Journal of the Brazilian Computer Society, 2025, 31:1, doi: 10.5753/jbcs.2025.4613

Enhancing Routed DEVS Models with Event Tracking

Maria J. Blas [**INGAR CONICET & UTN** | ***mariajuliablas@santafe-conicet.gov.ar***]
Mateo Toniolo [**Universidad Tecnológica Nacional - FRSF** | ***mtoniolo@frsf.utn.edu.ar***]
Silvio Gonnet [**INGAR CONICET & UTN** | ***sgonnet@santafe-conicet.gov.ar***]

Instituto de Desarrollo y Diseño INGAR, CONICET & Universidad Tecnológica Nacional, Avellaneda 3657, Santa Fe, 3000, Argentina.

Received: 13 June 2024 • **Accepted:** 06 May 2025 • **Published:** 17 June 2025

Abstract The Routed Discrete Event System Specification (RDEVS) is a modular and hierarchical Modeling and Simulation (M&S) formalism based on the Discrete Event System Specification (DEVS) formalism that provides a set of design models for dealing with routing problems over DEVS. At the formal level, RDEVS models (as DEVS models themselves) are defined mathematically. However, software implementations of both formalisms are based on an object-oriented paradigm. Furthermore, at the implementation design level, the RDEVS formalism is represented by a conceptual model that uses DEVS simulators as execution engines. Even when RDEVS models can be executed with DEVS simulators, the resulting data (obtained as execution outputs) remains DEVS-based, restricting the study of event flows between models influenced by routing policies. This paper shows how the RDEVS formalism design was enhanced to include event tracking in the models without altering their expected behavior during simulation. Such an improvement is based on adding new features to existing RDEVS components. These features are defined as trackers, which are responsible for getting structured data from events exchanged during RDEVS executions. The proposed solution employs the Decorator pattern as a software engineering option to achieve the required goal. It was deployed as a Java package attached to the RDEVS library, devoted to collecting structured event flow data using JavaScript Object Notation (JSON). The results highlight the modeling benefits of adding event tracking to the original capabilities of the RDEVS formalism. For the M&S community, the novel contribution is an advance in understanding how best modeling practices of software engineering can be used to enhance their software tools in general and the RDEVS formalism in particular.

Keywords: Routed Discrete Event System Specification, Conceptual modeling, Software design pattern, Event flow, Modeling and simulation formalism.

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), considered a general field, and DEVS formalism (as a particular specification) can be enhanced. This is the case with 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]. An illustrative example is given below.

Let's assume that a routing process is defined in a scenario based on three distinct machines: SELECTION, REPAIR, and PACKAGING (**Figure 1**). These machines are connected as shown by the arrows. If the process of SELECTION succeeds, the selected container goes to PACKAGING. Otherwise, the container goes to REPAIR. After repair, the container goes directly to PACKAGING. All the machines can process all types of containers (A, B, or C), but in this particular setup, the REPAIR machine is configured to fix only containers of type C. Both formalisms can be used to solve the scenario depicted in **Figure 1** on a discrete-event basis. However, when using DEVS models, a much more complex modeling structure arises to deal with the setup than when using RDEVS models.

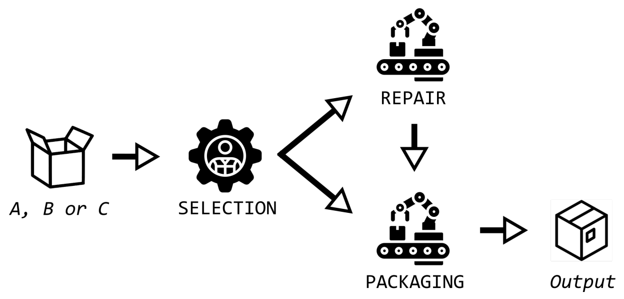


Figure 1. An illustrative scenario that can be solved with DEVS or RDEVS.

The processing of REPAIR is independent of the given setup. To handle this in DEVS, you should include an additional model to filter the processing of containers A and B by REPAIR. To define the same setup in RDEVS, you should modify the routing policy to reject the acceptance of container types A and B without adding any other model to the simulation. Hence, since well-defined M&S structures have a positive impact on the building of simulation models (because they can be applied systematically), the RDEVS formalism is conceptually defined as a subclass of DEVS. Such a subclass manages identified events within a model network where each node combines a behavioral description with a routing policy to adopt a routing behavior. The modeling strategy used when defining RDEVS models mathematically enhances the regular behavior of DEVS with new sets and functions used to handle the routing of events. Still, this modeling strategy does not affect the simulator execution (since it is defined as part of the model).

From the simulator's point of view, due to the shared discrete-event basis, DEVS and RDEVS models can be combined to define multi-formalism models executed with any DEVS abstract simulator. This is because RDEVS models are formalized as subclasses of DEVS models with particular modeling goals. However, since the execution is controlled following DEVS principles (i.e., a DEVS abstract simulator), the data collected from such a type of execution is DEVS-based. That means it is useful to study basic DEVS behaviors. As RDEVS is based on DEVS, DEVS-based data is a sound start to understanding simulation results. However, another feature to be studied when monitoring RDEVS simulation models is to analyze how the event flow works due to routing policies. For example, knowing how many containers were not accepted in the REPAIR machine because they were not of type C.

RDEVS models have DEVS traditional capabilities to send (receive) output (input) events. However, during such an exchange, RDEVS routing models have two extra responsibilities: i) they should add routing information (following their routing policy) to all outputs produced by the output function (to encapsulate the output values with routing data before producing output events), and ii) they should decide whether to accept or reject each input event received using the event routing data, the current state of the model, and the routing policy attached to the node. In this way, unlike regular DEVS events, RDEVS models use events embedded with routing data. Such data is essential for understanding several features of RDEVS dynamics, e.g., the number of accepted/rejected events in a routing node, the types of events accepted/rejected by specific nodes under certain state conditions, and the frequency of produced output events accepted/rejected by destination models. Hence, to monitor the main features of RDEVS models at execution time without losing the capability of running multi-formalism models, a distinct approach is required. Indeed, one of the following approaches can be used *i)* improving the design of RDEVS formalism through a reengineering process allowing the collection of RDEVS-based data directly from models without changing the formalism specification, or *ii)* redefining the DEVS abstract simulator to manage two new types of simulator components (i.e., RDEVS simulator and RDEVS coordinator) that allow for the collection of both DEVS-based and RDEVS-based data. This paper is an extension of [Blas *et al.*, 2023b] and presents a solution to *i)*. The latter is discussed in Section 2.2.

In this paper, we aim to create a flexible alternative to subclassing the original formalism, designing new functionality without altering what the formalism already does. Our work is devoted to developing a tracking feature for RDEVS executions based on the events exchanged at simulation time due to routing policy decisions. Both DEVS and RDEVS are defined mathematically, but the simulator used to execute the simulation of such models is defined as an algorithm implemented in a software module. Since DEVS is more likely to be computationally represented with an Object-Oriented (OO) paradigm [Zeigler *et al.*, 2018], we decided to follow the insights of DEVS and use the conceptual model of RDEVS as the representation to be redesigned with an add-on functionality. Then, our challenge is to adapt the conceptualization of RDEVS from a practical point of view that can be translated into existing software tools. This means we need to enhance the conceptual model of RDEVS with tracking functionalities without changing the expected behavior at simulation time.

