteristics of the real system [3], (iii) enable the prediction and correction of errors, and (iv) support the observation of expected and unexpected emergent behaviors of an SoS [9]. Nevertheless, simulation is also software. It often relies on dozens of lines of code, demanding techniques for its engineering. Moreover, simulations are often driven by labeled state machines, based on discrete input and output events, such as Discrete Event Systems Specification (DEVS) [2, 30]. However, some of the instructions are conflicting with each other, and the amount of lines of code can be enormous, which bring difficulties and high costs for production and maintainability. Moreover, state machines that represent multiples behaviors can be difficult to handle with manual approaches due to large dimension issues, which makes the production of these codes repetitive and error-prone.

Under this perspective, the identification of patterns can aid the conception of simulations for SoS, supporting automatic generation of these codes from specifications in high level of abstraction, such as architectural specifications of software of SoS. Thus, the aim of this paper is contributing for software engineering of SoS by externalizing two patterns for the conception of simulations of SoS. We established sets of instructions in DEVS that correspond to recurrent specifications of input and output events, avoiding conflicts that can make a simulation not-executable or full of failures.

This paper is structured as follows: Section 2 briefly outlines the foundations; Section 3 introduces the patterns; Section 4 reports our results, and Section 5 brings final remarks and future work perspectives.

[1]For sake of simplicity, henceforth, we use this term to express both singular and plural.

2 FOUNDATIONS

Constituents usually comprise physically decoupled entitities, with software, hardware, and stakeholders involved. SoS are often developed under the framework of SoS Engineering (SoSE), a branch of Systems Engineering that investigates specifically the engineering of SoS. However, software has become a ubiquitous element in SoS as constituents embed software and become smarter. Hence, the interest in development of software for SoS has grown up, raising a new area of investigation termed Software Engineering for SoS (SESoS) [4, 6, 16]. Among these techniques, patterns are examples of popular software engineering techniques that can bring productivity and quality for SESoS. They correspond to standard solutions for recurrent problems that emerge in a domain [8]. As the reuse of a well-succeeded experience can foster the construction of the right product, patterns can reduce costs and time, besides contributing to the trustworthiness expected from a SoS. And, as simulations become increasingly dominant and broadly used by various industries, it becomes paramount to establish techniques and methods for the effectiveness of developers [15].

Simulations are a recognized approach to deal with SoS dynamicity [14]. Systems engineering already exploits simulations in SoS Engineering (SoSE). However, there is still a lack of works that explore techniques and software engineering methods in the context of SoSE. Furthermore, SoSE is a young discipline, and general principles and patterns remain to be discovered [7]. DEVS (Discrete Event System Specification) is well recognized formalism to simulate SoS. In DEVS, constituents operations are specified via a labeled state machine, i.e., a state machine in which transitions occur due to data input, output events, or time elapsed. There exist many DEVS variants, including finite probabilistic (FP-DEVS), non-deterministic, and finite deterministic. As non-determinism is unfeasible, deterministic DEVS versions are more common on platforms, such as FD-DEVS (Finite Deterministic DEVS), implemented in platforms as MS4ME[2].

DEVS models are structured over atomic and coupled models. An atomic model represent a single constituent system. Such model exhibits the following elements [2]: (i) a state machine that performs transitions due to input or output events; (ii) variable initialization; and (iii) the definition of abstract data types, global variables, ports, and events. In turn, coupled models are composed by atomic models and couplings connecting them, representing the entire SoS structure. In DEVS, a constituent system is driven by a state machine. Such a state machine is specified based on a well-defined set of primitives. A state in a DEVS state machine can either be a 'hold state' or a 'passive state' (exclusively). A hold state is one in which the execution flow will remain in for a certain amount of time until automatically changing to another state (via an internal transition). A passive state is one that the model will indefinitely remain in (or until it receives a message that triggers an external transition). These are the basic constructs for a state machine in DEVS:

Passivate State (PS). A state in which the execution flow will remain until an input event causes a transition to another state.

```
passivate in STATENAME!
```

[2]http://www.ms4systems.com/pages/main.php

Hold State (HS). A state in which the execution flow will be stopped for a well-defined time, such as 5 time units.

```
hold in STATENAME for time 5!
```

Initial State (IS). The initial state of a DEVS state machine. It can be `hold` or `passivate`.

```
to start passivate in STATENAME!
or
to start hold in STATENAME for time 5!
```

Internal Transition (IT). This transition enables the execution flow to spontaneously go to another state after a specified amount of time. Every hold state must have one and only one internal transition.

```
from FROMSTATE go to TOSTATE!
```

Output Transition (OT). A transition that causes the output of a value. Any state that has an internal transition can also have one output message. Such message is delivered before that internal transition occurs. It is important to remark that every OT requires an IT.

```
after STATENAME output OUTPUTMESSAGE!
```

