Tracking Events as an Add-On Functionality of the Routed DEVS Formalism

Maria J. Blas[1,2], Mateo Toniolo[2], Silvio Gonnet[1,2]

[1]Instituto de Desarrollo y Diseño INGAR – CONICET & UTN
Avellaneda 3657 – Santa Fe – CP 3000 – Argentina

[2]Facultad Regional Santa Fe – Universidad Tecnológica Nacional (UTN)
Lavaisse 610 – Santa Fe – CP 3000 – Argentina

mariajuliablas@santafe-conicet.gov.ar, mtoniolo@frsf.utn.edu.ar,
sgonnet@santafe-conicet.gov.ar

Abstract. *The Routed DEVS (RDEVS) models improve traditional discrete-event models by enhancing the development of routing processes over predefined behaviors. This paper provides a novel solution for tracking events flowing in such routing processes as a new functionality of the RDEVS formalism. Such functionality is given by redesigning the original formalism following the "Decorator" pattern. An implementation of the redesign is developed as part of the RDEVS Java Library. As a result, we provide a solution that allows getting structured data from RDEVS models at execution time without changing their expected behavior or the simulator engine.*

1. Introduction

The Discrete-Event System Specification (DEVS) is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction [Zeigler et al. 2018]. A DEVS extension frequently suggests a direction in which Modeling and Simulation -M&S- (as a general field) and DEVS formalism (as a particular specification) can be enhanced. That is the case of Routed DEVS (RDEVS), an extension that promotes the M&S of routing processes over DEVS models through a new modeling level: routing behavior [Blas et al. 2022]. DEVS and RDEVS models can be combined to define multi-formalism models executed with any DEVS abstract simulator. The data collected from such execution will be DEVS-based. Hence, to monitor the main features of RDEVS models at execution time, a distinct approach is required.

In this paper, we show how a software engineering design pattern can be applied over the RDEVS formalism design to include event tracking into the models without changing the expected behavior of the formalism. We aim to create a flexible alternative to subclassing the original formalism with new functionality without altering what the formalism already does. Based on the Decorator pattern [Gamma et al. 1994], we preserve the routing functionality embedded in the models and the suitability of DEVS simulators as execution engines. Moreover, we allow models to collect event flow data dynamically. Given that when collecting data, it is better to do it using a structured format [Sagiroglu and Sinanc 2013], we propose a solution for collecting structured data from RDEVS models at execution time as a new type of functional (not behavioral) responsibility. Such a solution is implemented as a Java package attached to the RDEVS Library. The main contributions are *i)* from the M&S theory, the conceptualization of

https://doi.org/10.5753/mssis.2023.235691

add-on functionalities as part of DEVS-based formalisms using a design pattern; specifically, an add-on definition for RDEVS models to store the data collected during the simulation in a structured Object-Oriented form, and *ii)* from the M&S practical field, implementation of such a conceptualization as a Java package attached to the RDEVS Library that records data in JavaScript Object Notation (JSON).

The remainder of this paper is organized as follows. Section 2 introduces the foundations of RDEVS centered on how event tracking data is attached to simulation engines. Section 3 presents the tracking proposal as an add-on functionality to the RDEVS conceptualization. A discussion regarding the results is presented in Section 4. Finally, Section 5 is devoted to conclusions and future work.

2. Background

The RDEVS formalism reduces the modeling effort when routing processes are defined over DEVS models by introducing a new modeling level: the routing behavior [Blas et al. 2022]. In an RDEVS-based solution, the modeler employs three distinct models: *i)* the *essential model* defines the discrete-event behavior of a component used in a routing node, *ii)* the *routing model* defines a routing node by relating an essential model description with a routing policy through a precise behavioral definition of how event routing should be performed, and *iii)* the *network model* defines the routing process scenario by coupling a set of routing models using all-to-all connections (leaving the routing task to node policies). According to [Zeigler et al. 2018], RDEVS acts as a "layer" above DEVS providing routing functionality without requiring the user to "dip down" to DEVS itself for any functions. To do this, by using the models described above, RDEVS divides the behavioral modeling level of DEVS into two types: domain behavior and routing behavior. Hence, the flat behavior used in DEVS is replaced with a two-level structure, where the routing behavior is abstracted in the routing model specification using a routing policy as part of the acceptance/rejection process.