There are many options in Software Engineering for including new functionalities in an existing software design. However, not all of them support the addition of new functionalities to existing elements at runtime without changing their core structure (in our case, the core structure is used to execute the simulation). In particular, we use a Software Engineering design pattern that promotes the dynamic composition of behaviors, enabling the creation of modular, reusable components that can be easily combined to enhance functionality. This paper shows how the Decorator pattern [Gamma *et al.*, 1995] was applied over the RDEVS design to include event tracking in the models without changing the expected behavior during simulation. We preserve the routing functionality as part of the models and the suitability of DEVS simulators as execution engines, allowing 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 introduce 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

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

As a remark, our proposal is not focused on tracking simulation experiments as a distinct activity. Tracking is not about gathering information on how state variables evolve during the simulation in a structured model but rather focuses on how to capture the events managed by RDEVS models during the simulation execution. While the former is usually computationally expensive and data-intensive, the latter does not require any additional computation or time-consuming steps beyond the simulation itself.

The rest of this paper is organized as follows. Section 2 introduces the foundations of our work to provide a background for understanding how DEVS simulations are performed in software tools, why RDEVS models behave differently than DEVS models, and the importance of having event-tracking data as an outcome of simulation engines. It also introduces the Decorator design pattern, the existing RDEVS design and implementation, and the software tool used to execute the models. To overcome the main issues described when executing RDEVS models with DEVS simulators, Section 3 presents the tracking proposal as an add-on functionality to the RDEVS conceptualization by applying the Decorator pattern to the conceptual model design. Furthermore, it details how such a conceptual proposal was deployed as part of the RDEVS Library implemented in Java. Thus, this section shows how RDEVS simulation models incorporate tracking as part of their definition without changing the DEVS simulator. The results are presented in Section 4, while Section 5 includes a discussion regarding the main benefits of using RDEVS models with tracking add-on functionality instead of just using RDEVS simulation models. Finally, Section 6 is devoted to conclusions and future work.

2 Background: M&S of Routing Processes over DEVS

A formalism provides a set of conventions for specifying a class of objects in a precise, unambiguous, and paradigm-free manner. In particular, the DEVS formalism [Zeigler *et al.*, 2018] provides a set of conventions for specifying discrete-event simulation models based on a well-defined M&S framework [Zeigler and Nutaro, 2016]. Such a framework is composed of four entities (**Figure 2**):

- *Source system*: Environment to be modeled.
- *Model*: Set of instructions for generating data comparable to the data observable in the system.
- *Simulator*: Computational system that executes the instructions detailed in the model.
- *Experimental frame*: Conditions under which the source system is exercised.

Furthermore, these entities interact by two relationships:

- *Modeling*: Determines when a model is a valid representation of the source system within an experimental frame.
- *Simulation*: Specifies what constitutes a correct simulation of a model by a simulator.

For DEVS, the framework emphasizes the idea of *model* and *simulator* as two independent entities linked when a *model* is executed in a computational environment (*simulator*). In this approach, a *simulator* can be used to run several types of *models*. DEVS defines two types of models: *atomic* and *coupled*. The mathematical form of such models can be found in [Zeigler *et al.*, 2018]. An *atomic model* describes the behavior of a basic functional unit that cannot be further divided into sub-units. The DEVS atomic model specification requires defining three sets (input, outputs, and sequential states), two transition functions, and an output function. On the other hand, a *coupled model* describes the structure of a unit as a set of models. Such models can be either DEVS *atomic* or *coupled*. They should be defined in the DEVS coupled model specification. The specification also requires defining two sets (inputs and outputs) along with external and internal couplings among models. Hence, the *atomic model*

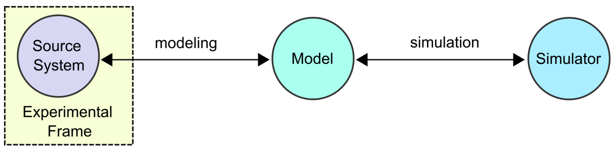


Figure 2. M&S framework (adapted from [Zeigler *et al.*, 2018]).

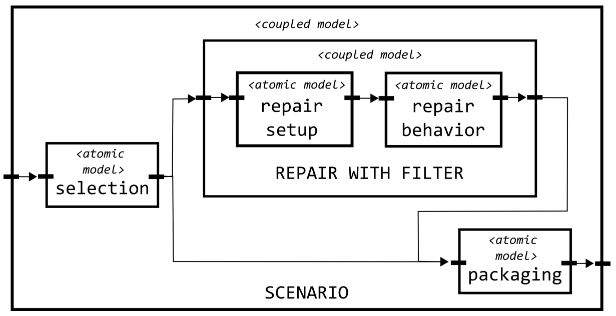


Figure 3. DEVS-based solution for the scenario depicted in **Figure 1**. Stereotypes are used to denote the type of DEVS model.

represents the *behavioral modeling level*, while the *coupled model* represents the *structural modeling level*.

Often, discrete-event simulation scenarios require selectively sending or receiving events between model components (as in the example described in the previous section). A *routing process* is "the part of a modeling scenario where the components should interact by distinguishing the event sources and destinations to ensure their diffusion into the correct model" [Blas *et al.*, 2022]. In our example, the *routing process* is attached to the interaction between SELECTION and REPAIR due to the defined setup. When building a simulation model to explain a routing process, the modeler is dealing with a *routing problem*.

In DEVS, routing processes are commonly solved as pre-wired connections (detailed as part of the coupling specification) or through the modeling of handlers. For example, the DEVS models used to solve the scenario depicted in **Figure 1** can be defined as shown in **Figure 3**, following the premises detailed in [Blas *et al.*, 2022]. This solution includes four atomic models (one per machine type, plus one filter to handle the setup within REPAIR) and two coupled models (REPAIR WITH FILTER and SCENARIO).

However, taking advantage of the modular and hierarchical nature of DEVS, a deeper analysis of DEVS design patterns has been conducted to improve the modeling of routing processes [Blas *et al.*, 2022]. Based on the analysis of explicit and implicit DEVS-based solutions to routing problems, we identified emerging structures, and as a result, the RDEVS formalism was proposed. In this way, RDEVS is a modular and hierarchical M&S formalism based on DEVS that provides a set of design models for addressing routing problems within DEVS models.

2.1 The Context: The Routed DEVS Formalism

As stated before, the RDEVS formalism is designed to simplify the modeling of routing problems. RDEVS is closed under coupling [Blas *et al.*, 2022]. Zeigler [Zeigler, 2018] considers that two questions arise regarding the closure under coupling of DEVS-based formalisms:

Q1) are they subsets of DEVS, behaviorally equivalent to DEVS but more expressive or convenient, or do they bring new functionality to DEVS?, and

Q2) have simulators been provided for them to enable verification and implementation?.

The answers to these questions for the RDEVS formalism are the following:

A1) 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 that enhance the DEVS formalism. By isolating the *routing behavior* from the *domain-specific behavior*, routing processes are built with a strict separation of concerns. All these features are related to the “model” perspective (from the M&S framework) explained in Section 2.1.1.

A2) While 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 (this is due to the equivalence shown by closure under coupling). This feature is related to the “simulator” perspective (from the M&S framework) explained in Section 2.1.2.

2.1.1 The “Model” Perspective

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). Hence, an RDEVS-based solution for the scenario depicted in **Figure 1** is presented in **Figure 4**. Such a solution is composed of three routing models with their routing policies (defined as mathematical functions of the models) and an additional network model. The essential models embedded in the routing models are equivalent to *selection*, *packaging*, and *repair behavior* (i.e., the atomic models) used in **Figure 3**. However, fewer models are defined in the RDEVS solution since the filter is introduced into the models as part of the routing policy. It is easy to see that RDEVS acts as a “layer” above DEVS, providing routing functionality without requiring the user to “dip down” to DEVS itself for any functions [Zeigler, 2018].

