International Journal of Modeling, Simulation,
and Scientific Computing
Vol. 7, No. 1 (2016) 1641003 (35 pages)

DOI: 10.1142/S1793962316410038

World Scientific
www.worldscientific.com

The high level language for system specification: A model-driven approach to systems engineering

Hamzat Olanrewaju Aliyu*
School of Information and Communication Technology
Federal University of Technology Minna
Gidan-Kwanu Campus, Minna, Nigeria
hamzat.aliyu@futminna.edu.ng

Oumar Maïga
Université des Sciences
des Techniques et des Technologies, Bamako, Mali
maigabababa78@yahoo.fr

Mamadou Kaba Traoré
LIMOS CNRS UMR 6158
Université Blaise Pascal, Clermont-Ferrand, France
traore@isima.fr

Received 1 October 2015
Accepted 29 January 2016
Published 10 March 2016

We present HiLLS (High Level Language for System Specification), a graphical formalism that allows to specify Discrete Event System (DES) models for analysis using methodologies like simulation, formal methods and enactment. HiLLS' syntax is built from the integration of concepts from System Theory and Software Engineering aided by simple concrete notations to describe the structural and behavioral aspects of DESs. This paper provides the syntax of HiLLS and its simulation semantics which is based on the Discrete Event System Specification (DEVS) formalism. From DEVS-based Modeling and Simulation (M&S) perspective, HiLLS is a platform-independent visual language with generic expressions that can serve as a front-end for most existing DEVS-based simulation environments with the aid of Model-Driven Engineering (MDE) techniques. It also suggests ways to fill some gaps in existing DEVS-based visual formalisms that inhibit complete specification of the behavior of complex DESs. We provide a case study to illustrate the core features of the language.

Keywords: HiLLS; modeling and simulation; integrated formalism; DEVS; Object-Z.

*Corresponding author.

1. Introduction

In science, engineering and many other disciplines, models are used to deal with the complexity of reality either as descriptive representations of existing systems or as constructive artifacts for creating nonexistent systems.[1] Analysis methodologies like Simulation, Formal Methods (FM) and Enactment are used to manipulate models for clearer understanding of the real world under study and forecast some realities that may be encountered in the future.

A typical simulation methodology allows to *compress time* and evaluate or analyze a model over a specified period. The results obtained may be used to predict system's performance, identify problems and their causes, etc.[2] Comprehensive lists of problems that are suitable for simulation are given in Refs. 2 and 3. FM are languages based on mathematical notations and techniques applied to the specification, analysis, design and implementation of complex software and hardware systems.[4–6] FM are used for rigorous logical investigation of the consistency of models through exhaustive exploration of its fulfilment of certain requirements and the generation of boundary test cases to give the assurance that a system will always produce desired results.[7,8] Enactment refers to the execution or interpretation of software process definitions which may involve interactions with the physical environment (e.g., human-in-the-loop).[9] In the context of model-based system engineering, enactment may be described as execution of the software representation of a system for a *real-time* verification of its behavioral properties.[10] The real time here refers to the use of clock-time as the reference for the scheduling and execution of activities specified in the software prototype of the system.

Due to the relative suitability of different methods to study disparate properties, a practical approach for exhaustive study of a complex system would be a systematic combination of different methods such that their results complement one another. For example, we may first use FM to investigate the fidelity of a model with regards to the system's requirements and then proceed to study the behavior of the accredited model and evaluate its performance using suitable simulation methodologies. We may also generate a software prototype for enactment. This approach will, on one hand, help each stakeholder see more clearly, how his/her interests affects those of others and vice versa and on the other hand, help engineers construct highly reliable systems.

We claim that a realization of this vision is possible through an integrated formalism that is expressive enough to specify reference models from which the models or specifications for the different target analysis methods can be obtained. Following this philosophy, we have defined HiLLS (High Level Language for System Specification), a graphical language whose abstract syntax is derived from an integration of concepts from system theory and software engineering. Figure 1 describes the rationale of HiLLS; it offers a unified abstract syntax for describing Discrete Event System (DES), a graphico-textual concrete syntax and three