The RDEVS formalism is closed under coupling [Blas et al. 2022]. Zeigler (2018) considers that two questions arise for the closure under coupling of DEVS-based formalisms: *1)* are they subsets of DEVS, behaviorally equivalent to DEVS but more expressive or convenient, or bring new functionality to DEVS? and *2)* have simulators been provided for them to enable verification and implementation? The answers to these questions for RDEVS formalism are the following:

1. The embedding of the essential model in the routing model definition and the use of routing policies over the well-defined structure of the network model are the main features of RDEVS improving the DEVS formalism. By isolating the routing behavior from the domain-specific behavior, routing processes are built with a strict separation of concerns.
2. Even when some DEVS-based extensions have required new simulators to improve the execution of the proposed models, for RDEVS models, any simulator implementing the DEVS abstract simulator can be used (that is due to the equivalence shown by closure under coupling).

Even when RDEVS models can be run with DEVS simulators, the data obtained from such a process will be DEVS-based. Such data is acceptable to analyze simulation results related to DEVS basic behaviors. However, when monitoring RDEVS models, it is essential to be able to track the flow of events between different models, always considering their routing policies. This data cannot be obtained from a DEVS simulator.

RDEVS models are allowed to send (receive) output (input) events selectively. Depending on the case, these models must: *i)* add routing information to all events produced (i.e., output events) combining the routing policy and output function, and *ii)* decide whether to accept/reject each event received (i.e., input event) using the event routing data, the current state of the model, and the routing policy attached to the node. Hence, when studying the dynamic of RDEVS models, the following questions arise: How many events were accepted/rejected in a routing node? How many events were sent? What types of events were accepted/rejected by a particular node? Under which state conditions? How many times have models produced output events accepted/rejected by all the destination models? Getting data to answer these questions becomes an issue to be solved as part of the RDEVS formalism.

There are two high-level solutions to this problem: *i)* improve the design of RDEVS formalism through a redesign process allowing to collect RDEVS-based data directly from models, and *ii)* redefine the DEVS abstract simulator to manage new types of components for collecting both DEVS-based and RDEVS-based data. As evident, each solution is closely related to *1)* and *2)*, respectively. To maintain the suitability of DEVS simulators as support of RDEVS models (i.e., to maintain a multi-formalism simulation approach), in Section 3, we introduce a suitable conceptual modeling-based solution for getting (structured) data from RDEVS simulations using event trackers (i.e., the solution *i)*). Such a modeling-based solution is centered on the Decorator pattern as a means to add responsibilities to the well-defined conceptualization of RDEVS simulation models defined in [Blas et al. 2022] without altering its behavior.

At this point, two remarks are critical for understanding our proposal. First, we do not want to introduce new functionality to the RDEVS formalism. By restructuring the foundational design, we want to show that it is possible to "decorate" formal models without changing their external behavior. The conceptual model (Section 3.1) is used to rebuild the existing RDEVS Java library (Section 3.2). Second, we do not want to track the simulation itself. We want to capture the event flow data produced during the simulation in response to the accept/reject actions taken by the routing behavior of nodes (i.e., routing models). As previously detailed, instead of handling regular DEVS events, RDEVS models use events surrounded by routing data. Then, at simulation time, a model can receive an event that will not be processed by its domain behavior because it has been rejected by its routing behavior (placed at a higher decision level as explained in Appendix A). Indeed, we want to track events processed by the domain behavior of both sides (source and destination) because of routing policies.

3. RDEVS with the Tracker Add-On Functionality

3.1. Redesign at Conceptual Modeling Level

The Decorator design pattern is one of the twenty-three well-known design patterns proposed by Gamma et al. (1994). Such a structural pattern defines a flexible approach to enclosing a component in another object that adds a "border" with the intent of attaching an additional responsibility dynamically. The enclosed object is called a "decorator" that, following the interface of the component, decorates such a component so that its presence is transparent to the component's client.