External Transition (ET). An external transition defines an input message that the constituent might receive when in a given state. This input triggers a state transition. The specification involved the current state, the expected input, and the state to which the model should transition in reaction to that input message. Any state can have one or more external transitions defined. The syntax for this is:

```
when in FROMSTATE and receive INPUTMESSAGE go to TOSTATE!
```

The code of a state machine in DEVS consists of an arrangement of such statements to guide the operation of a constituent. Essentially, the code is based on inputs and outputs. However, System Engineering guides and even DEVS textbooks usually do not teach how to group these statements conveniently to form functional input transitions and output transitions. Furthermore, some of these statements are conflicting (for example, if you specify a state as hold and passivate at the same time, or if you specify a hold state and you forget to specify the following transition). These conflicts can lead to a fail, error, or stop of the simulation. Next section details such conflicts and the patterns that emerge from it.

3 PATTERNS FOR SIMULATION OF SOS SOFTWARE ARCHITECTURES

DEVS represents the state of the art for simulation of SoS [2]. To be more didactic, we base our approach on a DEVS dialect called DEVS Natural Language (DEVSNL), [2] that enables to program atomic and coupled models expressed as FD-DEVS in a human-like format using tools such as MS4ME. Table 1 details the potential conflicts that we identified:

- **HS-PS**: A hold state can not be a passive state at the same time, and vice versa. This would mean to specify a state that simultaneously (i) is expecting for a input to make a transition and (ii) spontaneously transit to another state, which is unfeasible;
- **IT-PS**: Internal transitions and passivate states are incompatible. This would mean to ask a state to indefinitely be the current one , at the same time, specify that the same

state must transit to another state (without specifying time or any expected input);

- **IT-ET**: A internal transition can not be a external transition at the same time, as this means that a transition would simultaneously cause a state transition due to an input and a spontaneous transition, at the same time;
- **OT-PS**: Technically, an output transition is an internal transition. Since a passive state can not have an internal transition it can not have an output transition. This would mean to specify that a state is indefinitely the current one, and that it must transit to another state, producing an output, without specifying the forthcoming state;
- **OT-ET**: This conflict means that a same state would wait for an input message and produce an output, what is unfeasible.

Statement/ Statement	PS	HS	IS	IT	OT	ET
HS	X					
IS						
IT	X					X
OT	X					X

Table 1: Conflicts and compatible instructions in DEVS.

As discussed, it is important to be aware of these conflicts to design robust SoS simulations. From this need, two patterns emerge: one to group instructions that represent input transitions and their sub-activities involved, and another that represents output transitions. We established our patterns relying on the classical Gamma's structure [8]: recurrent problem, context, and solution. The recurrent problem is the same for all situations: conception of functional simulations of constituent systems that can form a SoS and can be simulated in DEVS. Since DEVS is based on inputs and outputs, we established one pattern for input (Table 2) and another one for output (Table 3). The context is the same for both (DEVS simulation models), and recurrent problem changes only about the purpose: input or output. We present them as follows.

3.1 DEVS Input

DEVS Input pattern expresses that an input will cause a transition when, at some state, it receives a data. If the `passivate` comes after the input instruction `when`, it could cause a conflict with a `hold` of a following output instruction.

Name	DEVS Input
Recurrent Problem	Specifying a set of simulation instructions that characterizes an input event without conflicts with other instructions.

Solution	`passivate in <<fromState>>!`
	`when in <<fromState>> and receive <<`
	`dataReceived>> go to <<toState>>!`

Table 2: Patterns for Input and Output in DEVS Simulation Models.

DEVS Input (Figure 1) specifies a `PassivateRule` that passivates in one and only one state, whose name is represented by a label. From this state, a `InputTransition` occurs when a pre-determined type of data is received, causing the transition from one state to one and only one another state.

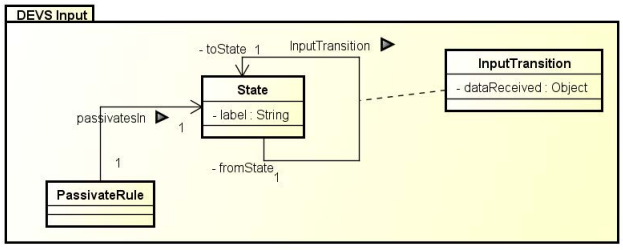

Figure 1: DEVS Input Pattern expressed as a class diagram in UML.

3.2 DEVS Output

DEVS Output pattern prescribes that, once an output occurs spontaneously (without any triggering event), it should (i) stay in that state of one second (this time can be specified according to convenience), (ii) perform the output, and (iii) transit to the next state. If the next state receives an input, it will be subject to a `passivate` instruction; otherwise, to a new `hold`.

Name	DEVS Output
Recurrent Problem	Specifying a set of simulation instructions that characterizes an output event without conflicts with other instructions.