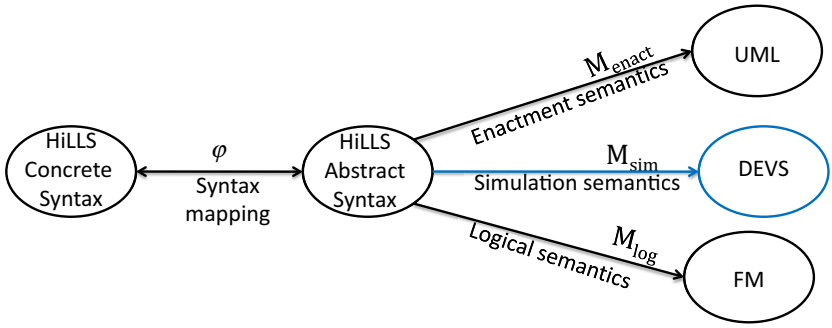


Fig. 1. HiLLS world view.

main semantics domains: Unified Modeling Language (UML),[11] Discrete Event System Specification (DEVS),[12] and FM for enactment, simulation and logical analysis, respectively. Therefore, a HiLLS model serves as a reference from which to automatically derive models for simulation, formal analysis and enactment using Model-Driven Engineering (MDE) techniques. We expect this approach to make the different analysis techniques more accessible especially to nonexpect users as well as reduce modeling efforts for expert users and foster communications among them.

The focus of the present paper is to present the syntax of HiLLS and its *simulation semantics* which is defined based on DEVS. Our choice of DEVS as the semantics domain for simulation-based analysis is motivated by its recognition as a common denominator for other DES formalisms as described in Ref. 13; therefore, we expect to be able to simulate a wider range of DES than would be possible with any other DES formalism as HiLLS' simulation semantics domain. Hence, to the discrete event simulation community, HiLLS may be seen as one of such graphical languages that seek to align the expressiveness and communicability of graphical concrete notations in Software Engineering with DEVS' foundational system theory to make the latter accessible to a diverse community of users. It must be noted, however, that HiLLS is not just a graphical notations for DEVS; DEVS is just one of its semantics domains as depicted in Fig. 1. We highlight some key benefits of HiLLS to DEVS-based graphical modeling:

- It allows for models to be subjected to rigorous logical analysis using FM to increase user's confidence in the model and hence, in the simulation results.
- It offers a more precise way to "graphically" describe the states and time advance of a complex system which, to our knowledge, are not yet adequately dealt with in the literature; these features are discussed in greater details in later sections of this paper.
- HiLLS allows for graphical description of dynamic structure systems. The details of this feature is out of the scope of this paper. Interested reader may refer to Ref. 14 for more details.
- It allows for code synthesis for system enactment.

In the next section, we present the state-of-the-art of DEVS-based visual languages and highlight some areas where there are needs for improvements. Overviews of DEVS and other formalisms from which DES concepts have been adopted to build the syntax of HiLLS are provided in Sec. 3. HiLLS' abstract and concrete syntax are presented in Sec. 4 followed by its simulation semantics, defined in DEVS, in Sec. 5 and a case study in Sec. 6 illustrate the use of the language. We compare and contrast HiLLS with existing DEVS-based visual languages in Sec. 7 before providing concluding remarks and directions for future work in Sec. 8.

2. DEVS-based Visual Languages: The State-of-the-Art

In this section, we present a short survey of existing DEVS-based graphical modeling languages and tools where we highlight their strengths and limitations with respect to some selected features. Based on the methodology and methods of model presentation, we classify the existing languages and tools into three categories:

- UML-Based presentation: a group of UML diagrams and/UML profiles to describe DEVS models.
- SysML-Based presentation: System Modeling Language (SysML) profiles to describe DEVS models.
- DSL: a new Domain-Specific Language (DSL) is created with user-defined concrete syntax.

Languages in the first category rely on the universality of the UML for software modeling, its wide acceptability in the industry and availability of supporting tools to leverage the complexity of abstract DEVS specification and to make DEVS accessible to a wider community of users. Examples of such proposals can be found in Refs. 15–17. They combine different kinds of formalisms in the UML family of diagrams to describe structural and behavioral aspects of DEVS. What is common to the different approaches is that they all use restricted stereotypes of UML component diagram to describe the structures of atomic and coupled DEVS where the required and provided UML component interfaces are used to express DEVS output and input ports, respectively. Hierarchical compositions in DEVS coupled models are expressed as UML components with sub-components to describe the DEVS sub-models while the connectors between the interfaces of components and sub-components express the different DEVS couplings. Different combinations of UML formalisms e.g., activity, sequence and state diagrams are used by different authors to describe the behaviors of atomic DEVS models. For instance, in eUDEVS,[15] the authors show how the variants of UML's sequence, timing and state machine diagrams can independently express the internal details of atomic DEVS models. In Ref. 17, the authors use activity diagrams attached to input ports to model DEVS external state transitions and state diagrams to model DEVS internal state transitions.