From the UML class diagram described as a metamodel of the RDEVS specification (proposed in [Blas et el. 2021]), Figure 1(a) shows how the pattern has

been applied to support event trackers. Due to space reasons, the formal definition of the conceptualization in the mathematical form of RDEVS models is omitted. On the other hand, Figure 1(b) shows the conceptualization used for trackers to maintain data related to the attached RDEVS model (i.e., the *RDEVS Component*).

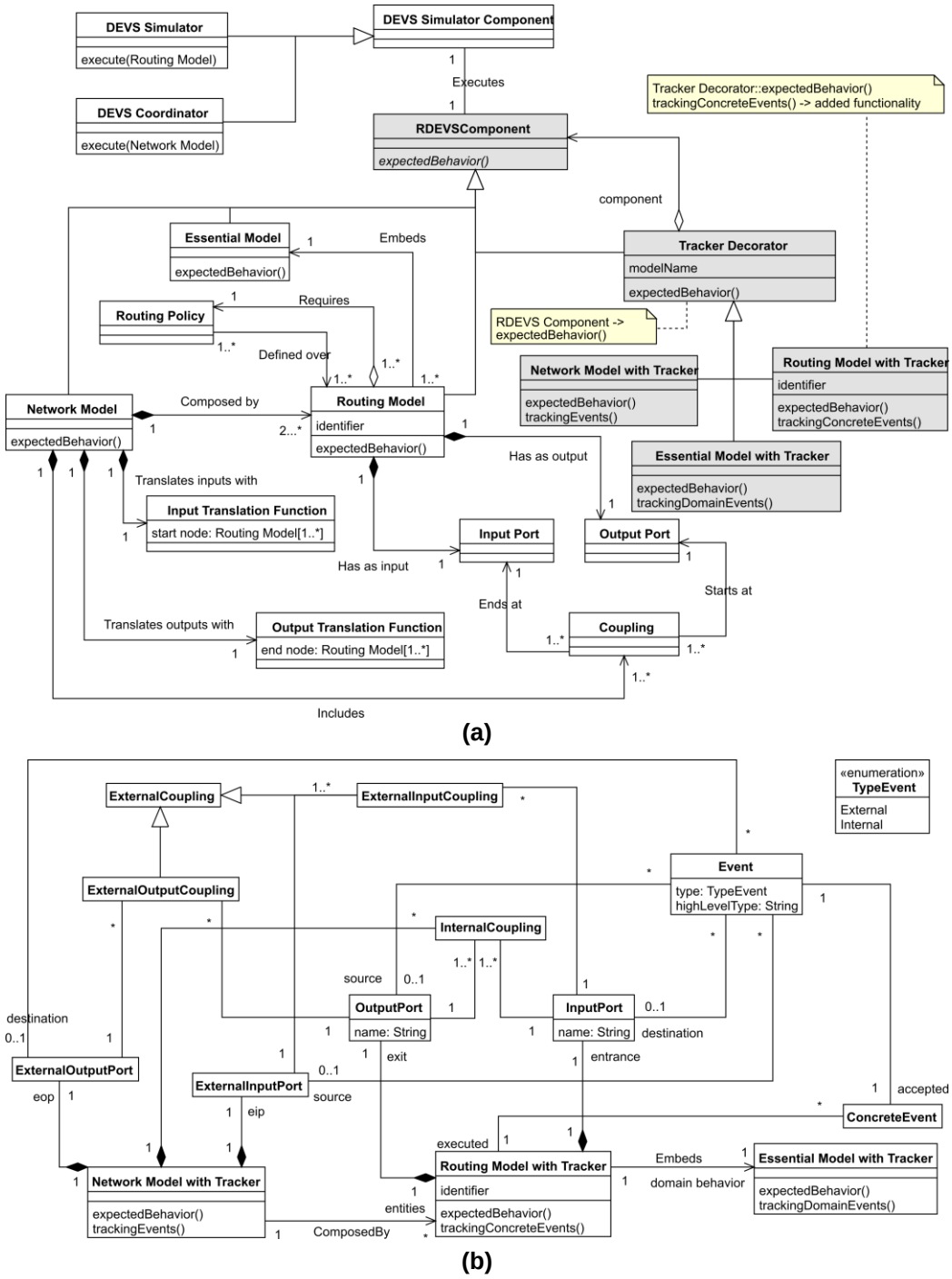

Figure 1. (a) UML class diagram of the RDEVS metamodel taken from [Blas et al. 2021] updated with the event trackers. New classes are the ones highlighted in gray. (b) UML class diagram of the conceptual model used to structure the trackers definition.