Solution	`hold in <<fromState>> for time 1!`
	`after <<fromState>> output <<dataType>>!`
	`from <<fromState>> go to <<toState>>!`

Table 3: Patterns for Input and Output in DEVS Simulation Models.

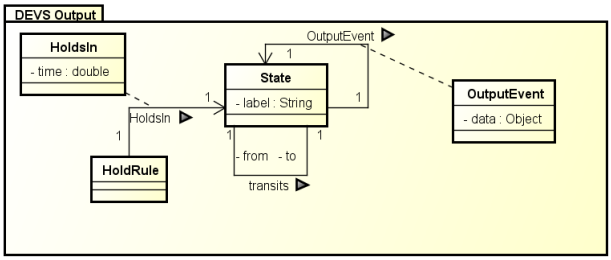

Figure 2: DEVS Output Pattern expressed as class diagram in UML.

DEVS Output (Figure 2) specifies a `HoldRule` that holds in one and only one state for a pre-determined amount of time. From that state, an output event occurs, delivering some data, and transiting from one state to another state.

4 EVALUATION

Aiming at gathering multiple sources of evidence, we carried out case studies, i.e., an exploratory type of empirical method for investigating a phenomena in its natural environment using data gathered from few entities (people, organizations, and sensors) [27]. We adopted SosADL, a novel architectural description language (ADL) conceived for the specification of architectural descriptions for software architecture of SoS [24]. With SosADL, it is not necessary to know all the constituents that could join the SoS at design-time, as it is possible to design abstract software architectures. We specified an abstract architecture, instantiated in a concrete architecture, and transformed via a model-based transformation written in Xtend[3], in a DEVS simulation model (written in DEVSNL, a DEVS dialect), supported by MS4ME[4] environment. We applied the patterns in two distinct domains: a ***Space SoS*** and a ***Flood Monitoring SoS***. We used the patterns in a model-based transformation that takes SoS architectural model in high-level of abstraction as input (documented in SosADL) and automatically produces functional simulation models as outcome.

Evaluation Approach. To evaluate our approach, we adopted GQM (Goal-Question-Metric) [33], an approach that creates an explicit link between measured data and the goals of measuring before data collection, avoiding misinterpretation of data [22]. The **goal** of this evaluation was *to determine whether our patterns support the correct generation of functional simulations of SoS software architectures.* We apply it in two different contexts, and describe both solutions. We established the following **research questions** to be applied for both domains:

RQ1. Do such patterns represent reusable solutions?
Rationale. Patterns are intended to be reusable solutions for recurrent problems. As such, it is important that, despite the necessary adaptations for different domains, the same solution can be applied for many contexts. This research question evaluates whether this happens.
M1. Effectiveness: given by the amount of functional atomic models and state machine lines of code effectively generated, representing how many times the same patterns were applied.

RQ2. Was the transformation successful?
Rationale. Since the simulation model is automatically generated, it is important to check the validity of the produced model. A transformation can be considered successful if the simulation runs without errors. As the simulation is functional, it means that the patterns are applicable repetitively and in a large scale.
M2. Simulation failures: given by the quantity of detected failures during model simulation, such as simulation crashing or stopping.

4.1 Case 1: Space SoS

A Space SoS is a SoS composed of constituents in ground and space to fulfill missions such as telecommunication, Global Positioning System (GPS), weather forecast, Earth and space observation, meteorology, resource monitoring, military observation, and many others, as illustrated in Figure 3. Space SoS can contain around 800 constituents [34]. This type of system is usually divided into three main segments: Space, which is the part placed in orbit (satellites, probes, space stations); Launcher, that is used to place the space instruments and constituents in orbit (rockets, space shuttles); and Ground, which supervises satellite operation. The Ground consists of mission control system, operation control system, ground stations and data communication networks [1, 31]. Each segment materializes one or more systems that have their own attributions. Each segment can be itself a different SoS that plays the role of constituent in the space SoS.

Figure 3: Illustration of the Brazilian Space SoS for Data Collection, adapted from [20].

Satellites are the main constituents of a space SoS. Each satellite is divided into several subsystems, such as onboard computer, power system, propulsion system, attitude control and communication system. Satellites are considered in two parts: payload and platform. The payload part assures that a system accomplishes the mission (e.g., sensors, cameras, infrared, in case of a forest monitoring, for example). The platform part is responsible for leaving the satellite in operation, such as solar panel, batteries, and reaction control system. The satellite only establishes contact when it is passing over the geographic location on which the ground station is positioned. Launching a satellite into space has a high cost. For instance, a CubeSat, i.e., an open source architecture with 10cm x 10cm x 10cm, for example, has cost estimated in $80,000 dollars to be launched into space. Due to the high costs and relevant potential losses, this domain is considered a critical one. Hence, it is important to anticipate SoS behaviors by means of simulation. In the space domain, some concepts are specially important: **Telecommand**, which consists of an operation sent in a remote way to satellites in order to perform some action, such as, capturing images or opening the solar panel (satellite uplink of missions); and **Telemetry**: technical name given to information received from the

[3] http://www.eclipse.org/xtend/
[4] http://www.ms4systems.com/pages/ms4me.php

status of the satellite. It happens during the passage of the satellites on the ground stations (satellite downlink).