The second category, SysML-based representations,[18,19] use stereotypes of SysML[20] diagrams to describe DEVS models. The authors of Ref. 18 argue that SysML is more suitable than UML for graphical description of DEVS than UML since the former, and especially its block diagrams naturally provide the means to describe model compositions in line with the DEVS formalism. Basically, the external structure of DEVS atomic and coupled models are expressed with SysML blocks with inflow and outflow ports describing the DEVS input and output ports. Hierarchical compositions of sub models in DEVS coupled models are described with the SysML's Block Definition Diagram while the couplings between the various sub-models are described in a separate SysML Internal Block Diagram (IBD). The internal structure and behavior of a DEVS atomic model is described with four separate diagrams to express its state space, state transitions and output functions.

The third category is the creation of new DSL to express DEVS concepts. Some prominent examples in this category are CD++,[21,22] DEVS diagram,[23] and DEVS-Driven Modeling Language (DDML)[24–26] and MS4 Modeling Environment (MS4 Me)[27,28] which offers graphical and textual modeling interfaces for domain and system experts, respectively.

In Ref. 23, Song and Kim argue that a dedicated DSL would allow to create more suitable notations originally for DEVS to describe complex systems effectively than relying on existing notations that have been created for some other purposes. CD++ describes a DEVS model as a directed graph with bubbles as nodes and arrow edges connecting them. The sub-models of a DEVS Coupled model are described by bubbles with directed arrows indicating the couplings between them. In DEVS atomic models, states are bubbles containing the state id and the value of time advance while directed edges connecting them express state transitions. The DEVS Diagram describes the external structure of a DEVS model as a rectangle with input and output ports indicated on its left and right sides, respectively. The hierarchical construction of DEVS coupled models are achieved by nesting the sub-models inside the parent models and expressing the couplings with directed lines between ports. The behavior of a DEVS atomic model is described by a structured form of DEVS Graph where states are structured into state variables and a finite set of phases such that a phase is an abstraction of states that produce the same outputs and/or have the same time advance. The DDML present DEVS models in a similar manner as the DEVS Diagram except for a few differences in the description of DEVS atomic models; in addition to the features provided by the latter, the former also allows for the specification of methods to describe functional processes that may be used in the computation events and reconfiguration of state variables.

MS4 Me is a DEVS-based Modeling and Simulation environment that provides interfaces to support users with varying/different levels of expertise, concerns, needs and roles. As such, it serves as a platform for collaborative building of models and simulations between domain experts and DEVS modelers. MS4 Me allows a domain expert to generate DEVS structural and behavioral models from the

UML[11] sequence and state diagrams, respectively. For the DEVS modeler, however, complex DEVS models are built in MS4 Me with a combination of two main formalisms: 'enhanced' Finite Deterministic DEVS (FDDEVS) (see Chapters 4 and 12 in Ref. 27) and System Entity Structure (SES).[29] *Enhanced* FDDEVS and SES use Xtext-based constrained/restricted natural language to precisely specify DEVS atomic and coupled models, respectively. The former uses Java codes placed in tag blocks to specify the logics of state transition and output functions so that the codes are directly copied in the appropriate file during code synthesis. A major strength of the SES is that it allows for the specification of a family of hierarchical models rather than a single composition. It offers the concept of decomposition that breaks a complex system into simpler hierarchical modules (from certain perspectives) called components that are coupled together through the specification of message flows between their I/O ports. The specialization feature allows for the specification of multiple kinds of certain components so that the modeler can explore different combinations of alternatives offered by the specialization through a process called pruning.

Having studied a good number of previous work in this context, we suggest four questions that may need to be answered by a DEVS-based visual modeling language in order to make modeling more easier and to provide means to represent complex systems more accurately:

(1) Can component model be reused within the same specification?
(2) How are the states of atomic DEVS models described?
(3) How are the time advances of states specified?
(4) Can a complex input event or port type be accurately specified?

These questions are not problems with code-based modeling tools as the underlying programming languages often offer the flexibility to deal with each of them. It is, however, not the case for graphical modeling languages and that is why majority of them have to resort to manual programming to complete the modeling tasks especially in the specification of system behaviors.