Figure 1(a) shows the structure of RDEVS models, how these models are related to *DEVS Simulator Components*, and how *Tracker Decorators* were added to the

conceptualization. When DEVS abstract simulator is used to execute RDEVS models *i)* routing models are executed by a *DEVS Simulator*, and *ii)* network models are performed by a *DEVS Coordinator*. That is defined through the *execute* operation placed at each type of *DEVS Simulator Component.* The *RDEVS Component* is introduced to abstract RDEVS models and the *Tracker Decorator* concept. The *expectedBehavior* operation is used to define how models should work. Then, the Tracker Decorator defines its *expectedBehavior* as the one described in the RDEVS model attached (i.e., the *RDEVS Component* aggregated). Each specific type of tracker includes its own *tracking* operation (see Section 3.2).

In Figure 1(b), compositions are used to relate ports with trackers. A *Routing Model with Tracker* is composed of an *Input Port* (identified as *entrance*) and an *Output Port* (identified as *exit*). A *Network Model with Tracker* is composed of an *External Input Port* and an *External Output Port*. Couplings between ports are also defined as *Internal Coupling*, *External Coupling*, *External Input Coupling,* and *External Output Coupling* to describe the model structure. Events are represented in the *Event* concept. Each *Event* is defined by a *type* and a *high-level type*. The *type* refers to how it is distinguished in the network model. An event is set as *External* when it is received/sent by the network. Instead, an event is set as *Internal* when it is exchanged between routing models composing the network model. On the other hand, the *high-level type* refers to the content of the event. Such content is defined by the modeler in the RDEVS simulation model that produces the *Event*. Each port registers the events that have been sent/received. In this way, for each event, the conceptual model describes which is the routing model that sends/receives an event through its output/input port. For example: *i)* an *Event* can be sent by an *Output Port* (*source*) to an *Input Port* (*destination*), *ii)* an *Event* can be sent by an *External Input Port* (defined as *source*) to an *Input Port* (*destination*), or iii) an *Event* can be sent by an *Output Port* (*source*) to an *External Output Port* (*destination*). Then, *i)* represents an *Event* with *type = Internal*, while *ii)* and *iii)* refer to an *Event* with *type = External* (an external input event in *ii)* and an external output event in *iii)*). The *Concrete Event* is used to denote that an *Event* has been accepted in a model due to the routing policy.

3.2. Implementation in the RDEVS Java Library

To update the existing implementation of RDEVS models with event trackers, we perform a refactoring of the RDEVS Java Library [Espertino et al. 2022] following the conceptualizations previously described. Such models were implemented as a new Java project related to the existing implementation.

At the beginning of each simulation, the trackers attached to the models are automatically created through the initialization process. During this step, the data related to the model structure (i.e., static data) is collected in each tracker (e.g., for the routing model tracker: *identifier*, *name*, *input port names*, *output port names*, and so on). On the other hand, the data related to the simulation execution (i.e., dynamic data) is obtained during the simulation process (once the models are initialized). Such data is produced by the identified events exchanged among routing models. For each exchange, the following data is collected in an *Event*: *highLevelType*, *type* (*Internal* or *External*), *source* (output port from which it has been sent), and *destination* (input ports to which it was intended). Then, a list of *Events* is dynamically built in the trackers representing each port from which identified events depart (*Output Port* for an *Event* with *type* =

Internal or *External Input Port* for an *Event* with *type = External*). The same strategy is used on the trackers representing ports on which events are attempted to be sent.