By using the models described above, RDEVS divides the *behavioral modeling level* of DEVS into two types: *domain behavior* and *routing behavior* (see **Table 1**). 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. Thus, the design applied over DEVS to build RDEVS allows: *i)* reuse of DEVS atomic models as the behavioral specification in RDEVS essential models, and *ii)* coupled RDEVS network models to DEVS models (either atomic or coupled) by taking advantage of routing processes only when required (this is because routing processes are performed inside network models and routing policies cannot be shared by several networks).

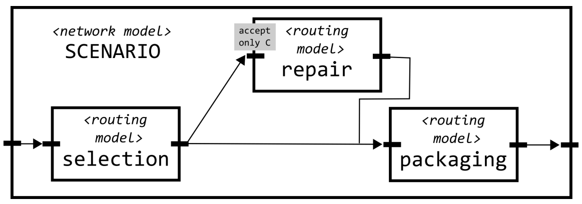


Figure 4. RDEVS-based solution for the scenario depicted in **Figure 1**. Stereotypes are used to denote the type of RDEVS model. The routing policy of REPAIR is introduced with a mathematical function included in the model. Such a function is represented in the figure by the gray box.

2.1.2 The “Simulator” Perspective

In DEVS, models are executed using a modular and hierarchical definition of simulators. Three types of simulators are defined:

- *Simulator*: Simulator used to execute DEVS atomic models.
- *Coordinator*: Simulator used to execute DEVS coupled models.
- *Root-Coordinator*: Simulator used to initialize the simulation and manage the time advance at execution.

These simulators share information using a well-defined message exchange protocol based on an event list. An initialization message *(i,t)* is sent (at the initialization stage) by each parent to all its subordinates. The next scheduled internal event is carried out through the state transition message *(*,t)* (sent by the coordinator to its imminent child). In this case, an output message *(y,t)* is sent by the child to the coordinator. Meanwhile, the coordinator sends input messages to its subordinates to produce external events. **Figure 5** shows how the hierarchical DEVS model presented in **Figure 3** is related to the hierarchical structure of simulators (i.e., a concrete realization of the “simulation” relationship illustrated in **Figure 2** for the model depicted in **Figure 3**). Since the SCENARIO model (**Figure 3**) is defined as a set of two coupled models (SCENARIO and REPAIR WITH FILTER) and four atomic models (selection, packaging, repair setup, and repair behavior), the set of simulators required to support its execution is composed of one root coordinator, two coordinators (one per each coupled model), and four simulators (one per each atomic model). These simulators interact at runtime to execute the instructions detailed in the models.

As stated before, the simulator structure proposed in the DEVS abstract simulator can be used to execute RDEVS simulation models. Since routing models manage the routing policy as part of their behavior, the simulator does not need to handle additional information regarding the simulation algorithm. In this way, a DEVS simulator can be attached to

Table 1. Modeling levels proposed in DEVS and RDEVS formalisms.

	Modeling Level	**Model**	**Description**
DEVS	Behavior	Atomic Model	An *atomic model* describes the autonomous behavior of a discrete-event system as a sequence of deterministic transitions between sequential states by specifying how the system reacts to external input events and generates output events.
	Structure	Coupled Model	A *coupled model* describes a system as a structure of coupled components that can be either atomic or coupled models. Connections are defined as couplings denoting how components influence each other.
RDEVS	Domain Behavior	Essential Model	An *essential model* describes the behavior of a component in the same way as the DEVS atomic model.
	Routing Behavior	Routing Model	A *routing model* describes the external behavior of a discrete-event model that takes action inside a routing process. The internal behavior of such a model is defined by the essential model embedded in it.
	Routing Structure	Network Model	A *network model* describes a routing process as a structure of routing models (i.e., components) designed to solve event routing across components.

each routing model while, for the network model, DEVS coordinators can be employed. Because essential models are embedded inside routing models, explicit simulator components are not required. Then, the message exchange protocol is a convenient tool to run both DEVS and RDEVS simulation models combined in the same execution.

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 *Tin* 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 *Rd* routing models composing N. Each *Rd* 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 λM is executed to produce the identified output event y'. To get y', the model combines the result of λE 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).

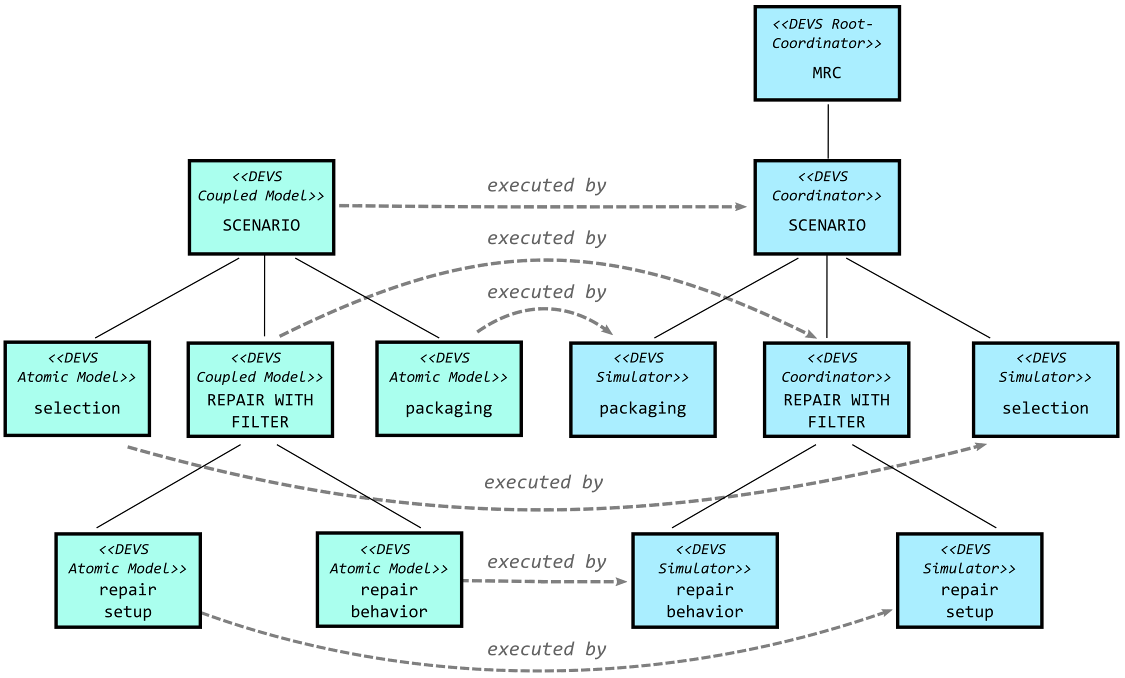
If at any time, an event y' has no internal destination in N, y' should be sent outside N. Then, the network model executes the output translation function *Tout* to get an event y (with $y \in Y$) from the identified event y'. Such a function removes the routing information attached to y' propagating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

2.2 The Problem: Getting Event-based Data from RDEVS Simulations

Even though RDEVS models can be run with DEVS simulators, the data obtained from such a process will be DEVS-based. This 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) by 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 dynamics 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 that allows collecting 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 *Q1)* and *Q2)*, respectively. The former presents an alternative solution with a separation of concerns that allows maintaining the advantages of using DEVS simulators as defined in the RDEVS conception. In this solution, the DEVS-based data collected by the simulator can be complemented with RDEVS-based data in a way that allows studying the dynamic behavior of routing policies and how simulation models interact during the simulation. On the other hand, the latter maintains the simulator as the engine