Recent works on DEVS-based textual languages have proffered some interesting answers to some of these questions; notable among them are DEVSML 2.0 (DEVS Modeling Language 2.0) by Mittal and Douglass[30] and CML-DEVS by Hollmann *et al.*[31] The Xtext-based[32] DEVSML 2.0 deals with (1) and (4). It addresses (1) through the importation of packages containing model components and entities wherever they are needed in similitude to what is done in programming languages like Java and C++ while nonprimitive message and port types are defined as struct-like data structures to address (4). CML-DEVS, inspired by FM, addresses (2) and (4) using rigorous mathematical system specifications. In addressing (2), CML-DEVS declares typed state variables such that the instantaneous states of the system are defined by the sets of values of the variables at such instants. Using the type system that is inherent in FM, it addresses (4) by defining complex data structures

Table 1. State-of-the-art of DEVS-based visual modeling languages.

Category	DEVS-based visual languages	Component reuse	Time advance	State specification	Complex input
UML-based	eUDEVS[a]	×	Fixed-valued	Enumerated[b]	×
SysML-based	see Refs. 18 and 19	✓	Fixed-valued	Enumerated[b]	×
DSL-based	CD++	×	Fixed-valued	Enumerated[b]	×
	DEVS Diagram	×	Fixed-valued	Structured state[c]	×
	DDML	×	Fixed-valued	Structured state[c]	×
	MS4 Me	✓	Fixed-valued	Structured state[c]	✓

Note: [a]Other examples in Ref. 17.
[b]A finite set of arbitrary state identifiers.
[c]State variables and a finite set of phases; a phase is a set of states that satisfy some conditions.

as port and message types. Since the work presented in this paper is a graphical language, our discussions of related work will be limited to DEVS-based graphical languages in the rest of the paper.

Table 1 shows the answers of the three categories of DEVS-based graphical languages to these questions. SysML-based languages favor the reuse of component specifications among coupled models more than the other two categories for graphical languages; DDML takes this feature into consideration in its earliest proposal,[24] however, it is not developed further in its implementation stages. Among the formalisms cited by this paper in the DSL category, only MS4 Me supports the reuse of specifications as components in multiple coupled DEVS models; DDML takes this feature into consideration in its earliest proposal,[24] however, it is not developed further in its implementation stages.

In Table 1, a *structured state* is meant to describe a specification of the states of a system based on the instantaneous values of its state variables. This is done by defining unique *predicates* that mark the properties of each state such that a state is assumed if and only if its predicate is true. We argue that this approach offers the modeler a handy means to effectively describe the reality of systems with reasonable amount of details. An *enumerated* state specification refers to the definition of a set of strings to describe states with little details about variables responsible for each state. With the exception of CD++, the DSL-based languages we have studied support structured state specifications. Languages in the other two categories often support enumerated state specification. Considering that the real state spaces of most systems are infinite, and that it is practically impossible to graphically represent each individual states, the approach used in DEVS Diagram and DDML provide a means to accurately model complex systems by specifying constraints to partition the state space into manageable finite set of phases.

Our study also reveals that languages in all categories describe time advance as constant values associated with the specified states/phases. We share the opinion of Schülz *et al.*[33] that the time advance function plays a critical role in the definition of the semantics of a model; therefore, we think that more efforts should be

made to ensure the time advance of states are as close as possible to their actual values. We argue that all the states grouped together in a phase (as described in DEVS Diagram) may not necessarily have 'exactly' the same value of time advance; rather, the time advance should be obtainable dynamically from a given equation or expression which may likely depend on the exact values of some state variables for each individual state.

Lastly, with the exception of MS4 Me, languages studied in all categories often permit modeling input events as strings or primitive data types such as integer, boolean, etc. Where some more complex objects are required as input events, this is usually done at the programming implementation level. It would be fair, however, to state that MS4 Me supports this feature at the advanced level with the Xtext-based textual Constrained Natural Language and not with the graphical editors meant for use by domain experts. We think it would be interesting to also be able to specify such complex input objects as part of graphical models. This was also considered in the early proposal of DDML but it was not development further.

We can arguably say that the realization of a language that satisfactorily answers the four questions suggested in this section is possible with the exploitation of Software Engineering techniques. We will show in later sections of this paper how HiLLS attempts to answer the questions.

