


Mapping RDEVSNL-based Definitions of Constrained Network Models to Routed DEVS Simulation Models

Clarisa Espertino  [Universidad Tecnológica Nacional - FRSF | cespertino@frsf.utn.edu.ar]

María Julia Blas  [INGAR - CONICET & UTN | mariajuliabl意思@santafe-conicet.gov.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: 15 December 2022 • Accepted: 19 April 2023 • Published: 27 March 2024

Abstract The Routed DEVS (RDEVSNL) formalism has been introduced recently to provide a reasonable formalization for the simulation of routing processes over Discrete Event System Specification (DEVS) models. Due to its novelty, new software tools are required to improve the Modeling and Simulation (MS) tasks related to the RDEVSNL formalism. This paper presents the mapping between constrained network models obtained from textual specifications of routing processes and RDEVSNL simulation models implemented in Java. RDEVSNL context-free grammar (previously defined) is used to support the textual specification of a routing process as a constrained network model. Such grammar is based on a metamodel that defines the syntactical elements. This metamodel is used in this paper as a middleware that allows mapping constrained network model concepts with RDEVSNL simulation models. From such a constrained network model template, RDEVSNL Java implementations are obtained. The proposal is part of a work-in-progress intended to develop MS software tools for the RDEVSNL formalism using well-known abstractions to get the computational models through conceptual mapping. Using these tools, modelers can specify simulation models without needing to codify any routing implementation. The main benefits are i) reduction of implementation times and ii) satisfactory simulation model correctness regarding the RDEVSNL formalism.

Keywords: Discrete Event System Specification, Metamodeling, Context-free grammar, Modeling and Simulation.

1 Introduction

A formalism provides a set of conventions for specifying a class of objects in a precise, unambiguous, and paradigm-free manner. The Discrete Event System Specification (DEVS) is a modular and hierarchical Modeling and Simulation (M&S) formalism based on systems theory that provides a general methodology for the construction of reusable models [Zeigler *et al.*, 2018]. The Routed DEVS formalism (RDEVSNL) employs the “embedding routing functionality” strategy over DEVS models to provide routing capability from the simulation model conception. Such a DEVS extension was presented by Blas *et al.* [2017] as a subclass of the classic DEVS [Zeigler *et al.*, 2018] including routing features to the atomic model capabilities by adding the *routing model*. Through this new simulation model, the formalism act as a “layer” above DEVS that provides routing functionality without requiring the user to “dip down” to DEVS itself for any functions.

Although RDEVSNL is based on DEVS, it is a new formalism that emerges from the M&S community to address new types of problems. Due to its nature, RDEVSNL simulation models can be executed using DEVS simulators. However, structural differences between DEVS and RDEVSNL simulation models imply that DEVS modeling tools cannot be used to define RDEVSNL models. The core of RDEVSNL is to abstract the event flow into a new type of discrete-event model (i.e., the *routing model*) that arrange events independently from the domain behavior of components using a *routing policy*. The *routing policy* is isolated from the domain behavior (i.e., the *essential*

model). This feature allows the reuse of the formalized behavior in several *routing models*. All routing models together define a *network model*. A single *network model* can support different settings only by changing the *routing policies* attached to its *routing models*. The novelty of RDEVSNL structural features promotes the need to develop new software modeling tools that i) help practitioners with RDEVSNL specification tasks and ii) take advantage of DEVS simulators as execution engines. Hence, modeling tools based on high-level abstractions are preferred to promote implementation-independent modeling.

Aiming to build these software tools, we employ a two-level modeling strategy to get RDEVSNL simulation models. At the first level, we use a high-level description to define an abstraction model. Then, we employ a *routing process* definition as a standard formalization that can be mapped to RDEVSNL models. A *routing process* is a system of interacting components in which the operation of an element (i.e., a routing process component) and the routing of its outputs depend on what is happening throughout the process [Alsharief *et al.*, 2022]. That means interactions between components depend on local information and external data derived from the process structure. As shown later in this paper, for RDEVSNL models, local information is related to component behavior, and external data is attached to routing functions. Hence, routing processes provide a formalization definition for RDEVSNL simulation models.

In this context, network theory is a useful technique to model relationships between entities defined as nodes con-

nected by a set of links [Newman et al., 2011]. A particular case of *network models* is *constrained network models* (i.e., restricted networks). When *routing processes* are defined as a network of related entities, constraints are required to get reasonable definitions. As shown later in this paper, these models are defined as *constrained network models*.

This paper presents an approach in which textual specifications of *constrained network models* are translated to Java implementations of RDEVS simulation models. The objectives are *i*) to provide a transformation process between *constrained network models* and RDEVS simulation models by a *routing process* description and *ii*) to develop a new M&S software tool using Eclipse technology that supports both the textual specification of *routing processes* as *constrained network models* and the transformation process described at *i*). By achieving both objectives, we provide a solution in which modelers can describe a high-level routing situation as a constrained network textual description based on a Context-Free Grammar (CFG) and then get the related RDEVS simulation model implementation.

Following the two-level modeling strategy, we combine a *constrained network model* (i.e., a high-level description) with a *routing process* definition (i.e., a formalization-level description) to describe an accurate transformation that allows obtaining Java code (i.e., a low-level implementation) for RDEVS models. We use the CFG named RDEVS Natural Language (RDEVSNL) to define the structure of the *constrained network models*. Such a CFG has been proposed in previous work to offer a textual definition of routing situations using *constrained network models* and is supported by a metamodel that acts as a representation of *routing processes*. From an instance of such a metamodel, in this paper, we show how Java code can be obtained using Eclipse technology [The Eclipse Foundation, 2022a]. As a result, we present an Eclipse plug-in that extends the RDEVSNL Editor tool with the transformation process used to map a metamodel instance (i.e., a *routing process*) to Java classes. The template used to create these classes is based on the RDEVS Java Library. Hence, the main contributions of the paper are the two-level modeling approach used to support the development process and the software tool.

The remainder of this paper is organized as follows. Section 2 presents the RDEVS formalism introducing the Java Library used at the core of the transformation process. Section 3 details the implementation-independent modeling approach used to build the proposal. It includes a description of previous work devoted to the textual specification that supports constrained network models. It introduces the metamodel used to map a routing process with a constrained network model instance. From this metamodel, Section 4 details the transformation process developed from the high-level description obtained as a metamodel instance (i.e., a routing process) to the low-level implementation (i.e., Java classes for RDEVS implementations). It includes proof of concepts developed following the example presented by Blas et al. [2021] and two real-life scenarios modeled with RDEVS formalism taken from Blas et al. [2022]. Section 5 discusses our results and their relationship with other proposals. Section 6 is devoted to conclusions.

2 The RDEVS Formalism

The RDEVS formalism is an adaptation of Classic DEVS that adds routing features to the models by introducing a new modeling level: routing behavior [Blas et al., 2022].

DEVS models are designed to provide *behavior* and *structure* definitions through *atomic* and *coupled* models, respectively. In RDEVS, three types of models are formalized: *essential*, *routing*, and *network* models. These models take advantage of DEVS modeling levels by partitioning the behavior into two distinct modeling levels: *domain behavior* and *routing behavior*. The RDEVS *essential model* defines a DEVS atomic model that specifies a *domain behavior* (i.e., the primary behavior of a component). The *routing model* defines a container of an *essential model* that uses a routing policy to manage its inputs and outputs (i.e., it adds a *routing behavior* over a *domain behavior*). Finally, the *network model* defines a set of *routing models* coupled all-to-all to leave the routing functionality to routing policies (i.e., it describes a *structure* over *routing behaviors*). Based on the use of routing policies, RDEVS simulation models provide an accurate formal specification of *routing processes* using three modeling levels: *domain behavior*, *routing behavior*, and *structure*.

Centered in the M&S framework of DEVS [Zeigler et al., 2018], RDEVS models can be executed using DEVS simulators. Such a framework shows how M&S activities are related by using four conceptual entities [Zeigler and Nutaro, 2016]: *i*) the *source system* as the real/virtual environment to be modeled, *ii*) the *model* as the set of instructions for generating data comparable to the data observable in the *system*, *iii*) the *simulator* as the computation specified in the *model*, and *iv*) the *experimental frame* as the conditions under which the *source system* is observed. In this way, it emphasizes the notion of *model* and *simulator* as two independent entities linked by the *simulation* relationship when a model is executed on a computational environment (i.e., a *simulator*). Furthermore, starting from the M&S framework, RDEVS can be seen as a subclass of DEVS for modeling new types of problems [Blas et al., 2018].

DEVS extensions can be classified as variants or subclasses. The variants of DEVS refer to the subset of DEVS extensions in which the alternative formalism models a new type of system that, previously, could not be modeled with the original formalism. This is the case with Dynamic Structure DEVS, an extension that allows modeling systems with dynamic structures [Barros, 1997]. On the other hand, subclasses refer to the subset of DEVS extensions in which the alternative formalism improves the solution of a simulation scenario by applying DEVS models in a meaningful way. This is the case with RDEVS, an extension of DEVS designed to handle routing information in DEVS models. Such an extension improves the M&S of routing scenarios as a general mechanism that can be applied in distinct domains (e.g., supply chain, network protocol communications, and software architectures) [Blas and Gonnet, 2021; Alshareef et al., 2022; Blas et al., 2022]. The RDEVS capability of defining routing processes as a high-level abstraction for any domain (Section 3.1) provides a solid basis for mapping domain concepts with RDEVS models. Since RDEVS is a sub-

class of DEVS, the *model* entity of RDEVSNL formalism can be seen as a subclass of the DEVS *model entity*.

In conceptual modeling theory, the subtyping used to represent class-subclass dependence reflects the concepts at a more detailed specification level [Parsons and Wand, 1997]. Hence, any subclass of the DEVS *model* must be more specific than the DEVS *model*. For RDEVSNL, introducing a new modeling level is the core of such a specification. Given that the RDEVSNL model entity inherits a base set of the superclass properties (i.e., the properties of the DEVS *model entity*), the *simulator* used for its execution could be the same (i.e., the DEVS abstract simulator can execute RDEVSNL models). However, due to the modeling level distinction between DEVS and RDEVSNL formalisms, new modeling strategies are needed to support the specification of RDEVSNL models.

One way to do this is by using abstraction models to design formalization descriptions that allow getting the computational models (i.e., executable implementations) through conceptual mapping. This is the modeling approach called “implementation-independent modeling” presented in Section 3. For RDEVSNL models, such an approach is supported by the RDEVSNL Java Library (Section 2.2) to get RDEVSNL computational models.

2.1 Simulating Routing Processes

A *routing process* is a system of interacting components in which the operation of an element and the routing of its outputs depend on what is happening throughout the process. As this is a general definition, routing processes can be found in several domains, such as, for example, manufacturing and network communications. In this type of abstraction, the main objective of simulation studies is analyzing object flows among components (e.g., data or resources) along with components and system overall performance.

Depending on the domain, the routing process specification may contain different types of components. However, even when all components can be distinct among each other, each component type operates independently. This means that the internal operation of components can be defined as a black box that is independent from the structure of the routing process itself. Since routing depends equally on the operative description of the component and process structure, the component can decide the output destinations. Hence, components can take decisions about routing such as *i)* alternate the routing of its outputs to avoid congestion, *ii)* block the routing of its outputs from entering to a precise sector of predefined components, and *iii)* accelerate/decelerate the processing of its inputs (to produce faster/slower outputs) when knowing that downstream nodes are free/busy.