<<DEVS Root-Coordinator>> MRC
<<DEVS Coupled Model>> SCENARIO
executed by
<<DEVS Coordinator>> SCENARIO
executed by
executed by
<<DEVS Atomic Model>> selection
<<DEVS Coupled Model>> REPAIR WITH FILTER
<<DEVS Atomic Model>> packaging
<<DEVS Simulator>> packaging
<<DEVS Coordinator>> REPAIR WITH FILTER
<<DEVS Simulator>> selection
executed by
<<DEVS Atomic Model>> repair setup
<<DEVS Atomic Model>> repair behavior
executed by
<<DEVS Simulator>> repair behavior
<<DEVS Simulator>> repair setup
executed by

Figure 5. DEVS hierarchical simulator related to a hierarchical model depicted in **Figure 3**. Messages are not included. Background colors refer to the components presented in **Figure 2**.

that collects all the dynamic data attached to the model. The main problem with this approach is that the required feature depends on behavior modeling levels (domain and routing), mainly controlled by routing policies used in the nodes. In this case, the solution will require the simulator to know detailed information regarding the decisions made by the model at the behavioral level. To do this, we should: *i)* modify the message exchange protocol of simulators by introducing new messages to capture such behaviors, and *ii)* introduce a new type of simulator that can be combined with the DEVS simulator hierarchy.

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 obtaining (structured) data from RDEVS simulations using event trackers (i.e., the solution *i)*). Such a modeling-based solution is centered on the Decorator pattern (Section 2.3) as a means to add responsibilities to the well-defined conceptualization of RDEVS simulation models defined in [Blas *et al.*, 2022] without altering their behavior.

2.3 The Solution: The Decorator (Design) Pattern

In software engineering, restructuring is defined as the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)[Chikofsky and Cross, 1990]. On the other hand, refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Fowler, 2018]. Restructuring creates new versions that may propose changes to the subject system, but it does not involve modifications driven by new requirements. When combined with refactoring, it leads to reengineering, which is the examination and alteration of a subject system to reconstitute it in a new form followed by the implementation of that new form [Mens and Tourwé, 2004].

With the aim of changing the appearance of RDEVS models by introducing trackers as part of their practical definition, we employ the notion of restructuring implemented by refactoring the RDEVS library through the use of a design pattern. A design pattern is the reusable form of a solution to a design problem [Alexander, 2018]. Design patterns are general, reusable solutions defined from the study of commonly occurring problems within a given context, and they help ensure the success of the modeling task (e.g., OO design patterns capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities [Gamma *et al.*, 1995]).

The Decorator design pattern is one of the twenty-three well-known design patterns proposed by [Gamma *et al.*, 1995]. 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 (see **Figure 6**).

According to [Gamma *et al.*, 1995], the applicability of the pattern is mainly devoted to *i)* adding responsibilities to individual objects without affecting other objects, *ii)* responsibilities that can be withdrawn, or *iii)* situations when extension by subclassing is impractical.

As stated before, since DEVS is more likely to be computationally represented with an OO paradigm, we use the

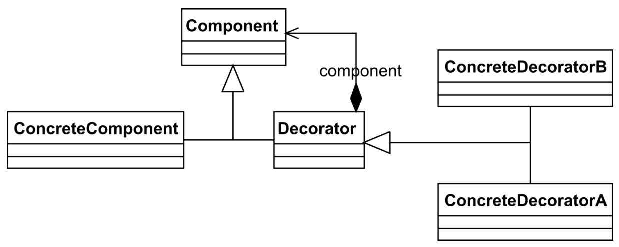


Figure 6. Structure of the Decorator design pattern (adapted from [Gamma *et al.*, 1995]).

conceptual model of RDEVS as the object model to be refactored with the decorator design pattern. Even though there are other alternatives, the use of the decorator pattern provides the following benefits to our proposal:

- It allows the addition of trackers (as decorators) to RDEVS models without modifying the simulator definition, avoiding subclassing (i.e., the need to create multiple subclasses to extend the behavior of an object) which would require adjusting the simulator to deal with trackers.
- It provides proper separation of concerns, since the decorator is responsible for a specific functionality while the model responsibilities remain unchanged, allowing the decorators to be included on demand at runtime.
- It improves maintainability, since the decorator is responsible for a specific functionality that can be added, removed, or modified without affecting the rest of the model structure.

Given that for the RDEVS formalism: *i)* we want to add a new responsibility (tracking) without affecting how models work, *ii)* we want (in the future) to selectively choose the models to track, and *iii)* defining new subclasses is not feasible, as it implies creating new types of simulation models; we decided to use the Decorator pattern for the redesign.

3 RDEVS with the Tracker Add-On Functionality

At this point, two remarks are critical for understanding our proposal. First, we do not intend to introduce new functionality to the RDEVS formalism. By restructuring the foundational design, we want to show that it is possible to "decorate" the formal model design without altering its 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 aim 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 Section 2.1.2). Indeed, we want to track events processed by the domain behavior of both sides (source and destination) as a result of routing policies.

3.1 Redesign at the Conceptual Modeling Level

Since DEVS is typically represented computationally using an OO paradigm, we use the conceptual model of RDEVS (**Figure 7**) proposed in [Blas and Gonnet, 2021] as the UML representation of the formalism to be redesigned. Such a diagram is presented as a computational representation in a metamodel of the RDEVS specification. From such a class diagram, **Figure 8** shows how the pattern was applied to support event trackers. Due to space limitations, the formal definition of the conceptualization in the mathematical form of RDEVS models is omitted.

Figure 8 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 the 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).

Once the trackers have been added as "decorators" of RDEVS models, their own conceptualization is needed. **Figure 9** shows the conceptualization used for trackers to record data related to the attached RDEVS model (i.e., the *RDEVS Component*).

In **Figure 9**, 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

As stated before, the RDEVS formalism is a subclass of DEVS that builds "new types of models" over DEVS. Hence, existing DEVS implementations can be used to support RDEVS implementations. We have used DEVSJAVA [Sarjoughian and Zeigler, 1998] as the underlying M&S layer for the RDEVS library. The Java classes implemented as RDEVS concepts were linked to the classes proposed in the M&S layer for simulation compatibility. All the models defined in the RDEVS formalism were implemented as Java classes that depend on DEVSJAVA classes. In this way, the library supports the implementation of routing structures over discrete-event system models based on the RDEVS formalism. Furthermore, since DEVS Suite [Kim *et al*., 2009] is a software tool defined based on DEVSJAVA, the implementation of RDEVS models can be tracked by DEVS Suite to get DEVS-based data.

To update the existing implementation of RDEVS models with event trackers, we perform a refactoring of the RDEVS Java Library [Espertino *et al*., 2024] following the conceptualizations previously described. These models were implemented as a new Java project related to the existing implementation.

3.2.1 How Does the Tracking Work?

To explain how the conceptual models presented in the previous section can record dynamic information regarding the event flow, **Figure 10** presents a UML sequence diagram. This diagram shows the interactions among trackers, models, and simulators at execution time. We use the notions of "Tracker Decorator" and "RDEVS Component" as the ones depicted in **Figure 9**, and "Simulator" as a generic DEVS Simulator that can be either a Simulator or Coordinator.