SoS Characterization. This small-scale Space SoS was designed based on a real SoS currently in operation in Brazil. Such SoS is composed of the following different constituents, as follows.

(1) **Command and Control Center (C2)**: It is located in São José dos Campos, Brazil (Point B in Figure 3), is responsible for the generation of telecommand and telemetry packet;
(2) **Satellite**: It generates images of the planet in a regular interval of days using a Wide Field Imager (WFI) with frequency bands in the visible spectrum and Near Infrared (NIR);
(3) **Ground Station:** It is located in Cuiabá, Brazil (Point A in Figure 3), involves reception and satellite data transfer(telemetry and telecommand), it temporarily stores image data and satellites tracking;
(4) **Remote Sensing Data Center:** It accomplishes activities of receiving, recording, processing, storage and distribution of images and data from remote sensing.

Every mission in a Space SoS is performed according to the following meta-process, called *Meta-process for Payload missions in Space SoS*, as shown in Figure 4 and following the steps described below:

(1) Remote Sensing Data Center requests payload data for Command and Control Center (C2);
(2) C2 Center creates the operations(telecommand and telemetry) and schedules their execution;
(3) Ground Station configures antennas and rotors;
(4) Ground Station establishes link with Satellite;
(5) Ground station sends remote control;
(6) Satellite executes commands;
(7) Satellite stores payload data;
(8) Ground Station requests payload data;
(9) Satellite forwards telemetry data;
(10) Ground Station stores raw data;
(11) Remote Sensing Data Center searches for telemetry data;
(12) Remote Sensing Data Center tags and stores data
(13) Remote Sensing Data Center distributed payload data to Mission Center.

As this is a meta-process, there are meta-activities and meta-constituents that execute it. In this sense, the instantiation of a concrete process consists in the identification of the constituents and activities that replace these elements in the process, as suggested by Garcés and Nakagawa [26].

Case Design. For modeling a SoS software architecture of a Space SoS, we conducted requirements elicitation meetings with an expert from the Brazilian National Institute of Space Research. He aided us in the comprehension of the SoS structure, the main constituents, and how they interoperate to achieve the main results that are expected. We modeled a small scale SoS with only one **mission** to be achieved: *Amazon forest monitoring via images taken by the satellite*. Many satellites, ground stations, and data centers could be part of a same SoS. However, for this context, only one constituent of each type already exercises our patterns. The following mission

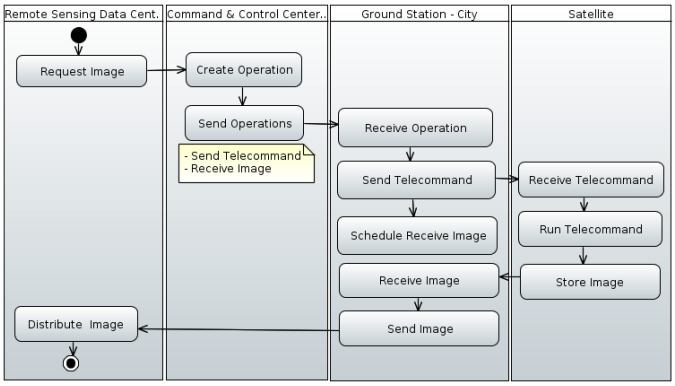

Figure 4: Activity Diagram illustrating the Meta-process for Payload missions in Space SoS.

was established:

Reporting. We specified one Space SoS architecture in SosADL with four constituents: one data center, one C2 center, one ground station, and one satellite. Besides that, four mediators were modeled to intermediate the constituents communication. For each one of these elements modeled in SosADL, one equivalent model was generated in DEVS using the patterns that we established. Moreover, a stimuli generator was automatically created as well to feed the simulation. The simulation run on an Intel core i5-3230M 2.60GHz (x64) processor, with 4 GB of RAM Memory, 1TB of HD, and running Ubuntu 16.04 with 64 bits.

RQ1. Do such patterns represent a reusable solution?
All of the generated code are driven by state machines specifications that were created using our approach. In total, 143 lines of code (LoC) were created to guide the behaviors of constituent systems, mediators, and stimuli generators. As DEVS Input pattern is expressed in two lines of code, whilst DEVS Output Pattern is expressed with three lines of code, there is an average of 2.5 lines per pattern. Hence, we can conclude that our patterns were used for automation purposes almost 60 times (M1 = 57.2) to produce constituents behaviors code as state machines for the Space SoS produced. For all of them, no conflicting instructions occurred and the systems run accordingly as predicted at design-time. Hence, we claim that the pattern is reusable for this context.