When routing processes are defined using models, several formalisms can be applied. For example, Petri nets have been used for modeling manufacturing systems [Kahraman and Tüysüz, 2010]. Such models can be later used in a simulation environment to study how the model works [Gehlot and Nigro, 2010]. DEVS models can be also used for the same purpose. However, when comparing the use of DEVS-based solutions with RDEVSNL-based solutions, RDEVSNL formalism reduces the modeling complexity. As explained in Blas *et al.* [2022], the reusability and flexibility of the RDEVSNL-based

solutions, along with model designs with low coupling and high cohesion, are the main benefits of using RDEVSNL instead of DEVS.

When getting RDEVSNL models from a *routing process* specification, the following statements are valid: *i)* the *network model* structures a *routing process functionality* as a composition of *routing models*; *ii)* the *routing model* represents the *routing functionality* of a *component* included in the *routing process*; and *iii)* the *essential model* describes the internal operation of a routing process component (i.e., the *component behavior*). Hence, each RDEVSNL model belongs to a modeling level and plays a role in a *routing process* specification (see **Table 1**).

The advantages of employing RDEVSNL for the M&S of routing situations are the following: *i)* the modeler does not need to dip down to DEVS itself to add routing functionality to models, *ii)* existing DEVS atomic models can be used to structure routing processes, *iii)* RDEVSNL and DEVS models can be combined to define complex M&S scenarios where routing processes interact with other types of phenomena, and *iv)* DEVS simulators can be used to execute RDEVSNL models.

Table 1. Routed DEVS simulation models, modeling levels, and routing processes. For simplicity, we consider a routing process as a set of linked components.

RDEVSNL Model	Modeling Level	Role in Routing Process
Essential model	Domain behavior	Component (behavior)
Routing model	Routing behavior	Component (routing functionality)
Network model	Structure	Process (routing functionality)

2.2 The RDEVSNL Library

As stated before, the RDEVSNL formalism is a subclass of DEVS. This means that existing implementations of DEVS can be used to support RDEVSNL implementations. From this perspective, a Java library was developed using DEVSNL [Sarjoughian and Zeigler, 1998] as the underlying M&S layer.

DEVSNL is a software tool implemented in Java that supports defining models in DEVS formalism through an object-oriented conceptualization. By extending DEVSNL, the RDEVSNL library provides a solution for implementing RDEVSNL simulation models in Java and, later, executing these models using the DEVSNL engine.

Figure 1 illustrates the main classes of the RDEVSNL library using a UML class diagram. As the figure shows, all the concepts included in the formalism were defined as Java classes. For example, the library includes a Java class for each type of RDEVSNL model defined in the formalism (i.e., *EssentialModel*, *RoutingMode*, and *NetworkModel*). Relationships are used to denote dependencies among classes. Then, for example, the library defines the routing policy of a rout-

ing model as the *RoutingFunction* linked to the *RoutingModel* definition as the *delr* property of the related Omega class (which is linked to the model by the *w* property). Moreover, the *NetworkModel* class is related to one instance of *InputTranslationFunction* class through the *Tin* property to define the transformation of input events to input events with identification (i.e., instances of the *IdentifiedEvent* class). A similar strategy is used to relate the *NetworkModel* with the *OutputTranslationFunction*. The *IdentifiedEvent* class is used to denote the structure of events managed by the models with routing functionality.

The diagram includes a few operations for the classes that represent RDEVSN simulation models. These operations are added to illustrate how the library supports the behavioral definition of routing processes. Considering the modeling levels detailed in **Table 1**, the classes defined in the library support the *structure* definition (i.e., the set of classes allows implementing the *NetworkModel* class using accurate routing model definitions). The *routing behavior* level is supported by the set of operations defined as “final” in *RoutingModel*. These operations (e.g., *delext()*, *delint()*, and *out()*) are implemented as Java code in the library and cannot be changed in further subclasses. In this way, the library ensures the correctness of the routing functionality defined at the core of the RDEVSN formalism. Finally, the operations defined as “abstract” in the *EssentialModel* class support the *domain behavior*. As with any abstract element, these operations require a Java implementation in further sub-classes to define the behavioral specification of routing components. Since the behavioral specification of such components is part of the domain problem, the RDEVSN library only provides the

interface required for their simulation.

The Java classes detailed in the library are designed as extension points for building RDEVSN implementations. An extension point is the definition of the provided interface for extensions [Klatt and Krogmann, 2008]. That is, an extension itself of the classes that represent RDEVSN simulation models is an implementation of the RDEVSN model according to the extension point defined in the library. Hence, the library includes extension points configured for designing explicit RDEVSN simulation models as reusable components slated for executing the routing simulation without any other consideration.

3 Implementation-Independent Modeling Approach

Abstraction creates a conceptual model by extracting only those elements needed for addressing a modeler’s concerns. M&S software tools with such a feature reduce the knowledge required for building discrete-event simulation models (here, RDEVSN models), and modeling tasks can be performed by anyone that understands the problem domain through the proposed abstraction. On the other hand, formalization makes it easier to work out the implications of an abstraction and implement them in reality [Zeigler et al., 2018]. As previously stated, the RDEVSN formalism is designed to level out the modeling effort of *routing processes* modeled in DEVS. Hence, *routing processes* can be used as formalization descriptions of high-level routing abstractions.

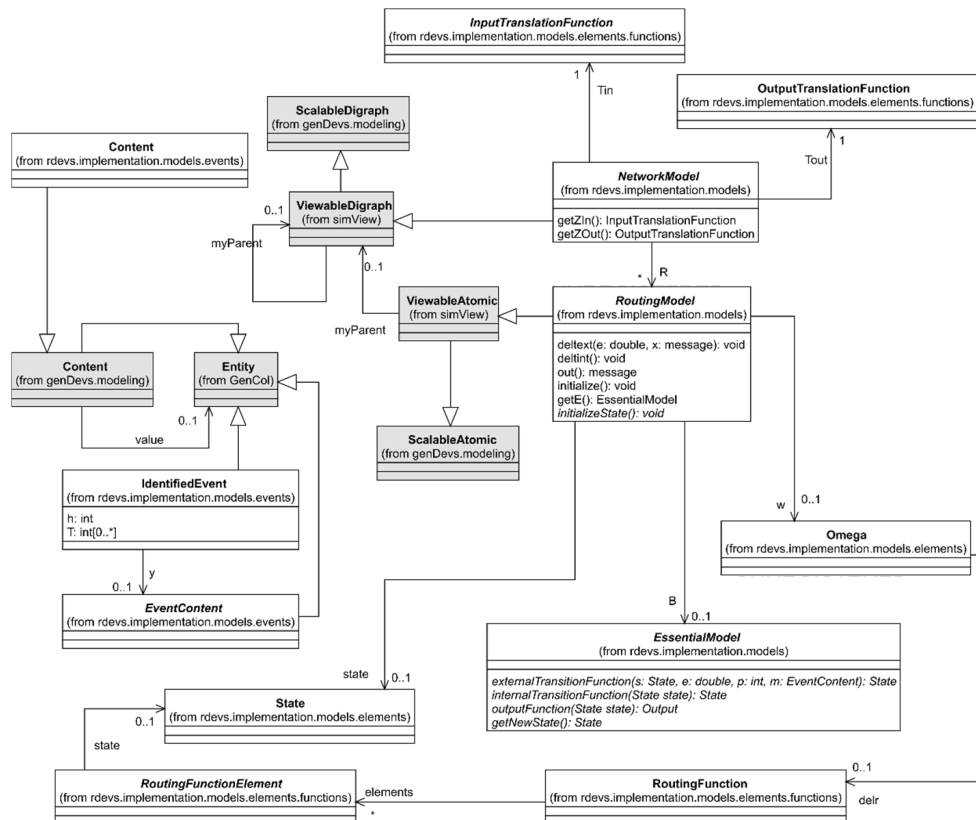


Figure 1. UML class diagram of the Java classes included in the RDEVSN library. Classes highlighted in gray belong to the DEVSNJAVA package.

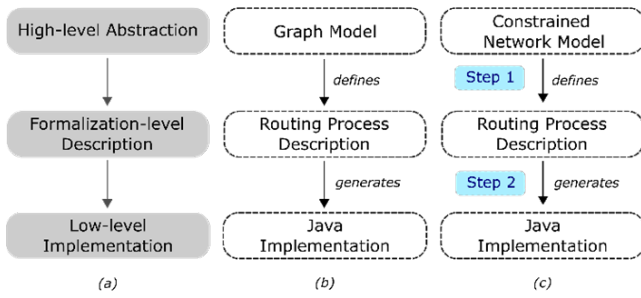


Figure 2. (a) Independent-implementation two-level modeling approach. (b) How the two-level modeling approach was used for graph model abstractions [Blas and Gonnet, 2021]. (c) How the two-level modeling approach is used in this paper for constrained network model abstractions. Step 1 was presented in Blas *et al.* [2021] and is described briefly in Section 3.1. Step 2 is the contribution of this work.

Figure 2(a) shows the independent-implementation modeling approach used in this paper. As the figure details, the high-level description is supported by the formalization-level description (i.e., a *routing process* specification) to define the two-level modeling strategy. Then, the routing process specification can be translated into a low-level implementation based on Java.

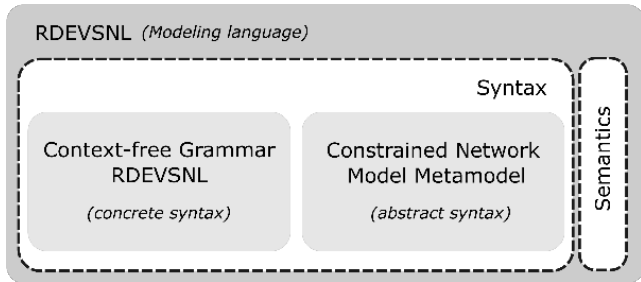


Figure 3. Implementation of the RDEVSNL modeling language using a syntax (composed of an abstract syntax and a concrete syntax) and semantics. To simplify, the CFG and the modeling language receive the same name. When we refer to the CFG, we mean the textual definition through the concrete syntax.

Following this approach in Blas and Gonnet [2021], we abstract the *routing process* definition into a high-level abstraction designed as a graph model based on nodes and edges (Figure 2(b)). Instead of using a modeling language, we define a graphical representation based on *i)* two types of nodes (one for the component behavior and the other for the component routing functionality) and *ii)* two types of links (one for the routing path and the other for linking the component behavior with its routing functionality). The graph representation is detailed in a metamodel that supports the instantiation of valid *routing processes*. Hence, the abstractions are mapped to the roles presented in **Table 1** to define RDEVSNL equivalences through the modeling levels. Then, we present a plug-in for Eclipse [The Eclipse Foundation, 2022b] that supports the graphical modeling of routing situations (through metamodel instantiation) as the core specification of RDEVSNL models. The simulation models obtained from the *routing process* definition were deployed as Java classes that extend the RDEVSNL Library. These classes together define the computational model attached to the abstraction defined through the graph. In this way, modelers can have RDEVSNL models without codifying any routing implementation (i.e., they only must define a routing situation graphically as a set of nodes connected by links).