At the beginning of each simulation, the trackers associated with the models are automatically created through the initialization process (steps 1 to 7). During these steps, the data related to the model structure (i.e., static data) is collected by each tracker (e.g., for the routing model tracker: *identifier, name, input port names, output port names*, and so on). This data depends on the type of RDEVS model.

Once the models are running in a simulation (from step 8 onwards), the dynamic data is collected. This data is produced by the identified events exchanged among routing models. When the simulator detects that a model has an imminent internal transition to be executed, it allows the model to fire such a transition after sending the corresponding output event (steps 9 to 15). When executing the output function, for each event sent, the tracker collects the following data in an *Event* (defined in **Figure 9**): *highLevelType, type* (*Internal* or *External*), *source* (output port from which it has been sent), and *destination* (input ports to which it is intended). Following these steps for each output event, 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*).

A similar strategy is used on the trackers representing ports on which events are attempted to be sent due to an external transition (steps 16 to 19). Moreover, RDEVS models use the routing function to decide whether to accept or reject the input event when executing the external transition function. For rejection, no data is particularly recorded in the tracker (step 22). However, 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 (steps 23 to 26). This *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 (from step 27 onwards), all the dynamic data related to event exchange is available as part of the tracker model's structure. 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 structure the JSON file, we use the tag "@Expose" for predefined navigation among the Java classes implemented. This navigation is designed to store the minimum set of data required to rebuild the model.

4 Results: Tracking Routing Processes implemented over RDEVS Simulation Models

To show how trackers are attached to models and how the data collection process works, we expand the three-node example presented in **Figure 1**. As previously stated, this scenario involves three machines: SELECTION, REPAIR, and PACKAGING connected to process distinct types of containers (A, B, or C). New setups 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, this scenario can be modeled through a routing process composed of three nodes (one per machine) using the model depicted in **Figure 4**, but updating the routing policies attached to each node (i.e., redefining their routing functions). Then, considering how the RDEVS formalism deals with events at execution time, some events will not be processed in the nodes due to their routing policies. It is important to note that, in the case of adopting DEVS-based modeling, the number of models involved in the solution will increase since each node should be modeled as the REPAIR WITH FILTER node depicted in **Figure 3**.

Following the model presented in **Figure 4**, **Figure 11** shows the structure of the RDEVS models in terms of the simulator hierarchy and its relationship with the trackers in-

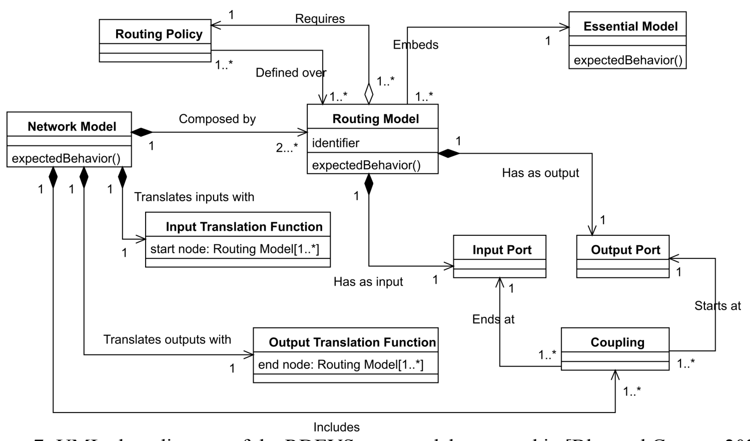


Figure 7. UML class diagram of the RDEVS metamodel proposed in [Blas and Gonnet, 2021].

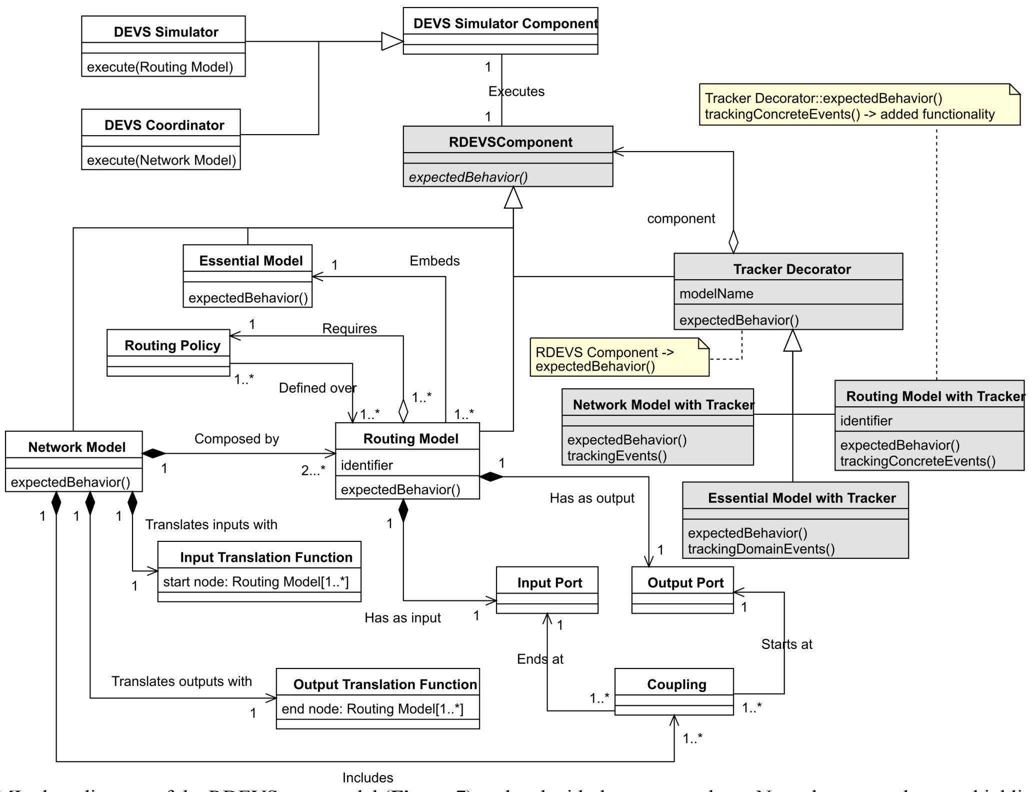


Figure 8. UML class diagram of the RDEVS metamodel (**Figure 7**) updated with the event trackers. New classes are the ones highlighted in gray.

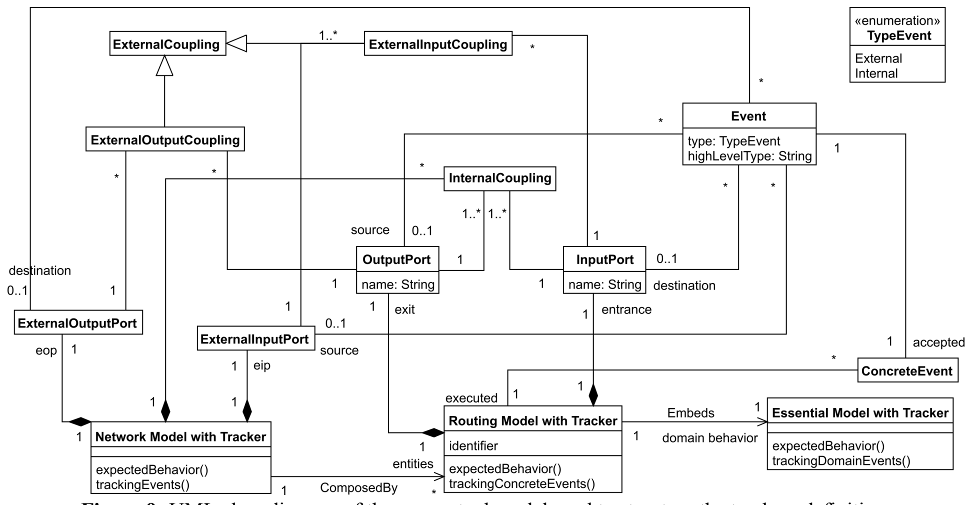


Figure 9. UML class diagram of the conceptual model used to structure the trackers definition.