RQ2. Was the transformation successful?
The simulation run accordingly with no failures. Thus, we can consider that the transformation was feasible and well-succeeded for this particular context. Further applications should be tested. For now, M2 is equals to 0%.

4.2 Case 2: Urban Flood Monitoring SoS

We evaluated our approach in another scenario: a flood monitoring SoS (FMSoS) intended to be part of a smart city. Rivers cross the city and, when rains are intense, floods often occur, causing property loss, damage, and serious danger to the population. FMSoS notifies possible emergency situations to residents, businesses owners,

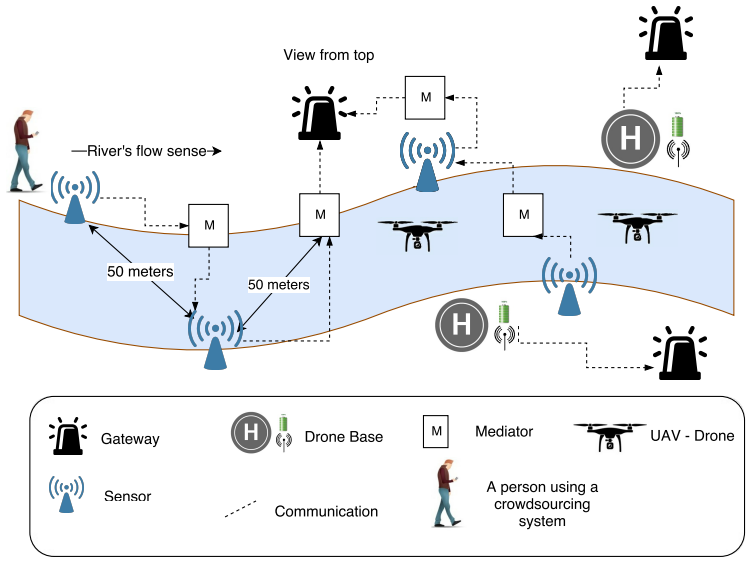

Figure 5: A Flood Monitoring System-of-System (FMSoS) as part of a Smart City System [11].

pedestrians, and drivers located near of the flooding area, and also to governmental entities and emergency systems. FMSoS is composed of five different types of constituents, as illustrated in Figure 5: **smart sensors**, which are fixed embedded systems monitoring flood occurrences in urban areas, located on river edges; **gateways**, which gather data from constituents and share them with other systems; **crowd-sourcing systems**, which are mobile applications used by citizens for real-time communication of water level rising; danger level is a pre-defined value (between 1 and 6, 1 being no risk, and 6 being flood effectively occurring) that can be classified by the human user according to what he/she observes; **drones**, which are UAVs also used to complement sensors observations by monitoring the river water level while they fly over it, sending pictures if some change in the water level occurs; and **drone bases**, which are fixed basis from where drones departure, and to where they come back for battery recharging, and data transmission.

Moreover, FMSoS is supposed to be part of a larger SoS composed of Wireless River Sensors, Telecommunication Gateways, Unmanned Aerial Vehicles (UAVs), Vehicular Ad Hoc Networks (VANETs), Meteorological Centers, Fire and Rescue Services, Hospital Centers, Police Departments, Short Message Service Centers and Social Networks, as described in [24]. Such SoS involves the National Center for Natural Disaster Monitoring, which monitors 1000 cities, with 4700 sensors, including 300 hydrological sensors, and 4400 rain gauges.

We specified one FMSoS architecture with 42 sensors, 9 crowdsourcing systems, and 18 drones, following the model shown in Figure 5. Each drone has its own base (18 drone bases), and transmits the information collected through a gateway that will be in the vicinity. 18 gateways are spread along the river boards. Mediators were produced as much as necessary to mediate these constituents. FMSoS is concerned with a single behavior: *flood alert*.

Data used. We chose a dataset collected by sensors [19] over four days, from November 23th 2015 to November 27th 2015. This interval was important because during these days a number of floods occurred. This enabled us to establish whether or not our simulation results in a diversity of situations. Data that arrive are chronologically ordered in gateway, and pairs of data are analyzed. If at least one pair has two measures equal or greater than 100 cm, a flood is confirmed.

Reporting. We used real data as input to our DEVS code to assess whether its behaviors correspond to those behaviors presented in the real FMSoS during such days. Data were stored in text files and automatically delivered to the simulation via stimuli generators that imitated the surrounding environment [10]. These stimuli generators delivered 1,000 samples for each sensor. Timestamps represented that each data sample was sent every five minutes for each sensor (i.e., 12 samples by hour, 288 per day, totalizing 3,47 days of data simulated), besides also delivering data for crowdsourcing systems, and drones.