Network theory is a useful technique to model relationships between entities [Newman *et al.*, 2011]. A network consists of nodes connected by a set of links. Such a representation has been widely adopted for modeling studies in several fields. For example, in the social network domain, it was used by Borgatti and Halgin [2011]. In software engineering, network theory was used to evaluate systems/software issues [Pan, 2011; Wen *et al.*, 2007; Zakari *et al.*, 2018].

Aiming to continue using abstractions to define *routing processes*, we developed a textual representation based on network theory [Blas *et al.*, 2021]. The textual representation was designed as a CFG used to define constrained network models through a metamodel template. This previous

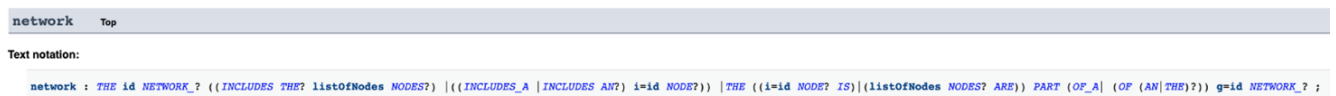


Figure 4. Production rule of the nonterminal symbol “network”.

Visual notation:

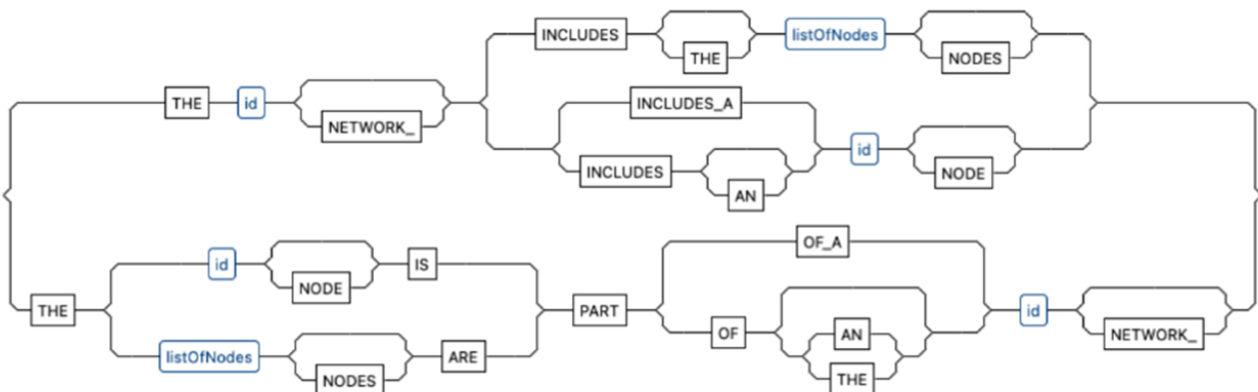


Figure 5. Syntax diagram of the nonterminal symbol “network”.

Table 2. Examples of syntactical expressions allowed in the context-free grammar RDEVSNL.

Primary Building Block	Allowed Sentences (English version)	Allowed Sentences (Spanish version)
Network	<ul style="list-style-type: none"> •The A node is part of C network. •The C network includes B node. •The A and B nodes are part of a C network. •The A node performs the behavior of D component. 	<ul style="list-style-type: none"> •El nodo A es parte de la red C. •La red C incluye al nodo B. •Los nodos A y B son parte de la red C. •El nodo A ejecuta el comportamiento del componente D.
Materialize	<ul style="list-style-type: none"> •The B node materializes D component. •The D component defines the behavior of A and B nodes. •The A node sends outputs to B node. 	<ul style="list-style-type: none"> •El nodo B materializa al componente D. •El componente D define el comportamiento de los nodos A y B.
Edges	<ul style="list-style-type: none"> •The E and F nodes receive inputs from the B node. •The connections are: A with B, B with E and B with F. 	<ul style="list-style-type: none"> •El nodo A envía salidas al nodo B. •Los nodos E y F reciben entradas desde el nodo B. •Las conexiones son A con B, B con E y B con F.

work is described briefly in Section 3.1. Following the two-level modeling approach, this paper uses such a constrained network model template as a high-level abstraction of *routing processes* to get RDEVSNL computational models (Figure 2(c)). The constrained network model defined over a textual specification is translated to a set of Java classes designed following the RDEVSNL Library (Figure 1) using the transformation detailed in Section 4.

3.1 Defining Routing Processes based on Constrained Network Models

We defined a new modeling language named RDEVSNL to specify a routing process using a constrained network model abstraction (i.e., to perform step 1 of the independent-implementation approach described above). The RDEVSNL was named “RDEVSNL Natural Language” in similarity with “DEVSNL Natural Language” (DEVSNL) [Zeigler and Nutaro, 2016].

DEVSNL is defined as a “constrained natural language specification of DEVS models”. Indeed, it is defined as an Extended Backus-Naur Form (EBNF), a meta-syntax notation for CFGs. A CFG is defined as a set of recursive rules used to describe a context-free language. Hence, due to the CFG definition, the processing of the textual expressions defined in DEVSNL does not involve any natural language reasoning (i.e., it does not involve dealing with, for example, natural language ambiguity). It only requires the text specification to satisfy the defined rules (without considering any context between them). As we will describe later, RDEVSNL is also based on a CFG. Hence, natural language ambiguity is not an issue to be solved.

A modeling language is commonly defined using three components: *abstract syntax*, *concrete syntax*, and *semantics* Krahn et al. [2007]. The *abstract syntax* identifies modeling concepts materialized in concrete syntaxes (that are frequently specified using visual/textual elements). Usually, *concrete* and *abstract syntaxes* are developed first, and then the *semantics* is designed to define the meaning of the language [Harel and Rumpe, 2004]. In this way, *semantics* de-

finer the meaning of the *abstract syntax* in terms of concepts already well-defined and well-understood in the semantic domain. Figure 3 depicts how the RDEVSNL language has been designed (i.e., how each component was developed).

As the figure shows, the *concrete syntax* is defined through the CFG named RDEVSNL (as the overall language). Since metamodelling is a popular method to define the *abstract syntax* of languages [Sprinkle et al., 2007], we use a conceptualization of a routing process through a metamodel that structures constrained network models. We consider that when modeling a routing process as a network, nodes define components, and links denote interactions between them. The *semantics* of the language is out of the scope of this paper.

3.1.1 The RDEVSNL Context-Free Grammar for Constrained Network Models

The CFG named RDEVSNL was presented in Blas et al. [2021]. It is based on a constrained network model to approach the definition of routing processes. It abstracts the definition of this type of model into textual representations using nodes and links to describe their structure. That is, it allows describing a routing scenario using notions of network theory (i.e., nodes and links as structural components of a network). Such a definition allows for building a textual specification of a constrained network model that can be verified to ensure syntactical correctness.

Two distinct versions of the syntax were developed, one in English and the other in Spanish. Then, modelers can choose to define the constrained network model according to their preference. Both grammars were specified and implemented using ANTLR4 [Parr, 2022].

In the English specification, three primary building blocks can be identified *i)* network, *ii)* materializes, and *iii)* edges. Network is the main block used to define a *constrained network model*. As an example, Figures 4 and 5 show, respectively, the production rule and syntax diagram of this symbol.

When a RDEVSNL specification is used to structure a routing process, a network is defined to model the overall constrained network model. Such a network model is defined

over a set of nodes. Each node denotes a routing component. Therefore, a network is specified using a name and it always includes a list of nodes. The sentences from the network block (Figures 4 and 5) allow us to define an identifier and describe the list of nodes that are part of the network. This can be done in a unique specification (e.g., “*the A and B nodes are part of a C network*”) or in multiple text lines (e.g., “*the A node is part of the C network*”; “*the C network includes the B node*”).

In a routing process, each component exhibits an internal operation and defines the behavior of a node or list of nodes. Sentences from the materializes block can be used to define the behavior that each node will execute, which is associated with the internal operation of a routing component (e.g., “*the D component defines the behavior of A and B nodes*”; “*the A node performs the behavior of D component*”).

Links define directed interactions between nodes. The grammar enables the definition of these interactions in multiple ways, using sentences from the edges block. For example, the modeler can use the following expressions: “*the A node sends outputs to the B node*” or “*the B and E nodes receive inputs from the A node*”.

Table 2 summarizes the main syntactical expressions included in RDEVSNL. For more details regarding the grammar design, refer to Blas *et al.* [2021].

The RDEVSNL Editor. As previously mentioned, the RDEVSNL grammar (i.e., both English and Spanish definitions) was implemented using ANTLR4. Based on such an implementation, a plug-in for the Eclipse platform was developed to allow modelers to use the grammar for defining constraints network models. Such an editor was presented along with the grammar in Blas *et al.* [2021].

The software tool is basically a text editor that allows specifying constrained network models using the RDEVSNL grammar and a “wizard” for creating files with the “*.rdevsln” extension (i.e., to store textual specifications). It provides writing aids during the edition (such as syntax highlighting and typing suggestions) using the language selected by the modeler.

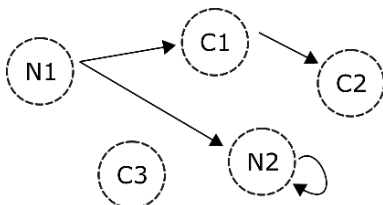


Figure 6. An example of a network model composed of five nodes. For the node labels, the letter identifies the component type.

Figure 7 shows an example of the network model defined in Figure 6. Such an example is defined in both languages (i.e., English and Spanish) to show how the grammar supports both types of specification for the same model. Later, we will discuss why this model is not defining a routing process (i.e., it is not structured as a correct constrained network model). Besides, in Section 4.1, we include a screenshot (Figure 12) with another English specification of a constrained network model that defines a correct routing process. Such an example is based on the one presented by Blas *et al.* [2021].

```

1 The N1,N2,C1,C2 and C3 are part of the M network
2 The N component defines the behavior of N1 and N2 nodes
3 The C component defines the behavior of C1, C2 and C3 nodes
4 The N1 node sends outputs to C1 and N2 nodes
5 The C2 node receive inputs from the C1 node
  
```

(a)

```

1 Los nodos N1, N2, C1, C2 y C3 son parte de la red M
2 El componente N define el comportamiento de los nodos N1 y N2
3 El componente C define el comportamiento de los nodos C1, C2 y C3
4 El nodo N1 envia salidas al nodo B
5 El nodo C2 recibe entradas desde el nodo C1
  
```

(b)

Figure 7. Specification of the network model defined in Figure 6 using the RDEVSNL Java editor developed. (a) The textual specification using the English grammar. (b) The textual specification using the Spanish grammar.

3.1.2 From Constrained Network Models to Routing Processes: Verifying the Network Model Structure

As previously stated, network theory proposes modeling a system as a set of nodes connected by links. Both nodes and links can have distinct meanings in a distinct context. When modeling a routing process, nodes define components, and links denote interactions between them. However, not just any network model defines a routing process.

By using RDEVSNL we can ensure a correct definition of a *network model* in which: 1. A network is composed of a set of nodes. 2. Nodes may (or may not) be connected between them using links. 3. Nodes may (or may not) have a behavior attached.