When a routing model accepts an event (i.e., the routing policy allows the model to execute its domain behavior), a *Concrete Event* is created. Such a *Concrete Event* is attached to the original *Event* produced by the sender through the *accepted* association. Moreover, it is attached to the *Routing Model* tracker related to the model that accepted it through the *executed* association. In this way, an *Event* collects all instances accepted by destinations at the *accepted* association (as a list of *Concrete Event* elements) and all intended targets at the *destination* association (as the list of ports that contain the *Event*).

Once the simulation ends, all the dynamic data related to event exchange is available as part of the tracker model's structure. Hence, to store such data, a JSON file is created. JSON is a lightweight data-interchange format defined as plain text written in JavaScript object notation. To get the JSON file, we add the tag "@Expose" to a predefined navigation among the Java classes implemented. This navigation is designed to store the minimum set of data required to rebuild the model.

Having data formatted in JSON allows us to use other software tools to study RDEVS models. For example, we can now design specific visualizations to improve the understanding of the routing process described in the RDEVS simulation. Vernon-Bido et al. (2015) identify four types of visualization for M&S: *1)* concept and diagram visualization, *2)* quantitative visualization, *3)* seek and find visualization, and *4)* pattern and flow visualization. We are interested in *quantitative visualization* (i.e., the static and semi-static graphs and time-series plots associated with M&S results and statistics).

3.3. Tracking a Routing Process: An Example

To show how the data collection process works, we propose a three-node example. Let's assume that a routing process is defined using three nodes: SELECTION, REPAIR, and PACKAGING[1]. Such nodes are connected to process distinct types of containers (A, B, or C). If the process of SELECTION succeeds, the container selected goes to PACKAGING. Otherwise, the container goes to REPAIR. After repair, the container goes directly to PACKAGING. Routing policies are defined as follows: *i)* SELECTION can accept all types of containers, *ii)* REPAIR can only fix containers of type C, and *iii)* PACKAGING can only process containers of types B and C. Hence, as evident, some events will not be processed in the nodes due to their routing policies. That is how RDEVS models will work during simulation.

Following this structure, Figure 2 shows the part of the JSON file obtained as output data of the routing model attached to the REPAIR node when performing a simulation. As expected, Figure 2 shows a set of events related to the *entrance* in line 207 (i.e., the *Input Port* attached to the *Routing Model Tracker* with *id = 2* and *name = "REPAIR"*). Since the REPAIR node only receives containers after the SELECTION node has processed them, all events received in the routing model are *Internal* (lines 210, 215, 220, 225, 230, and 235). Distinct types of containers are received (*highLevelType*). However, only the ones accepted by the routing policy (i.e., the ones with *highLevelType = "C"*) are marked as *concreteAccepted = true* (lines 217 and 232). That means these events are the only ones processed by the domain behavior of the

[1] The example is part of the scenario proposed by Toniolo (2021).

routing model. The output events produced by the routing model during the simulation are captured at the *exit* property (i.e., the *OutputPort* attached to the model). As the highlighted box shows, all these events are set as *highLevelType = "C"* (lines 427 and 432) and *concreteAccepted = true* (lines 428 and 433). The former is due to the REPAIR acceptance policy. Only containers of type C are processed before passing to the PACKAGING node. The latter is due to the PACKAGING node always can accept containers of type C.

Figure 2. Part of the structured data available in the JSON file for the routing model with id = 2 and name = REPAIR (i.e., the routing model attached to the REPAIR node).

As an example of quantitative visualization of the collected data, a Sankey diagram (i.e., a flow diagram that depicts nodes linked by flows) is presented in Figure 3. The quantity of each flow is represented as its width. This diagram is best used to show multiple paths through a set of stages. It helps to locate dominant contributions to an overall flow. For RDEVS models, we represent routing models as nodes and the number of events exchanged following routing policies as the width of the flows. We also include event information (e.g., *type* and *highLevelType*). Hence, the diagram helps to understand how RDEVS models have accepted/rejected events.