Our approach correctly produced atomic models in DEVS for constituent systems specified in SosADL. Behaviors of constituent systems in SosADL were represented, as expected, as state machines in atomic models in DEVS. Listing 1 shows an excerpt of code generated for a mediator in Flood Monitoring SoS case. Mediator is an specialized type of system that composes a SoS responsible for receiving data forwarded by other systems and forward it again to a next system [32]. This type of system is essential for SoS operation, since it reinforces the data being transmitted if there is a long distance between two sensors that are collecting data to send to a gateway. Listing 1 shows the patterns applied in an operational mediator. In short, such behavior specifies a initial state (IS) and respective transition (Lines 1-2, IS-IT, as explained), that after receiving the coordinates of the constituents that it mediates (Lines 4-8, Input Pattern), it waits for receiving data from the sensors

(Lines 10-11, Input Pattern) and forward them towards a gateway (Lines 13-15, Output Pattern).

Listing 1: Specification of a behavior of an atomic model for a Mediator in DEVSNL.

```
1  to start hold in s0 for time 1!
2  from s0 go to s1!
3
4  passivate in s1!
5  when in s1 and receive Coordinate go to s2!
6
7  passivate in s2!
8  when in s2 and receive Coordinate go to s3!
9
10 passivate in s3!
11 when in s3 and receive Measure go to s4!
12
13 hold in s4 for time 1!
14 after s4 output Measure!
15 from s4 go to s5!
```

Table 4: Amount of lines of code produced using our patterns for Flood Monitoring SoS.

Model	#	LoC per model	LoC per model type
Sensor	43	24	1032
Gateway	20	18	360
Drone	18	29	522
Drone Basis	18	20	360
Crowd	9	22	198
Crowd Gateway	3	17	51
Transmitter	43	13	559
Drone Transmitter	18	15	270
Crowd Transmitter	9	15	135
TOTAL	181	173	3487

RQ1. Do the patterns represent a reusable solutions?
3487 LoC were produced, including the behaviors of constituent systems, mediators, and stimuli generator. For all of them, no conflicting instructions occurred and the systems run accordingly as predicted at design-time. Considering that the average amount of lines for each pattern is 2.5 (2 for one pattern and 3 for the another pattern), we conjecture that the patterns were applied around 1,400 times (M1 = 1,394.8). This shows how our patterns are effective, being reused several times in an automated solution, without creating any error. Table 4 summarizes these data.

RQ2. Was the transformation successful?
The transformation was successful while using the externalized patterns. Simulation run accordingly with no failures. Hence, for this context, M2 is equals to 0%.

Our approach correctly produced atomic models in DEVS for constituent systems specified in SosADL. Behaviors of constituent systems in SosADL were represented, as expected, as state machine sequence in the atomic model in DEVS. The simulation code generated allowed the SoS to transmit, coordinate, and measure data from sensors. Hence, we claim that the code generated from our approach is functional since it allows automatic generation and complete simulation of a SoS with no failures.

Threats to Validity. We mention the following threats: the scale of our evaluation, verification of correctness of the transformation rules, and bias. We mitigated the first by developing instances with distinct amounts of constituents. In all of these instances, the code generated worked with no failures. Moreover, our solution requires modest changes to scale, as the same model transformation can be applied to generate any number of simulation code, despite the possibility to model other constituents in SosADL. Regarding transformation correctness, we established correspondences between entities in both models and the resulting simulation model relieves the threat, showing a solution. We intend to conduct a further evaluation by specifying this transformation using a formal notation. This can enable the adoption of an automatic model checking, that can verify the correctness of the model transformation. There is a bias, as the same experts were responsible for specifying/implementing the SoS in SosADL and run the case study. We mitigated this threat by submitting the results to an external expert, who attested the feasibility of our results.

5 DISCUSSION

We applied such patterns in the development of software code for simulation of a small-scale Space SoS and a Flood Monitoring SoS for a urban area. The automatically generated simulation was applied into the context of validation of an emergent behavior of SoS [9]. These patterns are important because even during our first manual code specification in DEVS, we had problems with conflicting instructions. Moreover, DEVS books often do not provide this sort of discussion. Patterns supported us to automate the generation of these simulations, encapsulating the rationale behind the specification of a SoS software architecture.