However, the CFG by itself is only useful to verify syntactical correctness. A textual specification can be syntactically correct but may not define a correct network model (*case 1*). Moreover, a textual specification can be syntactically correct and define a correct network model, but such a network model may not define a constrained network model that can be mapped to a routing process (*case 2*).

For case 1, take as an example the following definition:

“*The N1 and N2 nodes are part of the N network. The N1 node performs the behavior of the N2 component. The N2 and N3 nodes receive inputs from the N4 node.*”

The example follows all syntactical rules defined in Table 2 (i.e., it is syntactically correct for the CFG supporting RDEVSNL). Here, the network is defined as *N*. In sentence #1, two nodes are defined as components of the network identified as *N*: nodes *N1* and *N2*. However, if *N2* is the identifier of a node, then *N2* cannot be defined as the behavior of *N1* (this is the case of sentence #2). Moreover, if *N* is composed of *N1* and *N2*, then *N2* cannot be connected to *N3* and *N4* (as defined in sentence #3). Hence, even when the example is syntactically correct, the definition does not make sense.

For *case 2*, take as an example the definition of Figure 7(a). This example follows the syntactical rules defined in Table 2 and does not present the problem detailed for *case 1* (i.e., it is a correct network model). However, such a network model cannot be mapped to a routing process since it is not a correctly constrained network model (e.g., the node *C3* stands isolated from the others). To solve both issues (*cases 1* and *2*), we use a metamodel.

A metamodel is a model which defines the used language to design a model [OMG, 2002]. Due to their expressiveness, metamodels are powerful modeling tools to ensure the structural correctness of a model. They allow validating the model

instantiation regarding the set of rules defined at the meta-modeling level. Figure 8 presents a metamodel as a UML class diagram [OMG, 2017] that describes a routing process using the elements of a network model (detailed as stereotypes). This metamodel is an improved version of the metamodel defined in Blas *et al.* [2021] as an instance of the metamodel described to structure network models. This new version includes: *i)* the *NLSpecification* concept, *ii)* the behavior defined as a concept instead an attribute (i.e., the *Component Type*) and its associations, and *iii)* new Object Constraint Language (OCL) constraints (described below).

A specification (*NLSpecification*) includes a Routing Process. Each *Routing Process* is identified by a *routingProcessName*. The network model used to structure a *Routing Process* is defined over a set of nodes (*Node*), where each of them denotes a *Component* and executes the *behavior* of a *Component Type*. Each *Component Type* is identified by a *behaviorName*, while each *Component* is identified by a *nodeName*. The process is composed of a set of *Interactions* (*Link*) between *Components*. For each Interaction, a *Component* acts as a *source* (i.e., the node from which the interaction takes place) and another *Component* acts as a *destination* (i.e., the node to which the interaction is intended).

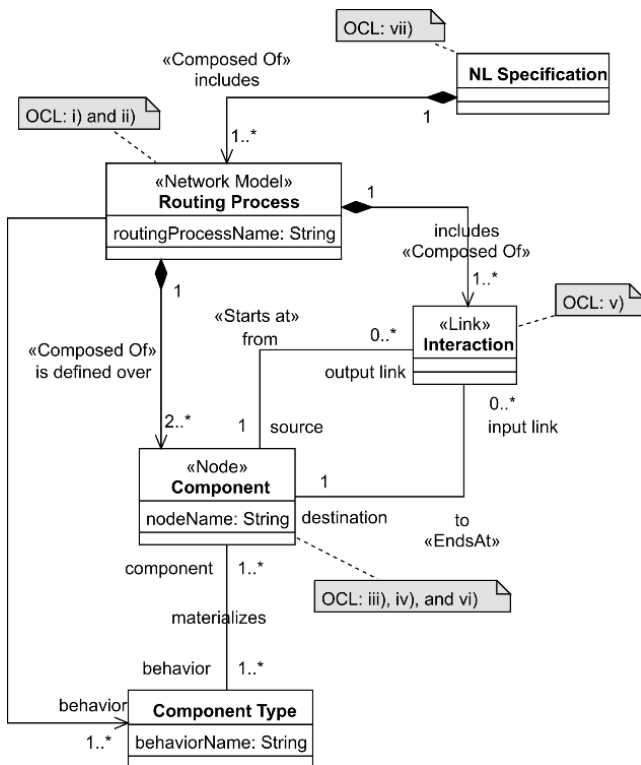


Figure 8. Metamodel used to structure the routing process definition from a constrained network model specification. Stereotypes are used to indicate the network model component to which the routing element refers. The notes highlighted in gray detail OCL constraints included in the model to obtain a routing process from a network model definition (see Table 3).

The metamodel of Figure 8 is restricted with OCL constraints [OMG, 2014] to ensure the network model defines a routing process. These constraints were designed to provide integrity to the metamodel as a template for defining properly network models that act as routing processes as follows: *i)* at least one node must be identified as initial, *ii)* at least one node must be identified as final, *iii)* nodes cannot be isolated,

iv) multiple links cannot connect the same pair of nodes, *v)* self-links are not allowed, *vi)* a component must execute a single behavior, and *vii)* a single routing process must be described in an NL specification. **Table 3** summarizes these constraints formally using OCL expressions. Constraints *i)* to *v)* were proposed originally by [Blas *et al.*, 2021]. Constraints *vi)* and *vii)* are introduced here as an improvement of the previous metamodel version.

When following these constraints, the network model presented in Figure 6 (and specified in Figure 7) cannot be used as a routing process specification since it does not satisfy the following rules: *i)* the node C3 is isolated - constraint *iii)*, and *ii)* the node N2 interacts with itself (i.e., it has a self-interaction) - constraint *v)*.

Table 3. OCL constraints (defined as invariants) used to get the “constrained network model”.

Id	OCL Constraint
i	context RoutingProcess invariant existsStartingComponent: self.component → select(c c.inputLink →size()=0 and c.outputLink → size() >0 → size() >0
	context RoutingProcess invariant existsEndingComponent: self.component → select(c c.inputLink → size() >0 and c.outputLink → size() = 0) → size() >0
iii	context Component invariant notIsolated:(self.inputLink → size() + self.outputLink → size()) >0
	context Component invariant multipleInteractions: self.outputLink → forall(e1,e2 e1 <> e2 implies e1.destination <> e2.destination)
v	context Interaction invariant notSelfInteraction: self.source < >self.destination
	context Component invariant singleBehavior: self.behavior → size() = 1
vii	context NLSpecification invariant singleSpecification: self.routingProcess → size() = 1

Including the Metamodel in the RDEVSNL Editor. To instantiate routing processes from constrained network models, the metamodel illustrated in Figure 8 was developed as an Ecore model using EMF [Eclipse Modeling Project, 2022]. EMF project is a modeling framework and code generation facility for building software tools and other software applications based on a structured data model. Hence, EMF allows getting a data model specification of the routing process definition for further instantiation and validation. Since this is Java technology, the Ecore model was embedded in the RDEVSNL Editor plug-in to provide an abstract syntax validation process.

When the validation option for a specification is activated, the RDEVSNL syntax analysis is executed over the current content of the “*.rdevsnl” file. Then, the parser tries to rec-

ognize the structures of sentences from a stream of tokens given by the actual specification. If the analysis is successful (i.e., all the sentences that the modeler used to create the specification are valid), using the tokens identified by the parser, an instance of the Ecore metamodel is automatically generated (i.e., the editor instantiates the defined metamodel following the parsing of the textual specification). Afterward, the metamodel's concepts, relationships, multiplicities, and OCL restrictions are verified over the obtained instance.

If no issues are found, the textual definition of the constrained network model is in correspondence with a valid routing process. This means that such a definition can be translated to Java code with the aim to obtain the RDEVS models attached (Section 4). On the contrary, if the textual definition has issues, the modeler will visualize an error message and a list with more details in the Problems view of Eclipse (having the possibility to fix its specification and check the new content).

4 Generating Java Implementations based on Routing Process Descriptions

The routing elements defined in the metamodel of Figure 8 can be related to the RDEVS modeling levels presented in **Table 1**. Such relations can be defined as follows:

- *Rel #1*) a *Routing Process* is attached to an RDEVS Network Model,
- *Rel #2*) a *Component* is attached to an RDEVS Routing Model, and
- *Rel #3*) a *Component Type* is attached to an RDEVS Essential Model.

How these metamodel elements are defined in terms of their properties and associations in an instantiation model determines how the RDEVS models should be designed. Take as an example the instance model defined in Figure 9. In this instantiation, a copying service composed of three devices is modeled as a routing process. Devices are two scanners (placed at offices 1 and 2) and one printer shared by both offices. Each *Scanner* (*Scanner Office 1* and *Scanner Office 2*) is modeled as an instance of *Component* type that *materializes* the *Scanner* defined as an instance of *Component Type*. For the printer shared by both offices, a *Component* instance is defined (i.e., *Shared Printer*) materializing the *Printer* defined as an instance of *Component Type*. Interactions are defined as follows: *i*) from *Scanner Office 1* to *Shared Printer*, and *ii*) from *Scanner Office 2* to *Shared Printer*. The overall routing process is defined through the *Copying Service* (instance of *Routing Process*). Such an instance is *defined over* *Scanner Office 1*, *Scanner Office 2*, and *Printer*.

Following the modeling levels presented in **Table 1** and the relations defined above, the RDEVS models required to simulate such a process are:

- Two RDEVS Essential Models for defining *Scanner* and *Printer*, respectively.

- Three RDEVS Routing Models for defining *Scanner Office 1*, *Scanner Office 2*, and *Shared Printer*, respectively.
- One RDEVS Network Model for defining the *Copying Service*.

Furthermore, the associations defined among instances define relationships between models as follows:

- The RDEVS Routing Models of *Scanner Office 1* and *Scanner Office 2* embed the RDEVS Essential Model of the *Scanner*.
- The RDEVS Routing Model of the *Shared Printer* embeds the RDEVS Essential Model of the *Printer*.
- The RDEVS Network Model of the *Copying Service* is composed of the RDEVS Routing Models of *Scanner Office 1*, *Scanner Office 2*, and *Shared Printer*.
- The Routing Policy of the RDEVS Routing Model of the *Scanner Office 1* is based on the *I1* instance of *Interaction*.
- The Routing Policy of the RDEVS Routing Model of the *Scanner Office 2* is based on the *I2* instance of *Interaction*.
- The Routing Policy of the RDEVS Routing Model of the *Shared Printer* is based on *I1* and *I2* instances of *Interaction*.

Hence, starting from an instantiation, a set of equivalences can be defined to get RDEVS simulation model implementations (i.e., the computational models). These equivalences were used to get the Java classes representing RDEVS models attached to textual specifications already validated by the plug-in as metamodel instances. To do this, we follow the extension points of the RDEVS library by creating new Java classes using Aceleo [The Eclipse Foundation, 2022a].