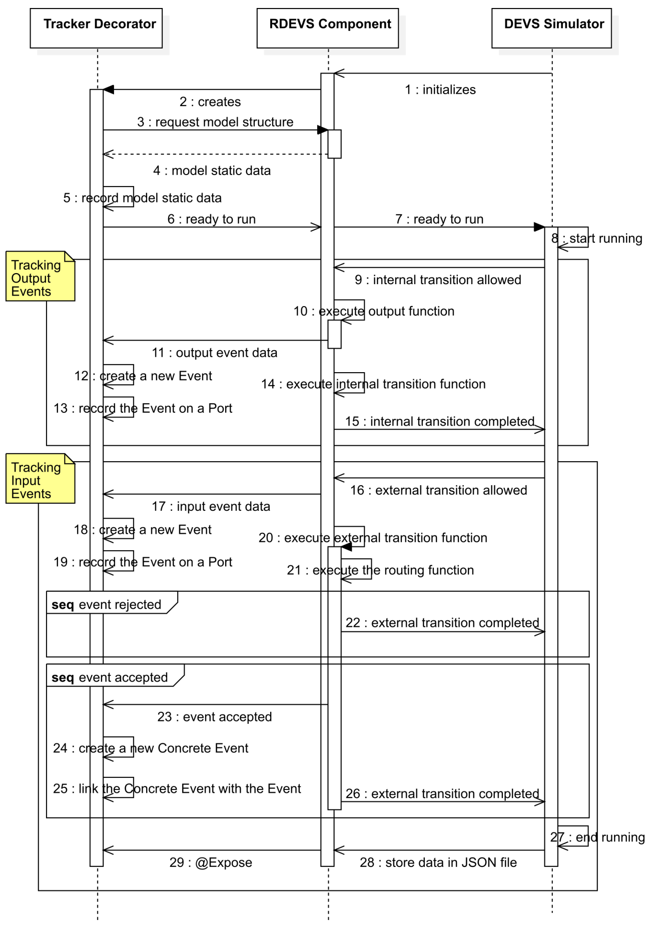

Tracker Decorator
RDEVS Component
DEVS Simulator
1 : initializes
2 : creates
3 : request model structure
4 : model static data
5 : record model static data
6 : ready to run
7 : ready to run
8 : start running
Tracking Output Events
9 : internal transition allowed
10 : execute output function
11 : output event data
12 : create a new Event
14 : execute internal transition function
13 : record the Event on a Port
15 : internal transition completed
Tracking Input Events
16 : external transition allowed
17 : input event data
18 : create a new Event
20 : execute external transition function
19 : record the Event on a Port
21 : execute the routing function
seq event rejected
22 : external transition completed
seq event accepted
23 : event accepted
24 : create a new Concrete Event
25 : link the Concrete Event with the Event
26 : external transition completed
27 : end running
29 : @Expose
28 : store data in JSON file

Figure 10. UML sequence diagram representing how the tracking process works.

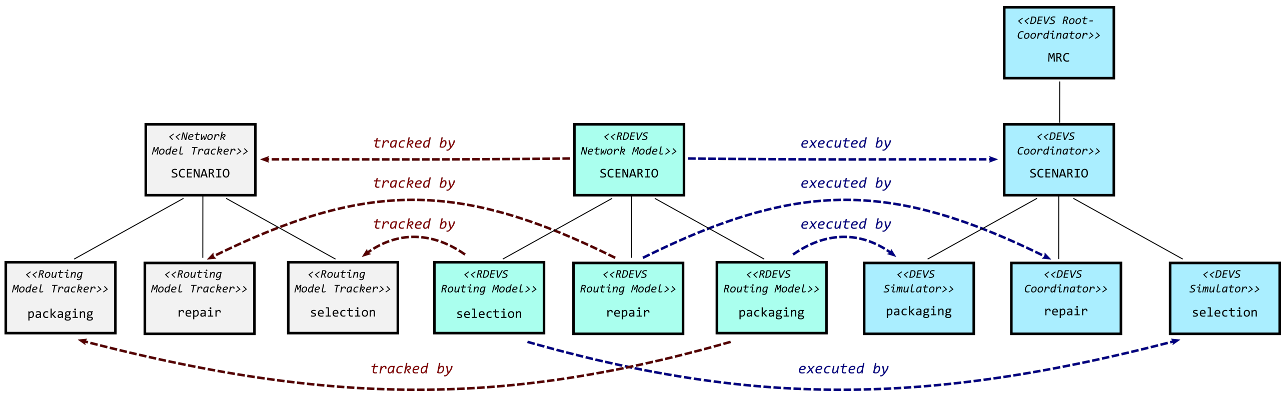
troduced as decorators in the model definition. At the center, each node is modeled as an RDEVS routing model composing the RDEVS network model named SCENARIO. On the right, the DEVS simulator hierarchy used to execute RDEVS models is presented. Such a hierarchy is based on the definitions introduced in Section 2.1.2. As in **Figure 5**, each type of model is attached to a simulator type. In particular, DEVS simulators are used to execute DEVS routing models and a single DEVS coordinator is used to execute the RDEVS network model. For example, the *repair* machine is modeled by the RDEVS routing model named *repair*, and such a model is attached to the DEVS simulator named *repair*. Finally, at the left, the trackers automatically created as decorators of the models are shown. These new elements are presented as distinct components attached to the models since, as detailed in **Figure 9**, they follow their own structure (similar to the simulator hierarchy). Hence, a Routing Model Tracker is defined for each RDEVS routing model to collect the event data attached to each node (e.g., the *packaging* Routing Model Tracker is attached to the *packaging* Routing Model). For the network model (i.e., SCENARIO), a Network Model Tracker (named SCENARIO) is used.

The structure presented in **Figure 11** shows how an RDEVS simulation is performed by combining the DEVS simulator hierarchy with our tracker proposal over RDEVS models. Such a structure is created each time a simulation is executed. As a result of execution, the trackers allocate data related to the event flow executed by the DEVS simulator hierarchy and accepted/rejected by the RDEVS models. **Figure 12** shows part of the JSON file obtained as output data of the routing model attached to the REPAIR node when performing a simulation. As expected, it 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 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 being enabled to accept containers of type C.

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. The authors of [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). Hence, 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 13**.

In a Sankey diagram, the quantity of each flow is represented by its width. This diagram is particularly useful to show multiple paths through a set of stages, helping to locate dominant contributions to an overall flow. For our purposes, we decided to represent RDEVS routing models as nodes, and the number of events exchanged, following routing policies as the width of the flows. In this way, this type of diagram can help to understand how RDEVS models have accepted or rejected events.

It is important to note that the Sankey diagram depicted in **Figure 13** is just an example of how the data collected by the trackers can be used to analyze the RDEVS simulations. Other forms of graphical representations can be used starting from the JSON file obtained as the output of execution. The Sankey diagram shown here was built using the data recovered from the simulation as input in a graphical application of our research laboratory. This application generates the diagram using the JSON file and produces an HTML file that contains the Sankey diagram. The diagram is directly visualized in the browser, allowing for a more accurate visualization.