Related work. Other proposals have explored patterns for simulation in DEVS, but under distinct perspectives [5, 17, 18, 21, 28]. Shulz et al. (2000) also present a mapping involving DEVS. They argue that the DEVS formalism is more expressive than StateCharts and present a mapping of the two system modeling formalisms to combine the benefits of formally well-defined models and a tool implementation, as we do. However, they do not externalize any pattern applied in such model transformation. Jéron et al. (2008) investigate the problem of predicting the occurrences of patterns in discrete-event systems [21]. They consider a pattern as a set of event sequences modeled by a finite-state automaton. They propose an off-line algorithm for automatic identification of patterns in DEVS simulations. However, they do not address patterns for conception of simulations. Hamri et al. (2010) present a specific catalog of design patterns for DEVS context. However, they do not provide details on how to group DEVS instructions to design constituents behavior, avoiding conflicts between them. Later, Hamri et al. (2013) present a work in progress in which they report behavioral design patterns to design and code DEVS behaviors in order to enhance the structure of the corresponding code and supply DEVS designers with software engineering techniques [17]. However, their patterns only express the state changes caused by the

occurrence of specific behaviors. They do not present any grouping of instructions as a set of patterns, as we do. Cetinkaya and Verbraeck (2011) established an approach to manage modeling and metamodeling in simulation engineering context [5]. They list a set of properties that model transformation rules should maintain to produce reliable simulations. However, they do not tackle SoS context, and they do not externalize patterns for the conception of a simulation. Finally, Petitdemange et al. established a solution based on patterns for reconfiguration of SoS software architectures [25]. Despite involving SoS domain and reconfiguration, they do not externalize patterns for simulation.

6 FINAL REMARKS

This paper presented two patterns for specifying non-conflicting input and output instructions of constituent systems in SoS DEVS simulations. We applied our patterns in two different domains to support automatic generation of SoS simulations from SoS architectural descriptions documented in SosADL. If a simulation is not reliable, the SoS produced using it will also not be. Hence, our contribution is important, as our patterns support the production of simulations with no conflicting instructions, making such simulation functional and reliable. As a consequence, we also increase the level of trustworthiness of the SoS produced based on such simulations. These patterns represent reusable solutions that can be applied for any SoS architecture design that adopts SosADL and DEVS for simulations. Indeed, the conceptual pattern described in UML can also be extended for any formalism that employs discrete events-based formalism and relies on labeled state machines to guide constituents behaviors. Future work include identification of other patterns, further empirical evaluations, and application of such patterns for other simulation formalisms.

ACKNOWLEDGMENTS

We thank São Paulo Research Foundation (FAPESP) (grants 2016/16870-2, 2017/17448-5, and 2017/06195-9); and Brazilian National Council for Scientific and Technological Development (CNPq) (grant 300394/2017-9).