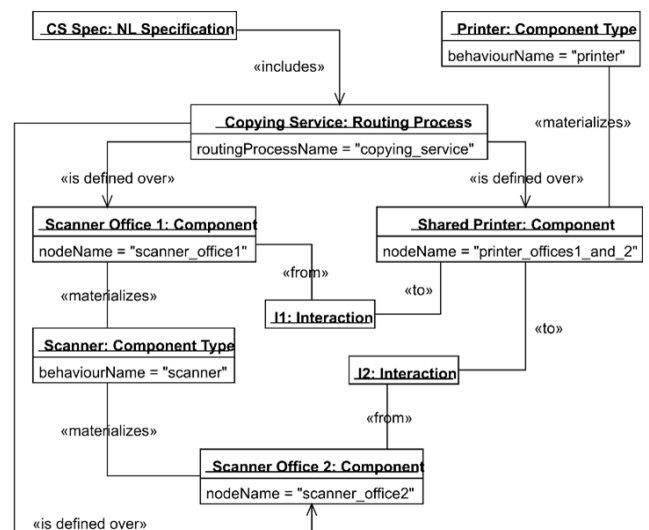


Figure 9. Example of a routing situation defined as a constrained network model obtained as an instance of the metamodel depicted in Figure 8.

Aceleo is a template-based technology that allows the creation of code generators from any data source available in EMF format. By defining a generation model for the text-to-code transformation, the elements defined in the abstraction model (i.e., the instance of the metamodel) are navigated to

```

1 [comment encoding = UTF-8 /]
2 [module generate('http://www.example.org/metamodel')]
3
4= [template public generateElement(nlSpecification : NLSpecification)]
5 [comment @main/]
6 [for (i : ComponentType | nlSpecification.routingProcess.behaviour)]
7 [generateComponentTypeStateClass(getComponentTypeStateClassName(i), i.behaviourName)/]
8 [generateComponentTypeEssentialModel(i, getComponentTypeClassName(i))/]
9 [//for]
10 [for (c: Component | nlSpecification.routingProcess.component)]
11 [registerComponent(c)/]
12 [registerRoutingModelList(c)/]
13 [registerEICList(c)/]
14 [registerEOCList(c)/]
15 [for (cl: Component | nlSpecification.routingProcess.component)]
16 [registerICList(c, cl)/]
17 [//for]
18 [//for]
19
20 [for (c: Component | nlSpecification.routingProcess.component)]
21 [for (b: ComponentType | c.behaviour)]
22 [generateRoutingFunctionClass(getRoutingFunctionClassName(c), c)/]
23 [generateRoutingFunctionElementClass(getRoutingFunctionElementClassName(c), getRoutingFunctionClassName(c), c.nodeName)/]
24 [generateNodeRoutingModel(c, getNodeClassName(c), b)/]
25 [//for]
26 [//for]
27
28 [for (r: RoutingProcess | nlSpecification.routingProcess)]
29 [generateRoutingProcessNetworkModel(r, getRoutingProcessClassName(r))/]
30 [generateInputTranslationFunctionForNetworkModel(getInputTranslationFunctionClassName(r), r.routingProcessName)/]
31 [//for]
32 [//template]

```

Figure 10. Definition of the process used to generate the Java code required to support the RDEVS models attached to a *NLSpecification* (i.e., the file named “generate.mtl” in Acceleo). Such a process requires other methods implemented as queries by combining Acceleo with Java. Main classes used to support the simulation models are defined in lines 8 (essential model), 24 (routing model), and 29 (network model).

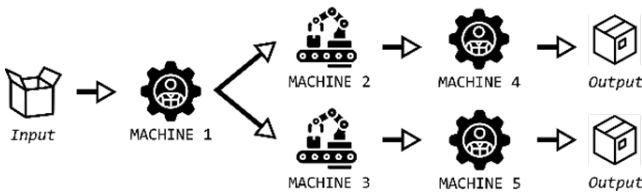


Figure 11. Industrial routing process proposed in Blas *et al.* [2021]. Icons depict machine types.

“write” the corresponding Java classes. The generation model used to get the RDEVS Java classes is presented in Figure 10. Such a model defines the main template for scrolling through the content of an *NLSpecification* (identified as *nlSpecification* in line 4).

It is important to note that the domain behavior (i.e., the first modeling level of **Table 1**) cannot be fully defined from the routing process definition. That is because the internal behavior of components is not defined in the constrained network models. Even so, following Rel #3), each *ComponentType* defined as part of the *nlSpecification.routingProcess.behaviour* set is used to structure an extension of the *EssentialModel* class (lines 6 to 9). The value of the *behaviourName* attribute is used to name the new subclass by adding “ComponentType” at the beginning. Also, a new subclass is defined to get an implementation of the model state as an extension of *State.java*. This new class uses the same name convention as the model by adding “State” before the *behaviourName*.

According to Rel #2), each Component included in the Routing Process should define an extension of the *RoutingModel.java* class. The set of *Components attached* to a *Routing Process* defined in the *nlSpecification* instance is obtained by the association *nlSpecification.routingProcess.component*. For each element included in such a set, subclasses required to support the attached *Routing Model* are created (lines 10 to 26).

Lines 10 to 18 are used to register all Components in the

process used to generate the Java code. Lines 20 to 26 define the extensions. These extensions are named using the value of the *nodeName* attribute adding an appropriate label to depict the part of the model defined (i.e., “Node”, “RoutingFunctionDefinition”, and “RoutingElement”). For example, the routing functionality of a Routing Model is defined through an extension of the *RoutingFunctionElement* class. This extension uses the output links to define available destinations. Accurate sources are defined considering the input links. In this way, Interactions and *Components* are used to define the routing behavior (i.e., the intermediate modeling level detailed in **Table 1**).

Finally, using the *Routing Process* definition and following Rel #1), an extension of the *NetworkModel* class is defined (lines 28 to 31). This class refers to the structure (i.e., the final modeling level of **Table 1**). The name of such an extension is defined using the value of the *routingProcessName* attribute by adding “Network” at the beginning. The extension is designed to include full couplings among instances of all classes used to implement routing models.

The “generator.mtl” file described above was included in the Acceleo project that supports the generation process of the Java code required for our implementation-independent modeling approach. Such a project was embedded in the Eclipse plug-in described in Section 3.1 to allow a direct translation when the metamodel validation is successfully verified. In this way, if both the CFG and metamodel produce a correct constrained network model instantiation, the Java code that implements the executable RDEVS simulation models attached to the network specified is automatically created.

4.1 Proof of Concepts

Figure 11 shows a routing scenario composed of two types of machines (which are denoted by the type of icon placed in the

scenario). The scenario involves five nodes used to connect inputs to outputs from two distinct paths. Then, Figure 12 presents a screenshot of an alternative textual specification of the scenario proposed in Figure 11 using the RDEVSNL Editor. Such a scenario is a valid constrained network model that can be translated to an RDEVS computational model as follows.

Take as an example line 1 of Figure 12. This line is syntactically correct according to **Table 2**. Therefore, during the RDEVSNL grammar checking, the plug-in creates the following elements:

- *i)* an instance of *RoutingProcess* with the attribute *routingProcessName* = “RoutingProcess”,
- *ii)* five instances of *Component* (each one with the label “Machine_1” to “Machine_5” in the attribute *nodeName*), and
- *iii)* five relationships *isDefinedOver* to link each *Component* to the *RoutingProcess*.

The same process is performed for the text specification detailed in lines 2 to 6 to define the instantiation of the metamodel (presented in Figure 8) attached to the routing situation (specified in Figure 12). Over the overall metamodel instantiation, the plug-in automatically runs the Ecore validation process to ensure its correctness. Such a validation checks concepts, relationships, multiplicities, and OCL constraints. In this case, the validation of the Ecore instance cre-

ated by the CFG is completed successfully. Then, the plug-in creates the RDEVS computational model following the process described in Section 4.

Figure 13 presents a screenshot of the Eclipse platform once the creation process is completed, and all Java classes have been created. In the Project Explorer (left side of the screen), all Java classes created by the M&S software tool are listed as part of the Java package named “rdevsmodels”. **Table 3** summarizes the Java classes created for each one of the elements instantiated from the metamodel. As the Table shows, several classes were created for the same metamodel element instantiated during the transformation (e.g., for the *Component* named “Machine_1”, three subclasses are created – *NodeMachine1.java*, *Machine_1RoutingFunctionDefinition.java*, and *Machine_1RoutingFunctionElement.java*).

The specification of the *NetworkRoutingProcess* class is presented on the right side of the screen (Figure 13). Such a class is defined as an extension of *NetworkModel* (line 13). As the yellow box shows, all classes automatically generated include documentation related to the metamodel element from which they are derived. Such documentation includes the authoring tag that indicates the Java code is auto-generated from the plug-in. It also includes the date on which the code was generated. Moreover, the green box highlights the model methods. In this case, the Java class includes only one method: the class constructor named *NetworkRouting-*

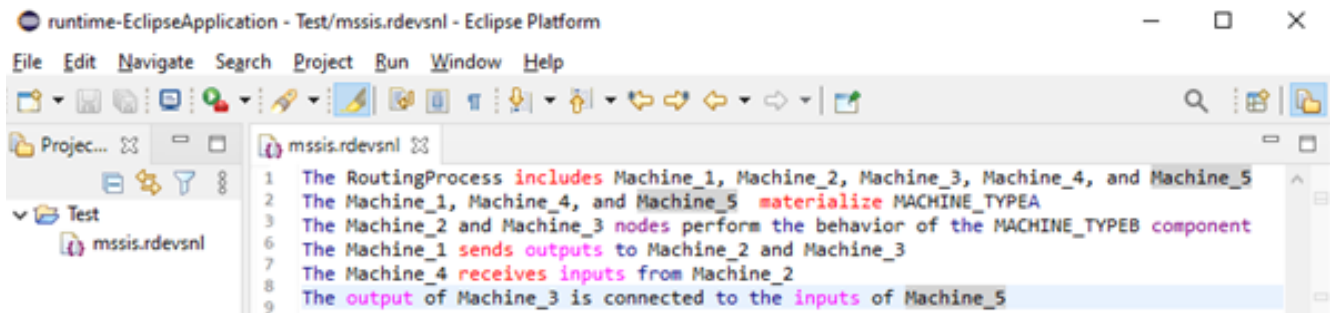


Figure 12. Screenshot of the RDEVSNL Editor when implementing the routing situation depicted in Figure 11. Suggestions are provided following the language configuration. Keywords are highlighted to help the modeler during the edition.