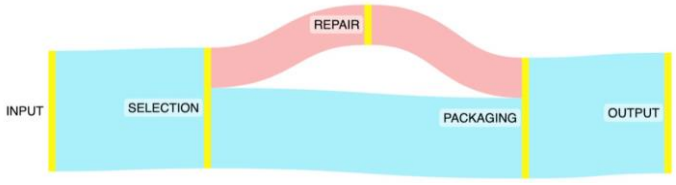

Figure 3. A Sankey diagram. Flow color is used to denote different paths. INPUT and OUTPUT are nodes denoting external input and output flows from/to the network model to/from routing models.

The diagram depicted in Figure 3 was built using the data recovered from the simulation as input for our graphical application. Such an application generates the diagram using the JSON file and produces an HTML file that contains the Sankey. The diagram is directly visualized in the browser to allow a more accurate visualization. For space reasons, details regarding the application development are not presented here.

Our intention in presenting this example is to show how trackers work starting from the conceptualization defined. Any case study where RDEVS models can be used (so far, RDEVS was used to simulate network protocols, software architectures, and electric power systems) will be enhanced with this additional feature since it allows to characterize event flows without extra effort.

4. Discussion

Collecting data is the pivotal step in the data processing of a model. For discrete-event models, the data is related mainly to events. Over the years, several researchers proposed trackers attached to DEVS models. A great example is DEVS Suite [Kim et al. 2009]. It was presented in 2009 as "a new generation of the DEVS Tracking Environment". In DEVS Tracking Environment [Sarjoughian and Singh 2004], a basic tracking environment was proposed allowing the execution of simulation experiments. In DEVS Suite, the data generated by the simulation models is collected dynamically and displayed as time-based trajectories.

From a different perspective, more recently, the authors Dahmani et al. (2020) have proposed a vocabulary of DEVS defined through an XML schema and an XML abstract simulator. The simulation is executed with XSLT transformations that generate an XML simulation tree at each event occurrence. In this case, XML is used as a meta-language that provides a standard for information exchange to encourage sharing models from different DEVS implementations. Given that an XML schema describes the structure of an XML document, documents produced following this approach can be considered structured data.

Our solution improves the understanding of the RDEVS formalism as an alternative conceptualization of DEVS models executed over DEVS simulators. As in [Dahmani et al. 2020], we get structured data stored in a well-known format that facilitates further analysis. Moreover, in all cases, tracing mechanisms are hidden from the modeler. In our case, it is also hidden from the simulator (maintaining the separation of concerns defined in the M&S Framework proposed with DEVS). To make the simulator aware of the trackers we should let it know detailed information regarding the decisions taken by models at both routing and behavioral levels. Likely, we should also modify the message exchange protocol of DEVS simulators introducing new messages to capture such behaviors.

By deploying trackers as part of the RDEVS library (supported by DEVSJAVA [Sarjoughian and Zeigler 1998]), DEVS Suite can be used to get DEVS-based results. Global reporting using both types of results could be produced without much effort.

5. Conclusions and Future Work

We have designed and implemented a conceptual modeling solution to the problem of tracking the event flow in RDEVS models. Our alternative solution provides an accurate separation of concerns that allows maintaining the advantages of using DEVS

simulators for executing RDEVS models while collecting structured data regarding how routing policies (at a higher decision level) are allowing/blocking the domain processing at the lower operational level.

Besides allowing to collect RDEVS-based data with the tracking, we expect to improve the extensibility, maintainability, and readability of RDEVS models. The DEVS-based data gathered by the DEVS simulator can be followed with the RDEVS-based structured data obtained from our trackers to allow a complete analysis of the simulation models. Since our data is structured, the processing to get information is less complex. Moreover, such processing can be developed using a general-purpose programming language or some specific software tool with JSON processing functionality. We are now working on the development of a web application that will allow practitioners to upload JSON files and get charts and tables with the processed data (i.e., information). That is the final goal of our research project at this stage. We already have developed an application for obtaining Sankey diagrams directly from the JSON file obtained from trackers to analyze the event flow among routing models.

Tracing simulation experiments is usually computation and data intensive. Still, it is noticeable that the solution presented is described as an add-on responsibility of the RDEVS models themselves. Performance analyses are not applicable at this stage.