REFERENCES

[1] 2008. *ECSS Space Engineering - Ground systems and operations.* Technical Report ECSS-E-ST-70C. European Cooperation for Space Standardization (ECSS).
[2] B. P. Zeigler et al. 2012. *Guide to Modeling and Simulation of Systems of Systems.* Springer.
[3] Jerry Banks. 1999. Introduction to Simulation *(WSC)*. ACM, Phoenix, USA, 7–13.
[4] Radu Calinescu and Marta Kwiatkowska. 2010. *Software Engineering Techniques for the Development of Systems of Systems.* Springer, Berlin, Heidelberg, 59–82.
[5] Deniz Cetinkaya and Alexander Verbraeck. 2011. Metamodeling and Model Transformations in Modeling and Simulation. In *Proceedings of the Winter Simulation Conference (WSC '11)*. Winter Simulation Conference, 3048–3058.
[6] Khalil Drira and Flávio Oquendo. 2015. Special Issue on Advanced Architectures for the Future Generation of Software-Intensive Systems. *Future Generation Comp. Syst.* 47 (2015), 60–61.
[7] John Fitzgerald, Simon Foster, Claire Ingram, Peter Gorm Larsen, and Jim Woodcock. 2013. *Model-based Engineering for Systems of Systems: the COMPASS Manifesto.* Technical Report Manifesto Version 1.0. COMPASS. http://www.compass-research.eu/Project/Publications/MBESoS.pdf
[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. Design patterns: elements of. (1994).
[9] Valdemar Vicente Graciano Neto. 2016. Validating Emergent Behaviors in Systems-of-Systems through Model Transformations. In *ACM Student Research Competition at MODELS*. Saint Malo, France, 1–6.
[10] Valdemar Vicente Graciano Neto, Carlos Eduardo Barros Paes, Lina Garcés, Milena Guessi, Wallace Manzano, Flavio Oquendo, and Elisa Yumi Nakagawa. 2017. Stimuli-SoS: a model-based approach to derive stimuli generators for simulations of systems-of-systems software architectures. *Journal of the Brazilian Computer Society* 23, 1 (13 Oct 2017), 1–22.
[11] Valdemar Vicente Graciano Neto, Lina Garcés, Milena Guessi, Carlos Paes, Wallace Manzano, Flavio Oquendo, and Elisa Nakagawa. 2018. ASAS: An Approach to Support Simulation of Smart Systems *(The Hawaii International Conference on System Sciences (HICSS-51))*. Hawaii's Big Island, USA, 1–10.
[12] Valdemar Vicente Graciano Neto, Milena Guessi, Lucas Bueno R. Oliveira, Flavio Oquendo, and Elisa Yumi Nakagawa. 2014. Investigating the Model-Driven Development for Systems-of-Systems *(ECSAW '14)*. ACM, Vienna, Austria, Article 22, 8 pages.
[13] Valdemar Vicente Graciano Neto, Flavio Oquendo, and Elisa Yumi Nakagawa. 2017. Smart Systems-of-Information Systems: Foundations and an Assessment Model for Research Development. In *GranDSI-BR - Grand Research Challenges in Information Systems in Brazil 2016-2026*, Clodis Boscarioli, Renata Araujo, and Rita Maciel (Eds.). Special Committee on Information Systems (CE-SI) - Brazilian Computer Society (SBC), Brazil, 13–24. ISBN: 978-85-7669-384-0.
[14] Valdemar Vicente Graciano Neto, Carlos Eduardo Barros Paes, Flavio Oquendo, and Elisa Yumi Nakagawa. 2016. Supporting Simulation of Systems-of-Systems Software Architectures by a Model-Driven Derivation of a Stimulus Generator *(WDES' 16)*. SBC, Maringá, Brazil, 1–10.
[15] Jeff Gray and Bernhard Rumpe. 2016. Models in simulation. *Software & Systems Modeling* 15, 3 (2016), 605–607.
[16] Milena Guessi, Valdemar Vicente Graciano Neto, Thiago Bianchi, Katia Romero Felizardo, Flávio Oquendo, and Elisa Yumi Nakagawa. 2015. A systematic literature review on the description of software architectures for systems of systems. In *SAC*. Salamanca, Spain, 1433–1440.
[17] Maamar Hamri, Rabah Messouci, and Claudia Frydman. 2013. Discrete Event Design Patterns.. In *1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, Montreal, Quebec, Canada, 349–354.
[18] Maamar ElAmine Hamri and Lassaad Baati. 2010. On Using Design Patterns for DEVS Modeling and Simulation Tools *(SpringSim '10)*. Society for Computer Simulation International, Orlando, Florida, Article 121, 9 pages.
[19] Flávio E.A. Horita, Joao Porto de Albuquerque, Lívia C. Degrossi, Eduardo M. Mendiondo, and Jó Ueyama. 2015. Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks. *Computers & Geosciences* 80 (2015), 84 – 94.
[20] INPE. 2017. Sistema Integrado de Dados Ambientais. http://sinda.crn2.inpe.br/PCD/SITE/novo/site/index.php. (2017). Accessed: 2017-12-30.
[21] Thierry Jéron, Hervé Marchand, Sahika Genc, and Stéphane Lafortune. Predictability of Sequence Patterns in Discrete Event Systems.
[22] Heiko Koziolek. 2008. Dependability Metrics. Springer-Verlag, Chapter Goal, Question, Metric, 39–42.
[23] Mark W. Maier. 1998. Architecting principles for systems-of-systems. *Systems Engineering* 1, 4 (1998), 267–284.
[24] Flávio Oquendo. 2016. Software Architecture Challenges and Emerging Research in Software-Intensive Systems-of-Systems. In *ECSA*. Copenhagen, Denmark, 3–21.
[25] Franck Petitdemange, Isabelle Borne, and Jérémy Buisson. 2016. Assisting the evolutionary development of SoS with reconfiguration patterns. In *ECSA Workshops*. Copenhagen, Denmark, 9.
[26] Lina Maria Garcés Rodriguez and Elisa Yumi Nakagawa. 2017. A process to establish, model and validate missions of systems-of-systems in reference architectures. In *SAC*. ACM, Marrakech, Morocco, 1765–1772.
[27] Per Runeson and Martin Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Softw. Engg.* 14, 2 (April 2009), 131–164.
[28] S. Schulz, T. C. Ewing, and J. W. Rozenblit. 2000. Discrete event system specification (DEVS) and StateMate StateCharts equivalence for embedded systems modeling. In *IEEE International Conference on Engineering of Computer Based Systems*. 308–316.
[29] SEBoK. 2016. Guide to the Systems Engineering Body of Knowledge, version 1.6. (2016). https://goo.gl/URBz7b [Online; accessed November 28th, 2017].
[30] Yentl Van Tendeloo and Hans Vangheluwe. 2017. An evaluation of DEVS simulation tools. *Simulation* 93, 2 (2017), 103–121.
[31] James R. Wertz and Wiley J. Larson. 1999. *Space mission analysis and design* (3rd illustrated edition ed.). Microcosm; Kluwer.
[32] G. Wiederhold. 1992. Mediators in the architecture of future information systems. *Computer* 25, 3 (March 1992), 38–49.
[33] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.
[34] Wilson Yamaguti, Valcir Orlando, and Sérgio de Paula Pereira. 2009. Sistema brasileiro de coleta de dados ambientais: status e planos futuros (Brazilian system of environmental data collection: status and future plans - in portuguese). In *Brazilian Symposium on Remote Sensing*, Vol. 14. 1633–1640.