3. Background

3.1. *DEVS*

DEVS[12] is a system-theoretic mathematical formalism for specifying DESs as abstract mathematical objects for simulation. It supports the specification of a full range of DESs as other formalisms for systems in this category have been proven to have equivalent DEVS representations.[13] It however does not provide any concrete syntax to express the system constructs; we can take advantage of this freedom by providing concrete specifications in a DSL with formal semantics that adopt the DEVS simulation protocol as its operational semantics.

Basically, DEVS defines two abstraction levels for DESs - *atomic* and *coupled* DEVS. An atomic DEVS has a time base; state, input and output sets; and functions that define successive states and outputs events. A coupled DEVS is an hierarchical composition of atomic and/or coupled DEVS as components with couplings between their input/output ports to enable their interactions.

Traditionally, DEVS exists in two major forms: classic DEVS (CDEVS) and parallel DEVS (PDEVS),[12,34] the main difference being that the later supports concurrent state transitions within components of a coupled DEVS while the former uses a tie-breaking function to enforce sequential state transitions within components. Subsequently in this paper, we will use PDEVS and refer to it simply as DEVS. Atomic DEVS, AM, is defined as:

$$\mathrm{AM} = \langle X, Y, S, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \delta_{\mathrm{conf}}, \lambda, \mathrm{ta} \rangle, \tag{1}$$

where:

$X = \{(p, v) \,|\, p \in IPorts \wedge v \in \text{dom}(p)\}$,
$Y = \{(q, v) \,|\, q \in OPorts \wedge v \in \text{dom}(q)\}$,
S: *Abstract state set* , $\delta_{\text{int}} \colon S \to S$,
$\delta_{\text{ext}} \colon Q \times X^b \to S$ with $Q = \{(s, e) \,|\, s \in S \wedge 0 \leq e < \text{ta}(s)\}$,
$\delta_{\text{conf}} \colon S \times X^b \to S$, $\lambda : S \to Y^b$ and $\text{ta} \colon S \to \mathbb{R}^+_{0,\infty}$.

X and Y are the sets of input and output *events* respectively with $IPorts$ as set of input ports and $OPorts$ as set of output ports. An event, v, in this context is a value generated in form of a message that triggers an action by its recipient. Intuitively, an input event is associated to an input port through which it was received. Similarly, an event generated for output is associated to an output port through which it is sent to the system's environment. S is the set of states; at any given moment, the system is in a state $s \in S$. The time advance function, ta, maps each state to a specified duration after which a scheduled internal state transition, δ_{int}, is automatically fired. The external transition function, δ_{ext}, specifies the system's response to the input event(s) before the expiration of the ta of current state; Q is called the set of total states and e is the elapsed time since the last state transition. If the input event coincides with the expiration of the ta, then a confluent transition function, δ_{conf}, is invoked instead. The superscript b of X^b denotes a bag of input events. The function λ defines the outputs that may accompany internal and/or confluent state transitions. Similarly, the superscript b of Y^b denotes a bag of output events.

The Coupled DEVS, CM is defined as:

$$CM = \langle X, Y, D, \{M_d\}_{d \in D}, EIC, EOC, IC \rangle, \tag{2}$$

where:

$EIC = \{((CM, ip_{CM}), (d, ip_d)) \,|\, ip_{CM} \in IPorts_{CM} \wedge ip_d \in IPorts_d\}_{d \in D}$,
$EOC = \{((d, op_d), (CM, op_{CM})) \,|\, op_{CM} \in OPorts_{CM} \wedge op_d \in OPorts_d\}_{d \in D}$,
$IC = \{((a, op_a), (b, ip_b)) \,|\, op_a \in OPorts_a \wedge ip_b \in IPorts_b\}$ with $a, b \in D$.

X and Y are as defined for atomic DEVS and D is the set of names of components of CM such that M_d is the DEVS specification referred to by d for all $d \in D$. An EIC, External Input Coupling, is a connection between an input port of CM and an input port of one of its components, an EOC, External Output Coupling, is a connection between an output port of CM and an output port of one of its components and an IC, Internal Coupling, is a connection between the output port of a component of CM and an input port of another peer component. The essence of the couplings is to allow for interactions between system components. Given an element $((S, p_S), (R, p_R))$ of any of the relations EIC, EOC and IC, S (sender) influences R (receiver) by sending a message (event) from p_S to p_R. It is important to note here that CM defines a logical boundary between all its components and the environment; therefore, any of the components can only influence (or be