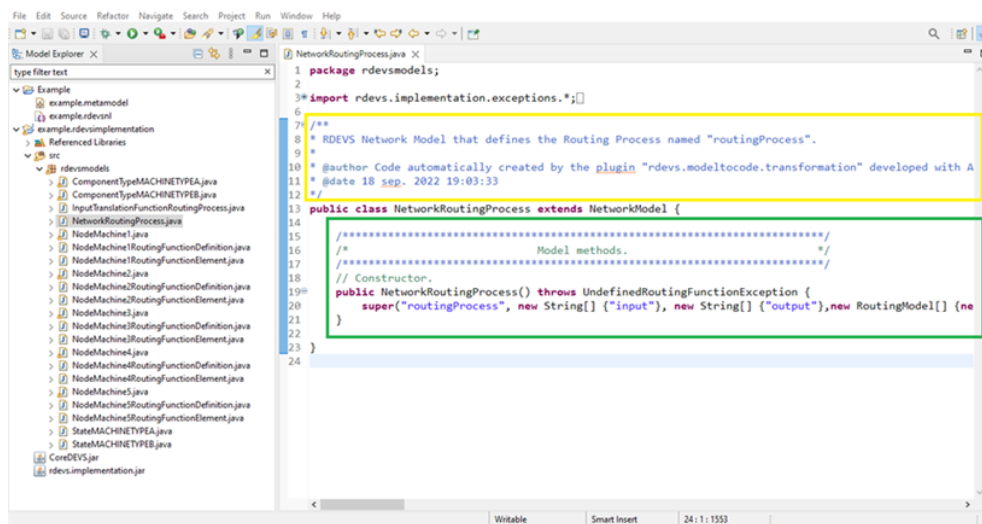
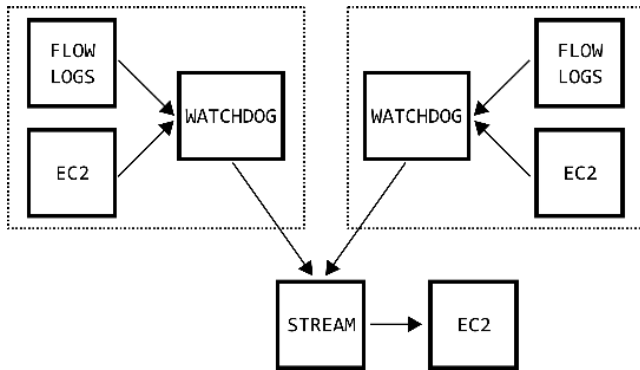


Figure 13. Screenshot of the Eclipse platform once the generation process developed has been executed for the textual specification detailed in Figure 12. Highlighted boxes refer to the documentation and model attributes definition of the new subclass extending the *NetworkModel* class.

Table 4. List of the Java classes obtained for the proof of concepts.

Metamodel Instance		RDEVS Java Implementation	
Name	Type	New class	“extends”
MACHINE_			
TYPEA	Component Type	ComponentTypeMACHINE_TYPEA.java StateMACHINE_TYPEA.java	EssentialModel.java State.java
MACHINE_			
TYPEB	Component Type	ComponentTypeMACHINE_TYPEB.java StateMACHINE_TYPEB.java	EssentialModel.java State.java
Machine_1	Component	NodeMachine_1.java Machine_1RoutingFunctionDefinition.java Machine_1RoutingFunctionElement.java	RoutingModel.java RoutingFunction.java RoutingFunctionElement.java
Machine_2	Component	NodeMachine_2.java Machine_2RoutingFunctionDefinition.java Machine_2RoutingFunctionElement.java	RoutingModel.java RoutingFunction.java RoutingFunctionElement.java
Machine_3	Component	NodeMachine_3.java Machine_3RoutingFunctionDefinition.java Machine_3RoutingFunctionElement.java	RoutingModel.java RoutingFunction.java RoutingFunctionElement.java
Machine_4	Component	NodeMachine_4.java Machine_4RoutingFunctionDefinition.java Machine_4RoutingFunctionElement.java	RoutingModel.java RoutingFunction.java RoutingFunctionElement.java
Machine_5	Component	NodeMachine_5.java Machine_5RoutingFunctionDefinition.java Machine_5RoutingFunctionElement.java	RoutingModel.java RoutingFunction.java RoutingFunctionElement.java
Routing_Process	Network Model	NetworkRouting_Process.java InputTranslation FunctionRouting_Process.java OutputTranslation FunctionRouting_Process.java	NetworkModel.java InputTranslationFunction.java OutputTranslationFunction.java



```

1 //network
2 The AWSNetflix network includes ec21, ec22 and ec23 nodes
3 The AWSNetflix network includes flogs1 and flogs2 nodes
4 The AWSNetflix network includes watchdog1 and watchdog2 nodes
5 The stream1 node is part of AWSNetflix network
6 //materializes
7 The ec21, ec22 and ec23 nodes materialize EC2 component
8 The Watchdog component is used in watchdog1 and watchdog2 nodes
9 The flogs1 and flogs2 nodes perform the behavior of Flowlogs component
10 stream1 performs the behavior of Stream component
11 //edges
12 The connections are: flogs1 with watchdog1 and ec21 with watchdog1
13 The connections are: flogs2 with watchdog2 and ec22 with watchdog2
14 The output of watchdog1 is connected to the inputs of stream1
15 The output of watchdog2 is connected to the inputs of stream1
16 The stream1 node sends outputs to ec23
  
```

Figure 14. (a) A high-level description of the web architecture proposed in the case study “AWS Netflix”. (b) The RDEVSNL specification of the architecture depicted in Figure 14(a).

Process(). All the parameters required to set up the structure of the network model are created by calling the *super()* constructor method (i.e., the constructor defined in the *NetworkModel* class). By using instances of all routing model classes created from other metamodel elements, the param-

eters of the *super()* method are defined. In this way, the network model used to simulate the routing process defined in the metamodel is structured in terms of all nodes (i.e., all RDEVS routing models) already implemented as Java code.

Furthermore, a video of the software tool operation can be seen here ¹.

4.2 Using the Tool for Modeling Real-Life Scenarios

To show how the overall software tool supports the M&S development process for real-life cases, we present two case studies already analyzed (from the modeling effort point of view) with the RDEVS formalism in Blas *et al.* [2022]. Such cases refer to real-life software architectures obtained from Amazon Web Services.

The first case study specified is the Multi-Region Resiliency of Netflix implemented with Amazon Route 53 (Amazon Web Services [2021a]). A sketch of such an architecture is presented in Figure 14(a). Following the textual specification purposed, Figure 14(b) shows the RDEVSNL definition. Here, each replica of a component is defined as a node (composing the network) that performs the component behavior. For example, line 4 details the *AWSNetflix* network is composed of two replicas of WATCHDOG named *watchdog1* and *watchdog2*. Then, line 8 details that both repli-

¹https://www.youtube.com/watch?v=4RbHn4AqRWM&ab_channel=RoutedDEVSGroup

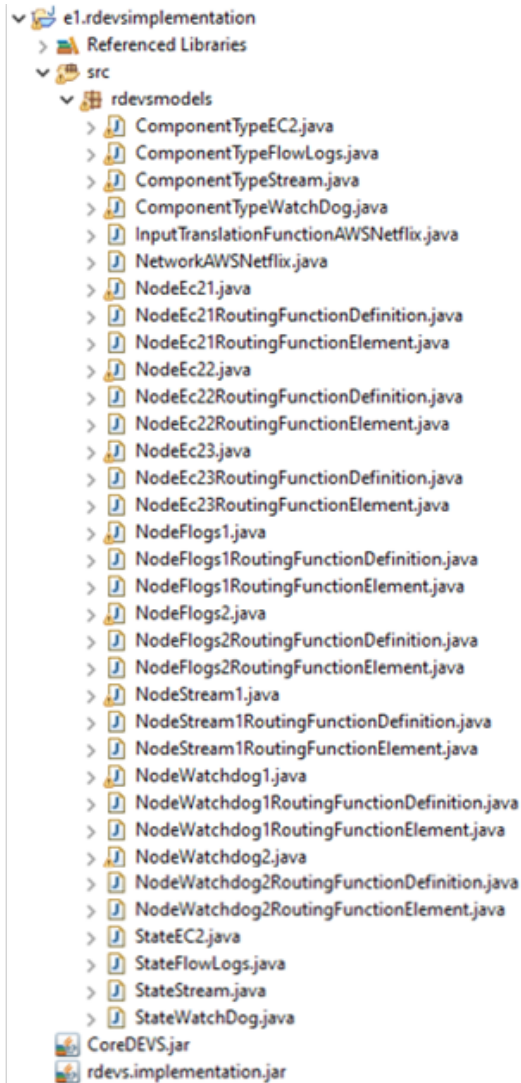


Figure 15. List of Java classes created when the translation process is executed over the definition detailed in Figure 14(b).

cas use the component named *WatchDog*. Finally, Figure 15 shows how the project explorer looks when the translation process is finished. As this last figure shows, a Java class is created for each element defined in the specification. All these classes together provide the structure required to perform an RDEVS simulation.

A larger case is the Deals Engine Architecture of Expedia Group Global (Amazon Web Services [2021b]), outlined in Figure 17. Following the development process used in our tool, Figure 18 shows both the textual specification (on the right side of the screen) and the set of Java classes obtained once the translation process is successfully performed (at the project explorer shown on the left side of the screen). As a complement, Figure 18 shows part of the metamodel instantiation produced to support such a translation. As Figures 16 and 17 show, mechanisms used to support the software tool proposed in this paper scale well when different numbers of nodes, components, and links are used. Moreover, the generated Java code is sound requiring the modeler to complete the behavioral definitions before executing the actual simulation.

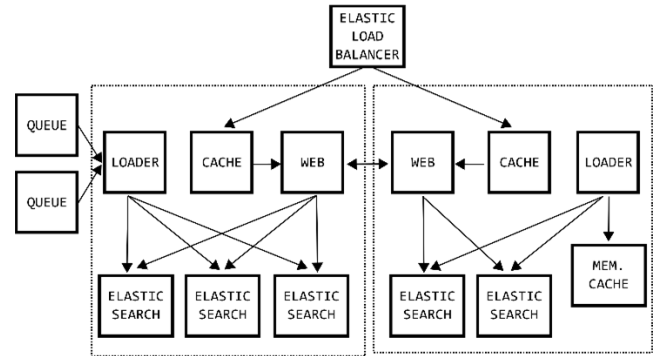


Figure 16. A high-level description of the web architecture proposed in the case study “AWS Expedia Group”.

5 Discussion

The RDEVSNL Editor has been proposed as an alternative modeling tool for building RDEVS simulation models. With the code generation addition, the eclipse plug-in defined initially as an RDEVSNL Editor creates the Java classes for the RDEVS implementation of simulation models designed using a constrained network model defined textually.

Even when building a plug-in can seem to be a high cost for a research project, the benefits obtained are higher. The advantages of our proposal are *i)* reduction of implementation times through fast modeling solutions and *ii)* simulation model correctness regarding the formalism through well-defined and standardized simulation models.

We promote the use of textual specifications as an alternative to graphical representations for two reasons:

1. Textual specifications following a CFG are always faster to develop than graphical definitions. Since we are working with a restricted set of sentences that can be combined in several ways, the modeler can define a “dense” constrained network model (i.e., a model with a lot of nodes and connections among them) using just a few lines. Such a specification is less time consuming than, for example, graphical models.
2. For large routing scenarios, graphical models tend not to be useful due to their size. Instead, as described before, textual specifications allow detail-ing a large set of definitions using just a few sentences.

We are now concluding the testing of the transformation process. All the cases conducted have been successfully translated into RDEVS computational models when valid routing processes are defined through constrained network models. We have tested textual specifications of both small (until ten nodes in a single network) and large (until 50 nodes in a single network) models. The time response of all cases has been acceptable concerning the number of Java classes autogenerated by the plug-in. As the proposal is defined as a modeling tool, the simulation execution analysis is outside the scope of the paper. Such an analysis concerns how RDEVS simulation models are implemented in Java and how DEVJSJAVA supports their execution.