<<DEVS Root-Coordinator>> MRC
<<Network Model Tracker>> SCENARIO
<<RDEVS Network Model>> SCENARIO
<<DEVS Coordinator>> SCENARIO
tracked by
executed by
<<Routing Model Tracker>> packaging
<<Routing Model Tracker>> repair
<<Routing Model Tracker>> selection
<<RDEVS Routing Model>> selection
<<RDEVS Routing Model>> repair
<<RDEVS Routing Model>> packaging
<<DEVS Simulator>> packaging
<<DEVS Coordinator>> repair
<<DEVS Simulator>> selection

Figure 11. Block diagram of the Tracker-Model-Simulator used to execute the scenario presented in **Figure 1**. Routing policies are not included. Background colors refer to the components presented in **Figure 2** (except for trackers).

5 Discussion

Nowadays, collecting data is a key step in the data processing of any model. If such data is structured, the processing to collect information is less complex. Structured data inserts a data warehouse already tagged and easily sorted, while unstructured data is random and difficult to analyze [Sagiroglu and Sinanc, 2013]. Hence, when collecting data, it is better to use a clearly structured format. Moreover, data collection is the pivotal step in the processing of a model.

For discrete-event based models, the data is primarily related to events. Over the years, several researchers have 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. From such experiments, data can be dynamically selected and observed (in tabular data output) for any number of DEVS models. In DEVS Suite, the data generated by the simulation models is collected dynamically and displayed as time-based trajectories using the same approach.

From a different perspective, more recently, the authors of [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. This allows tracing the simulation through the events. 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 using this approach can be considered structured data.

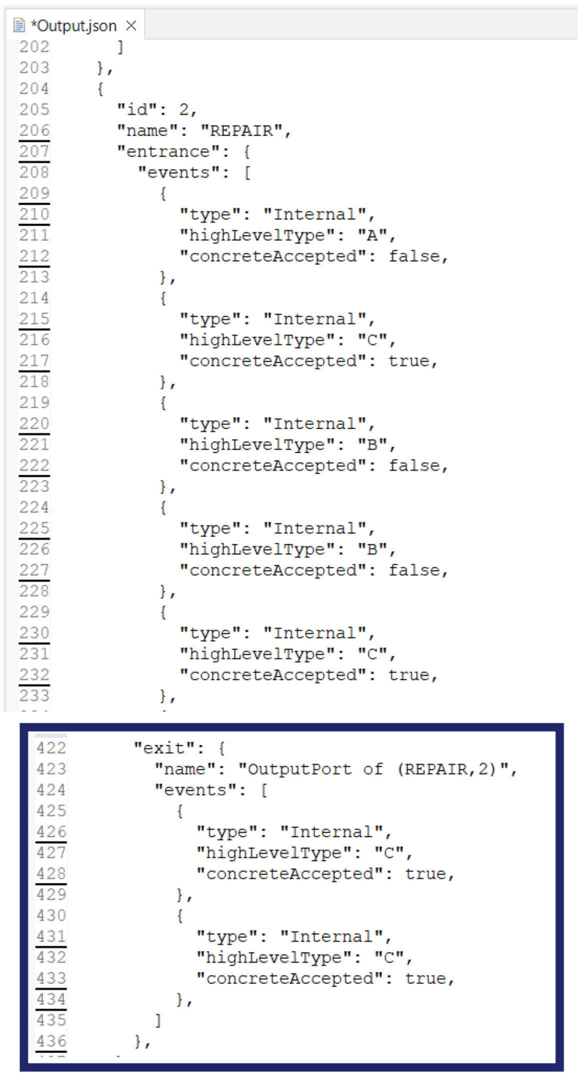
Our solution improves the understanding of the RDEVS formalism as an alternative conceptualization of DEVS models executed over DEVS simulators. As in [Dahmani *et al.*, 2020], we obtain structured data stored in a well-known format that facilitates further analysis. Moreover, in all cases, tracing mechanisms are hidden from the modeler. In our case, they are also hidden from the simulator (maintaining the separation of concerns defined in the M&S Framework proposed with DEVS). We avoid making the simulator aware of the trackers (i.e., we prevent it from knowing detailed information regarding the decisions made by models at both routing and behavioral levels). By deploying trackers as part of the RDEVS library (supported by DEVSJAVA [Sarjoughian and Zeigler, 1998]), DEVS Suite can be used to obtain DEVS-based results. Global reporting using both types of results can be produced without much effort. By using JSON formatting we enable integration with other software tools for in-depth analysis of RDEVS models. Furthermore, specific visualizations can be designed to enhance comprehension of the intricate routing processes modeled with RDEVS. Here, as an example of such visualizations, we employ Sankey diagrams to represent the flow of data between nodes. As stated, we developed a parser that takes the JSON data with the RDEVS structure and generates an HTML file to display the Sankey diagram automatically.

Tracing simulation experiments is usually computationally and data-intensive. As these examples show, our alternative solution provides an accurate separation of concerns that allows maintaining the advantages of using DEVS simulators (DEVSJAVA) for executing RDEVS models (implemented using the RDEVS library) while collecting structured data regarding how routing policies (at a higher decision level) are allowing/blocking domain processing at the lower operational level. At this point, it is important to note that the solution presented is described as an add-on responsibility of the RDEVS models themselves. Performance analyses are not applicable at this stage.

Given the nature of our proposal, adopting other DEVS simulators to execute RDEVS models decorated with trackers can be seen as a threat to validity. Due to the novelty of RDEVS, we are employing the only existing implementation of the RDEVS conceptualization (i.e., the one based on DEVSJAVA). To overcome this threat, new software related to RDEVS should be developed. Even so, it is worth highlighting that our proposal is centered on the conceptual design of the formalism (so the main features of it should be maintained across different software tools).

6 Conclusions and Future Work

To address the limitations produced by DEVS simulators, a conceptual modeling-based solution using event trackers was



```
        ]
      },
      {
        "id": 2,
        "name": "REPAIR",
        "entrance": {
          "events": [
            {
              "type": "Internal",
              "highLevelType": "A",
              "concreteAccepted": false,
            },
            {
              "type": "Internal",
              "highLevelType": "C",
              "concreteAccepted": true,
            },
            {
              "type": "Internal",
              "highLevelType": "B",
              "concreteAccepted": false,
            },
            {
              "type": "Internal",
              "highLevelType": "B",
              "concreteAccepted": false,
            },
            {
              "type": "Internal",
              "highLevelType": "C",
              "concreteAccepted": true,
            },
```

```
        "exit": {
          "name": "OutputPort of (REPAIR,2)",
          "events": [
            {
              "type": "Internal",
              "highLevelType": "C",
              "concreteAccepted": true,
            },
            {
              "type": "Internal",
              "highLevelType": "C",
              "concreteAccepted": true,
            },
          ]
        },
```

Figure 12. 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).

proposed to gather structured data from RDEVS simulations. We have altered the conceptual design of the RDEVS implementation by including tracking responsibilities as "decorators" of the set of elements used in discrete-event models. Through restructuring, we change the appearance of RDEVS models by introducing the trackers as part of their practical definition. As shown in Section 4, the addition of trackers does not affect either the model's behavior or its structure. The restructuring was implemented (in practice) by refactoring the RDEVS library. In this way, RDEVS-based data can be obtained from RDEVS simulations as a complement to the DEVS-based data obtained from the DEVS simulator. Hence, our solution maintains the key benefits of the model-simulator separation that underpinned the design of the RDEVS formalism.