We strongly believe that the solution presented to the problem of gathering structured data from RDEVS models might apply to other DEVS extensions. Moreover, the properties enjoyed by the solution are valuable from a software engineering point of view to be used to incorporate other functionalities to the simulation models as add-on responsibilities. In the future, we are planning to include more add-on functionalities in our proposal. Other design patterns will be explored.

Appendix A. How are Events Routed in RDEVS Models?

Let N be a RDEVS network model representing a system over which a routing process should be solved. When a value x (with $x \in X$) arrives at N, the input translation function T_{in} is executed to get an identified event x'. With the extra routing information added (i.e., destination models), the event x' is sent to all R_d routing models composing N. Each R_d model determines how to handle x' following its routing policy.

When a routing model M (that embeds an essential model E) receives an input event x', it executes the external transition function $\delta_{ext,M}$. Such a function has two distinct behaviors as follows. If the event should be accepted due to the routing policy, the model evolves to the next state by executing $\delta_{ext,E}$. Otherwise, the model remains in the current state (i.e., the event x' is ignored). If no external event occurs during a state s, an internal transition will take place when the time (in s) expires. Such a transition is defined by $\delta_{int,M}$ and produces a state change in the model. Before changing the state, the output function λ_M is executed to produce the identified output event y'. To get y', the model combines the result of λ_E with its routing policy. Once the output event is released, the state changes following the internal transition function based on $\delta_{int,E}$ (to perform the domain behavior).

If at any time, an event y' has no internal destination in N, y' should be sent outside N. Then, the network model executes the output translation function T_{out} to get an event y (with $y \in Y$) from the identified event y'. Such a function removes the routing information attached to y' propagating outside the related value as y.

References

Blas, M. J. and Gonnet, S. (2021). Computer-aided design for building multipurpose routing processes in discrete event simulation models. *Engineering Science and Technology, an International Journal*, 24(1):22–34.

Blas, M. J., Leone, H., and Gonnet, S. (2022). DEVS-based formalism for the modeling of routing processes. *Software and Systems Modeling*, 21(3):1179–1208.

Dahmani, Y., Ali, H. and Boubekeur, A. (2020). XML-based DEVS modelling and simulation tracking. *Inter. Journal of Simulation and Proc. Modelling*, 15:155-169.

Espertino, C., Blas, M., and Gonnet, S. (2022). Developing RDEVS Simulation Models from Textual Specifications. In *Anais do IV Workshop em Modelagem e Simulação de Sistemas Intensivos em Software,* pages 41–50, Uberlândia, MG, Brasil. SBC.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Kim, S., H. S. Sarjoughian, and Elamvazhuthi, V. (2009). DEVS-Suite: A simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 1-7, Times Square, New York City. ACM.

Sagiroglu, S., and Sinanc, D. (2013). Big data: A review. In *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems*, pages 42-47, Piscataway, New Jersey. IEEE.

Sarjoughian, H. S. and Zeigler, B. P. (1998). DEVSJAVA: Basis for a DEVS-based Collaborative M&S Environment. *Simulation Series*, 30:29-36.

Sarjoughian, H. S. and Singh, R. (2004). Building simulation modeling environments using systems theory and software architecture principles. In *Proceedings of the 2004 Advanced Simulation Technology Conference*, pages 99-104.

Toniolo, M. (2021). Desarrollo de una herramienta de software basada en Java para la captura de eventos en la simulación de modelos RDEVS. In *Actas de las Jornadas Argentinas de Informática e Investigación Operativa* 2021, pages 32-41, Buenos Aires, Argentina. SADIO.

Vernon-Bido, D., Collins, A., and Sokolowski, J. (2015). Effective visualization in modeling & simulation. In *Proceedings of the 2015 Spring Simulation Multiconference*, pages 33-40, Times Square, New York City. ACM.

Zeigler, B. P. (2018). Closure Under Coupling: Concept, Proofs, DEVS Recent Examples. In *Proceedings of the 2018 ACM International Conference of Computing for Engineering and Sciences*, pages 1-6, Times Square, New York City. ACM.

Zeigler, B. P., Muzy, A., and Kofman, E. (2018). *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press.