It is important to note that, as explained before, the specification of the abstract operations to be redefined at the subclass level for Essential Models cannot be derived from the instance definition. Such a behavioral specification is part

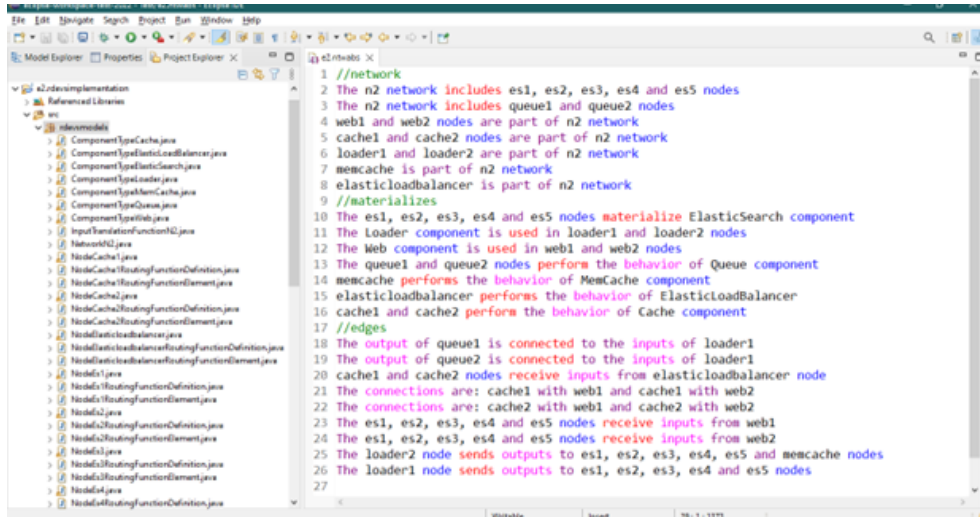


Figure 17. How the “AWS Expedia Group” case study (structured as the web architecture presented in Figure 16) can be defined using our software tool to get the related RDEVS models.

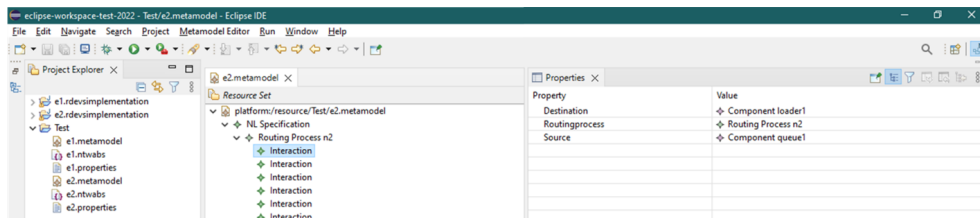


Figure 18. Part of the metamodel instantiated for the case study presented in Figure 17. The XML version of the complete metamodel can be seen at Appendix A.

of the domain problem and is not included in the constrained network model specification. Then, the modeler should detail these operations using Java code or another type of DEVS modeling specification. Since essential models are defined as DEVS atomic models, DEVS modeling tools can be used to achieve the domain behavior specification. For example, in Dalmaso *et al.* [2023], the authors present a modeling software tool that employs enriched UML statecharts for the graphical representation of the domain behavior modeling level (exhibit in Table 1). Once the behavioral operations are defined, their specification should be included in existing Java classes by writing their code in the appropriate methods.

The main limitation is the use of Java as support for RDEVS implementations. Until there are new implementations of RDEVS in other programming languages, such a limitation cannot be overcome. Still, if new RDEVS Libraries are developed, only new translation processes should be defined. The RDEVSNL grammar and the editor can remain the same to support the textual specifications of constrained network models.

The implementation-independent modeling strategy used to build the plug-in allows for providing a more suitable tool than other software tools currently available. Since it uses a general abstraction model (i.e., a network model) as the core definition of the RDEVS simulation models, it allows the modeler to obtain computational models without the need to codify the implementation in Java. Moreover, it provides a more accurate representation of the problem to modelers because they can work with a new level of abstraction. That is the main difference with other approaches like DEVS Modeling Language (DEVSML) and DEVSNL (mentioned in Sec-

tion 3.1).

DEVSML [Mittal and Douglass, 2012] provides a platform-independent way to specify DEVS models that are transformed to platform-specific language implementation in Java, C++, or any other programming language. On the other hand, DEVSNL [Zeigler and Sarjoughian, 2017] provides a natural language specification to understand FDDEVS (Finite Deterministic DEVS) models. These models can be used to automatically generate DEVS atomic models in Java that have full capability to express messages and states.

In both cases (i.e., DEVSML and DEVSNL), the modeler needs to understand how DEVS models are structured to build specifications. Instead, in our case, the modeler is abstracted from the notions of RDEVS formalism and generates an (abstract) constrained network model to represent a problem (i.e., a routing scenario). Such an abstract model is used to create the related simulation models in Java. The separation of concerns between the abstraction model and the programming language used to support the simulation model implementations allows a further mapping to other programming languages. That is the main benefit of using our implementation-independent modeling strategy.

5.1 Threats to Validity

Given the nature of our proposal, adopting RDEVS models to test our modeling approach can be seen as a threat to validity. Due to the novelty of RDEVS, no public repository for RDEVS models is available. Nevertheless, it was possible to reproduce all those RDEVS examples reported in the literature (Blas and Gonnet [2021]; Blas *et al.* [2017]; Blas *et al.*

[2022]) and check whether our independent-implementation modeling approach is valid to generate well-defined RDEVS simulation models. Consequently, we had to specify the routing process detailed in the original RDEVS models as constrained network models. Then, we use a textual specification to define constrained network models that were translated into a set of Java classes designed following the structure of RDEVS Library. To overcome the threat, we made it evident that original RDEVS models can be correctly abstracted as constrained network models and can be transformed into a Java implementation of the RDEVS simulation model. Finally, additional scalability tests should be performed to analyze the space and time overhead introduced during the model transformation phase.

6 Conclusions and Future Work

The RDEVS formalism provides a formal definition for the M&S of general routing processes employing the “embedding routing functionality” strategy over DEVS models. In this paper, we have presented an implementation-independent modeling approach to build a plug-in in development intended to obtain Java implementations of RDEVS models from an abstract model defined in a textual specification. For the textual representation, we propose a context-free grammar named RDEVSNL which is based on a constrained network model. Such grammar has been implemented using ANTLR4. A metamodel is used to map the textual definition with routing processes. This metamodel allows a direct mapping between its concepts and RDEVS simulation models. Then, Java classes are derived using Acceleo. The RDEVS library is used as support since it enhances the development of RDEVS models in Java using some features provided by DEVSJAVA.

Our proposal is part of a work-in-progress intended to develop M&S software tools for the RDEVS formalism as a discrete-event specification for routing processes. Our final aim is to provide a tool that allows modelers to *i)* define the problem domain using well-known abstractions and *ii)* get the computational models attached to such abstraction models through conceptual mapping. Then, modelers will be able to have simulation models without needing to codify any routing implementation. Moreover, they can get simulation models without having programming skills. The plug-in presented in this paper is a fundamental part of such research as an additional feature of the graphical specifications already defined by Blas and Gonnet [2021].

Future work is devoted to the development of new representations for large-scale routing processes as new abstraction models to be used as additional features for M&S based on RDEVS. Moreover, since the RDEVSNL is still not a language (because the semantics definition is missing), we plan to define such a component to provide a full modeling language (by completing the set of elements described in Figure 3).

Declarations

Funding

This work was supported by UTN [Grant Number SIT-CBFE0008464TC].

Authors' Contributions

Clarisa Espertino performed the development of the M&S software tool including the conceptual modeling and mapping with RDEVS implementations. Maria Julia Blas contributed to the conception of this study and checked the correctness of the computational models obtained regarding the RDEVS formalism. Espertino and Blas are the main contributors and writers of this manuscript. Silvio Gonnet supervised the research project. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this manuscript.

Availability of data and materials

Data can be made available upon request.

References

- Alshareef, A., Blas, M. J., Bonaventura, M., Paris, T., Yacoub, A., and Zeigler, B. P. (2022). Using devts 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*, pages 215–266. Springer. DOI: 10.1007/978-3-030-87049-2₈.
- Amazon Web Services (2021a). Aws expedia group. Available at: <https://aws.amazon.com/pt/solutions/case-studies/expedia/> Accessed: June 2021.
- Amazon Web Services (2021b). Aws netflix case study. Available at: <https://aws.amazon.com/pt/solutions/case-studies/innovators/netflix/> Accessed: June 2021.
- Barros, F. J. (1997). Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4):501–515. DOI: 10.1145/268403.268423.
- Blas, M. J., Espertino, C., and Gonnet, S. (2021). Modeling routing processes through network theory: A grammar to define rdevs simulation models. In *Anais do III Workshop em Modelagem e Simulação de Sistemas Intensivos em Software*, pages 10–19. SBC. DOI: 10.5753/mssis.2021.17255.
- 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., Gonnet, S., and Leone, H. (2017). Routing structure over discrete event system specification: a devs adaptation to develop smart routing in simulation models. In *2017 Winter simulation conference (WSC)*, pages 774–785. IEEE. DOI: 10.1109/WSC.2017.8247831.
- Blas, M. J., Gonnet, S. M., Leone, H. P., and Zeigler, B. P. (2018). A conceptual framework to classify the extensions of devs formalism as variants and subclasses. In *2018 Winter Simulation Conference (WSC)*, pages 560–571. IEEE. DOI: 10.1109/WSC.2018.8632265.
- 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.
- Borgatti, S. P. and Halgin, D. (2011). On network theory. organization science. *Articles in Advance*, pages 1–14. DOI: 10.1287/orsc.1100.0641.
- Dalmasso, F., Blas, M. J., and Gonnet, S. (2023). Enriching uml statecharts through a metamodel: A model driven approach for the graphical definition of devs atomic models. *IEEE Latin America Transactions*, 21(1):27–34. DOI: 10.1109/TLA.2023.10015142.
- Eclipse Modeling Project (2022). Eclipse modeling framework. Available at: <https://www.eclipse.org/modeling/emf/>.
- Gehlot, V. and Nigro, C. (2010). An introduction to systems modeling and simulation with colored petri nets. In *Proceedings of the 2010 winter simulation conference*, pages 104–118. IEEE. DOI: 10.1109/WSC.2010.5679170.
- Harel, D. and Rumpe, B. (2004). Meaningful modeling: What’s the semantics of “semantics”? *Computer*, 37(10):64–72. DOI: 10.1109/MC.2004.172.
- Kahraman, C. and Tüysüz, F. (2010). Manufacturing system modeling using petri nets. In *Production Engineering and Management under Fuzziness*, pages 95–124. Springer. DOI: 10.1007/978-3-642-12052-7_6.
- Klatt, B. and Krogmann, K. (2008). Software extension mechanisms. *Fakultt fr Informatik, Karlsruhe, Germany, Interner Bericht*, 8:2008. Available at: <https://sdq.kastel.kit.edu/publications/pdfs/klatt2008a.pdf>.
- Krahn, H., Rumpe, B., and Völkel, S. (2007). Integrated definition of abstract and concrete syntax for textual languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 286–300. Springer. DOI: 10.1007/978-3-540-75209-7_20.
- Mittal, S. and Douglass, S. A. (2012). Devsml 2.0: the language and the stack. *SpringSim (TMS-DEVS)*, 17. Available at: <https://apps.dtic.mil/sti/citations/ADA556733>.
- Newman, M., Barabási, A.-L., and Watts, D. J. (2011). *The structure and dynamics of networks*. Princeton university press. DOI: 10.1515/9781400841356.
- OMG (2002). Meta object facility (mof) specification, version 1.4. Available at: <https://www.omg.org/spec/MOF/1.4/About-MOF>.
- OMG (2014). Object constraint language specification, version 2.4. Available at: <https://www.omg.org/spec/OCL/2.4>.
- OMG (2017). Unified modeling language, version 2.5.1. Available at: <https://www.omg.org/spec/UML/2.5.1>.
- Pan, W. (2011). Applying complex network theory to software structure analysis. *International Journal of Computer and Systems Engineering*, 5(12):1634–1640. DOI: 10.5281/zenodo.1332884.
- Parr, T. (2022). Antlr. Available at: <https://www.antlr.org/>.
- Parsons, J. and Wand, Y. (1997). Choosing classes in conceptual modeling. *Communications of the ACM*, 40(6):63–69. DOI: 10.1145/255656.255700.
- 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://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6edc00b4669d73aa3cc94de431ed12951f0e5b7f>.
- Sprinkle, J., Rumpe, B., Vangheluwe, H., and Karsai, G. (2007). 3 metamodelling: State of the art and research challenges. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76. Springer. DOI: 10.1007/978-3-642-16277-0.
- The Eclipse Foundation (2022a). Acceleo. Available at: <https://www.eclipse.org/acceleo/>.
- The Eclipse Foundation (2022b). Eclipse. Available at: <https://www.eclipse.org/>.
- Wen, L., Kirk, D., and Dromey, R. G. (2007). Software systems as complex networks. In *6th IEEE International Conference on Cognitive Informatics*, pages 106–115. IEEE. DOI: 10.1109/COGINF.2007.4341879.
- Zakari, A., Lee, S. P., and Chong, C. Y. (2018). Simultaneous localization of software faults based on complex network theory. *IEEE Access*, 6:23990–24002. DOI: 10.1109/ACCESS.2018.2829541.
- Zeigler, B. and Sarjoughian, H. S. (2017). Devs natural language models and elaborations. *Guide to Modeling and Simulation of Systems of Systems*, pages 43–69. DOI: 10.1007/978-3-319-64134-8_4.
- 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.