The example presented in Section 4 was used to show how trackers work starting from the conceptualization defined. Even so, any case study where RDEVS models are used will be enhanced with the tracker's additional feature since it allows the characterization of the event flow without extra effort. Since our research group is devoted to developing RDEVS models in different areas, we have used the proposal of this paper for the M&S of network protocols [Alshareef *et al.*, 2022], software architectures [Blas *et al.*, 2020] [Blas and Gonnet, 2022], and electric power systems [Blas and Alvarez, 2022] [Blas *et al.*, 2023a].

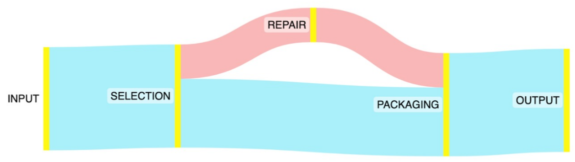


Figure 13. A Sankey diagram. Flow color is used to denote different paths. INPUT and OUTPUT are nodes denoting external input and output flows from/to the network model to/from routing models.

By allowing the collection of RDEVS-based data through tracking, we expect to improve the extensibility, maintainability, and readability of RDEVS models. The DEVS-based data gathered by the DEVS simulator can be complemented 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 extract information is less complex. Moreover, such processing can be developed using a general-purpose programming language or a specific software tool with JSON processing functionality. Hence, defining an output structure in the JSON format was essential to effectively present information from all simulation runs, ensuring accessibility and clarity for further analysis and interpretation. We are now working on the development of a web application that will allow practitioners to upload JSON files and generate charts and tables with the processed data (i.e., information). This is the final goal of our research project at this stage. This tool significantly contributes to RDEVS research, helping us to understand the model's routing behavior. In future work, we plan to extend the trackers' proposals with other features required based on the past experiences of our research group when analyzing RDEVS models.

We strongly believe that the solution presented for 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 and can be used to incorporate other functionalities to the simulation models as add-on responsibilities. In the future, we plan to include more add-on functionalities in our proposal. Other design patterns will be explored.

Declarations

Acknowledgements

The authors would like to express their gratitude to the Consejo Interuniversitario Nacional (CIN) and Universidad Tecnológica Nacional (Argentina) for their financial support, which facilitated the scholarship awarded to Mateo Toniolo for the accomplishment of this work.

Funding

This research was funded by Universidad Tecnológica Nacional [Grant Numbers FESIEC275 and FESIEC278].

Authors' Contributions

Maria J. Blas contributed to the conception of this study. Mateo Toniolo developed the implementation of the RDEVS Trackers, designed in collaboration with Maria J. Blas and Silvio Gonnet. Maria J. Blas is the primary contributor and writer of this manuscript. Silvio Gonnet supervised the entire work. All authors have read and approved the final manuscript.

Competing interests

The authors declare that they have no conflict of interest.

Availability of data and materials

Data can be made available upon request.

References

Alexander, C. (2018). *A pattern language: towns, buildings, construction*. Oxford university press. Book.

Alshareef, A., Blas, M. J., Bonaventura, M., Paris, T., Yacoub, A., and Zeigler, B. P. (2022). Using devs for full life cycle model-based system engineering in complex network design. In *Advances in Computing, Informatics, Networking and Cybersecurity: A Book Honoring Professor Mohammad S. Obaidat's Significant Scientific Contributions*. Springer. DOI: 10.1007/978-3-030-87049-2_8.

Blas, M., Alvarez, G., and Sarli, J. (2023a). Improving the optimization of electric power systems through a discrete event based simulation model. *Journal of applied research and technology*, 21(1):17–35. DOI: 10.22201/icat.24486736e.2023.21.1.2167.

Blas, M., Toniolo, M., and Gonnet, S. (2023b). Tracking events as an add-on functionality of the routed devs formalism. In *Anais do V Workshop em Modelagem e Simulação de Sistemas Intensivos em Software*, pages 41–50, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/mssis.2023.235691.

Blas, M. J. and Alvarez, G. E. (2022). A simulation approach to solve power system transmission problems. *International Journal of Mathematical, Engineering and Management Sciences*. DOI: 10.33889/IJMEMS.2022.7.2.011.

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. DOI: 10.1016/j.jestch.2020.12.006.

Blas, M. J. and Gonnet, S. (2022). Using model-to-model transformations for web software architecture simulation. *IEEE Latin America Transactions*, 20(4):545–552. DOI: 10.1109/TLA.2022.9675459.

Blas, M. J., Leone, H., and Gonnet, S. (2020). Modeling and simulation framework for quality estimation of web applications through architecture evaluation. *SN Applied Sciences*, 2(3):374. DOI: 10.1007/s42452-020-2171-z.

Blas, M. J., Leone, H., and Gonnet, S. (2022). Devs-based formalism for the modeling of routing processes. *Software and Systems Modeling*, pages 1–30. DOI: 10.1007/s10270-021-00928-4.

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17. DOI: 10.1109/52.43044.

Dahmani, Y., Ali, H. N. B., and Boubekeur, A. (2020). Xml-based devs modelling and simulation tracking. *International Journal of Simulation and Process Modelling*, 15(1-2):155–169. DOI: 10.1504/IJSPM.2020.106980.

Espertino, C., Blas, M. J., and Gonnet, S. (2024). Mapping rdevsnl-based definitions of constrained network models to routed devs simulation models. *Journal of the Brazilian Computer Society*, 30(1):17–34. DOI: 10.5753/jbcs.2024.3061.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional. Book.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH. DOI: 10.1007/3-540-47910-$4_2$1.

Kim, S., Sarjoughian, H. S., and Elamvazhuthi, V. (2009). Devs-suite: a simulator supporting visual experimentation design and behavior monitoring. *SpringSim*, 9:1–7. Available at:`https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2eac111762a2ea8e28ca4fbdfcdb65b0c5ec4177`.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139. DOI: 10.1109/TSE.2004.1265817.

Sagiroglu, S. and Sinanc, D. (2013). Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*, pages 42–47. IEEE. DOI: 10.1109/CTS.2013.6567202.

Sarjoughian, H. S. and Singh, R. (2004). Building simulation modeling environments using systems theory and software architecture principles. In *Proceedings of the advanced simulation technology conference*, pages 99–104. Available at`https://acims.asu.edu/wp-content/uploads/sites/18/2012/02/Building-Simulation-Modeling-Environments-Using-Systems-Theory-and-Software-Architecture-Principles.pdf`.

Sarjoughian, H. S. and Zeigler, B. (1998). Devsjava: Basis for a devs-based collaborative m&s environment. *Simulation Series*, 30:29–36. Available at:`https://www.academia.edu/18755704/Devsjava_Basis_for_a_devs_based_collaborative_ms_environment`.

Vernon-Bido, D., Collins, A., and Sokolowski, J. (2015). Effective visualization in modeling & simulation. In *Proceedings of the 48th annual simulation symposium*, pages 33–40. DOI: 10.5555/2876341.2876346.

Zeigler, B. P. (2018). Closure under coupling: concept, proofs, devs recent examples (wip). In *Proceedings of the 4th ACM International Conference of Com-*

puting for Engineering and Sciences, pages 1–6. DOI: 10.1145/3213187.3213194.

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

Zeigler, B. P. and Nutaro, J. J. (2016). Towards a framework for more robust validation and verification of simulation models for systems of systems. *The Journal of Defense Modeling and Simulation*, 13(1):3–16. DOI: 10.1177/1548512914568657.