A Appendix A

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <metamodel:NLSpecification xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:metamodel="
    http://www.example.org/metamodel">
3  <routingProcess routingProcessName="n2">
4  <interaction destination="//@routingProcess.0/@component.11" source="//@routingProcess.0/@component
    .5"/>
5  <interaction destination="//@routingProcess.0/@component.11" source="//@routingProcess.0/@component
    .6"/>
6  <interaction destination="//@routingProcess.0/@component.9" source="//@routingProcess.0/@component
    .14"/>
7  <interaction destination="//@routingProcess.0/@component.10" source="//@routingProcess.0/@component
    .14"/>
8  <interaction destination="//@routingProcess.0/@component.7" source="//@routingProcess.0/@component.9
    "/>
9  <interaction destination="//@routingProcess.0/@component.8" source="//@routingProcess.0/@component.9
    "/>
10 <interaction destination="//@routingProcess.0/@component.7" source="//@routingProcess.0/@component
    .10"/>
11 <interaction destination="//@routingProcess.0/@component.8" source="//@routingProcess.0/@component
    .10"/>
12 <interaction destination="//@routingProcess.0/@component.0" source="//@routingProcess.0/@component.7
    "/>
13 <interaction destination="//@routingProcess.0/@component.1" source="//@routingProcess.0/@component.7
    "/>
14 <interaction destination="//@routingProcess.0/@component.2" source="//@routingProcess.0/@component.7
    "/>
15 <interaction destination="//@routingProcess.0/@component.3" source="//@routingProcess.0/@component.7
    "/>
16 <interaction destination="//@routingProcess.0/@component.4" source="//@routingProcess.0/@component.7
    "/>
17 <interaction destination="//@routingProcess.0/@component.0" source="//@routingProcess.0/@component.8
    "/>
18 <interaction destination="//@routingProcess.0/@component.1" source="//@routingProcess.0/@component.8
    "/>
19 <interaction destination="//@routingProcess.0/@component.2" source="//@routingProcess.0/@component.8
    "/>
20 <interaction destination="//@routingProcess.0/@component.3" source="//@routingProcess.0/@component.8
    "/>
21 <interaction destination="//@routingProcess.0/@component.4" source="//@routingProcess.0/@component.8
    "/>
22 <interaction destination="//@routingProcess.0/@component.0" source="//@routingProcess.0/@component
    .12"/>
23 <interaction destination="//@routingProcess.0/@component.1" source="//@routingProcess.0/@component
    .12"/>
24 <interaction destination="//@routingProcess.0/@component.2" source="//@routingProcess.0/@component
    .12"/>
25 <interaction destination="//@routingProcess.0/@component.3" source="//@routingProcess.0/@component
    .12"/>
26 <interaction destination="//@routingProcess.0/@component.4" source="//@routingProcess.0/@component
    .12"/>
27 <interaction destination="//@routingProcess.0/@component.13" source="//@routingProcess.0/@component
    .12"/>
28 <interaction destination="//@routingProcess.0/@component.0" source="//@routingProcess.0/@component
    .11"/>
29 <interaction destination="//@routingProcess.0/@component.1" source="//@routingProcess.0/@component
    .11"/>
30 <interaction destination="//@routingProcess.0/@component.2" source="//@routingProcess.0/@component
    .11"/>
31 <interaction destination="//@routingProcess.0/@component.3" source="//@routingProcess.0/@component
    .11"/>
32 <interaction destination="//@routingProcess.0/@component.4" source="//@routingProcess.0/@component
    .11"/>
33 <component inputLink="//@routingProcess.0/@interaction.8 //@routingProcess.0/@interaction.13 //
    @routingProcess.0/@interaction.18 //@routingProcess.0/@interaction.24" nodeName="es1" behavi-our
   ="//@routingProcess.0/@behaviour.0"/>
34 <component inputLink="//@routingProcess.0/@interaction.9 //@routingProcess.0/@interaction.14 //
    @routingProcess.0/@interaction.19 //@routingProcess.0/@interaction.25" nodeName="es2" behavi-our
   ="//@routingProcess.0/@behaviour.0"/>
35 <component inputLink="//@routingProcess.0/@interaction.10 //@routingProcess.0/@interaction.15 //
    @routingProcess.0/@interaction.20 //@routingProcess.0/@interaction.26" nodeName="es3" behavi-our
   ="//@routingProcess.0/@behaviour.0"/>
36 <component inputLink="//@routingProcess.0/@interaction.11 //@routingProcess.0/@interaction.16 //

```

```

    @routingProcess.0/@interaction.21 //@routingProcess.0/@interaction.27" nodeName="es4" behavi-our
    = "//@routingProcess.0/@behaviour.0"/>
37 <component inputLink="//@routingProcess.0/@interaction.12 //@routingProcess.0/@interaction.17 //
    @routingProcess.0/@interaction.22 //@routingProcess.0/@interaction.28" nodeName="es5" behavi-our
    = "//@routingProcess.0/@behaviour.0"/>
38 <component outputLink="//@routingProcess.0/@interaction.0" nodeName="queue1" behavi-our="//
    @routingProcess.0/@behaviour.3"/>
39 <component outputLink="//@routingProcess.0/@interaction.1" nodeName="queue2" behavi-our="//
    @routingProcess.0/@behaviour.3"/>
40 <component inputLink="//@routingProcess.0/@interaction.4 //@routingProcess.0/@interaction.6" output-
    Link="//@routingProcess.0/@interaction.8 //@routingProcess.0/@interaction.9 //@routingProcess.0/
    @interaction.10 //@routingProcess.0/@interaction.11 //@routingProcess.0/@interaction.12"
    nodeName="web1" behaviour="//@routingProcess.0/@behaviour.2"/>
41 <component inputLink="//@routingProcess.0/@interaction.5 //@routingProcess.0/@interaction.7" output-
    Link="//@routingProcess.0/@interaction.13 //@routingProcess.0/@interaction.14 //@routingProcess
    .0/@interaction.15 //@routingProcess.0/@interaction.16 //@routingProcess.0/@interaction.17"
    nodeName="web2" behaviour="//@routingProcess.0/@behaviour.2"/>
42 <component inputLink="//@routingProcess.0/@interaction.2" outputLink="//@routingProcess.0/
    @interaction.4 //@routingProcess.0/@interaction.5" nodeName="cache1" behaviour="//
    @routingProcess.0/@behaviour.6"/>
43 <component inputLink="//@routingProcess.0/@interaction.3" outputLink="//@routingProcess.0/
    @interaction.6 //@routingProcess.0/@interaction.7" nodeName="cache2" behaviour="//
    @routingProcess.0/@behaviour.6"/>
44 <component inputLink="//@routingProcess.0/@interaction.0 //@routingProcess.0/@interaction.1" output-
    Link="//@routingProcess.0/@interaction.24 //@routingProcess.0/@interaction.25 //@routingProcess
    .0/@interaction.26 //@routingProcess.0/@interaction.27 //@routingProcess.0/@interaction.28"
    nodeName="loader1" behaviour="//@routingProcess.0/@behaviour.1"/>
45 <component outputLink="//@routingProcess.0/@interaction.18 //@routingProcess.0/@interaction.19 //
    @routingProcess.0/@interaction.20 //@routingProcess.0/@interaction.21 //@routingProcess.0/
    @interaction.22 //@routingProcess.0/@interaction.23" nodeName="loader2" behav-iour="//
    @routingProcess.0/@behaviour.1"/>
46 <component inputLink="//@routingProcess.0/@interaction.23" nodeName="memcache" behav-iour="//
    @routingProcess.0/@behaviour.4"/>
47 <component outputLink="//@routingProcess.0/@interaction.2 //@routingProcess.0/@interaction.3" node-
    Name="elasticloadbalancer" behaviour="//@routingProcess.0/@behaviour.5"/>
48 <behaviour component="//@routingProcess.0/@component.0 //@routingProcess.0/@component.1 //
    @routingProcess.0/@component.2 //@routingProcess.0/@component.3 //@routingProcess.0/@component.4
    " behaviourName="ElasticSearch"/>
49 <behaviour component="//@routingProcess.0/@component.11 //@routingProcess.0/@component.12" behaviour
    -Name="Loader"/>
50 <behaviour component="//@routingProcess.0/@component.7 //@routingProcess.0/@component.8" behaviour-
    Name="Web"/>
51 <behaviour component="//@routingProcess.0/@component.5 //@routingProcess.0/@component.6" behaviour-
    Name="Queue"/>
52 <behaviour component="//@routingProcess.0/@component.13" behaviourName="MemCache"/>
53 <behaviour component="//@routingProcess.0/@component.14" behaviourName="ElasticLoadBalancer"/>
54 <behaviour component="//@routingProcess.0/@component.9 //@routingProcess.0/@component.10" behaviour-
    Name="Cache"/>
55 </routingProcess>
56 </metamodel:NLSpecification>

```