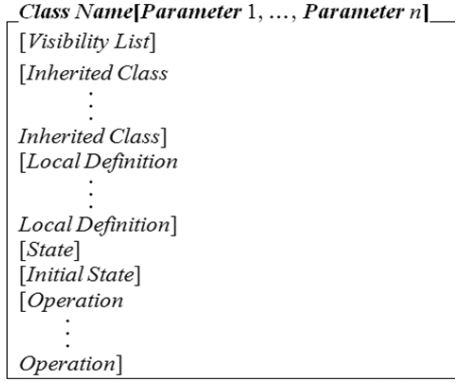


Fig. 2. Template of Object-Z class.

influenced by) the environment through the interfaces of CM, hence the need for EOC and EIC. More details on DEVS and its simulation protocols can be found in Ref. 12.

3.2. *Object-Z*

Object-Z (see Ref. 35) is an Object-Oriented extension of the Z formal specification language.[36] It adopts the concept of class in Object Orientation (OO) to add structure, modularity and clarity to Z specifications. The main units of specification in Z are the schemas that declare state variables with possible invariants and operation schemas that manipulate the state schema(s) to produce state transition events. On top of the Z's notion of schema, the Object-Z class encloses a single state schema and all the operation schemas that manipulate and/or use its declared variables. The Object-Z class also exhibits OO properties like inheritance, encapsulation and polymorphism. Figure 2 shows a general template (as described in Ref. 35) for specifying an Object-Z class, the basic building block for system specification in Object-Z, showing its possible elements and the orders in which they may appear.

An Object-Z class has a unique name as an identifier to differentiate it from other classes in the specification. In addition to the class name, the header may also specify some generic parameters. Since the Object-Z class encapsulates its contents, the *visibility list* specifies the interface through which the elements of an object of the class may be accessed i.e., a list of variables and operations that can be visible outside the class in similitude to public attributes and methods in OO. An *Inherited Class* designator provides a reference to an existing Object-Z class whose definition is imported for reuse in the current class similar to the concept of inheritance in OO. A *Local Definition* may be a local type or constant definition (usually specified in an axiomatic schema) or a reference to another class. A class may have a maximum of one *state schema* that defines its state space through

the declaration of state variables and invariants (if any); an invariant on a set of variables is a predicate that must always be satisfied throughout the lifespan of the system. This may be followed by a specification of the *initial state* simply referred to as init; the init specifies a set of predicates that must be satisfied by the state variables (declared in the state schema) of every new object of the class before it undergoes any change of state. Finally, the operations that use and/or manipulates the elements of the class. More details about Object-Z's syntax and semantics can be found in Ref. 35.

There are many specification formalisms for logical analysis; we have chosen Object-Z for three properties, two of which it inherits from Z (its base formalism). (1) Z is said to be considerably universal and suitable for describing DESs for most kinds of logical analysis; (2) Z allows for separation of concerns i.e., its syntax enable to decouple the specification of system properties from the requirements investigation logics and (3) OO (this is peculiar to Object-Z) which enables modular specification and analysis of complex systems.

4. HiLLS

HiLLS evolves from the DDML,[24–26] a graphical modeling language built on DEVS to facilitate the use of the latter by domain experts via graphical concrete syntax to describe system models. The goal of HiLLS is to be able to create multi-semantic models that can be used for simulation, formal analysis and enactment.

HiLLS' syntax combines system-theoretic and Software Engineering concepts adopted from the DEVS and Object-Z, respectively. Our choices of system constructs from Object-Z and DEVS are motivated by their universalities in their respective domains; while the former claims suitability for modeling most kinds of state-based systems for logical reasoning, the later has been proven to be a common denominator to most DES simulation formalisms.[13] Moreover, the combination allows to reuse Object-Z constructs such as predicates and expressions for the refinement of abstract constructs such as states and transitions functions adopted from DEVS. This feature also aids the synthesis of executable program codes for enactment.

In addition to the DEVS-based system-theoretic concepts in HiLLS, the syntax also adds concepts to describe structural changes in DSSs. HiLLS' approach to modeling DSSs is unique in that it provides a simple and graphical means of doing it; we demonstrate this with a case study in a later section.

4.1. *Abstract syntax*

Figure 3 is an excerpt from the HiLLS' abstract syntax. The segment within a dashed-box describes a DES as an *HSystem* which may be an atomic unit or composed of interacting components (*hComponents*). It may have input and/or output ports (class *Port*) for interacting with its environment by exchanging messages called *Event*s. By its inheriting the class *HClassifier*, an *HSystem* may have

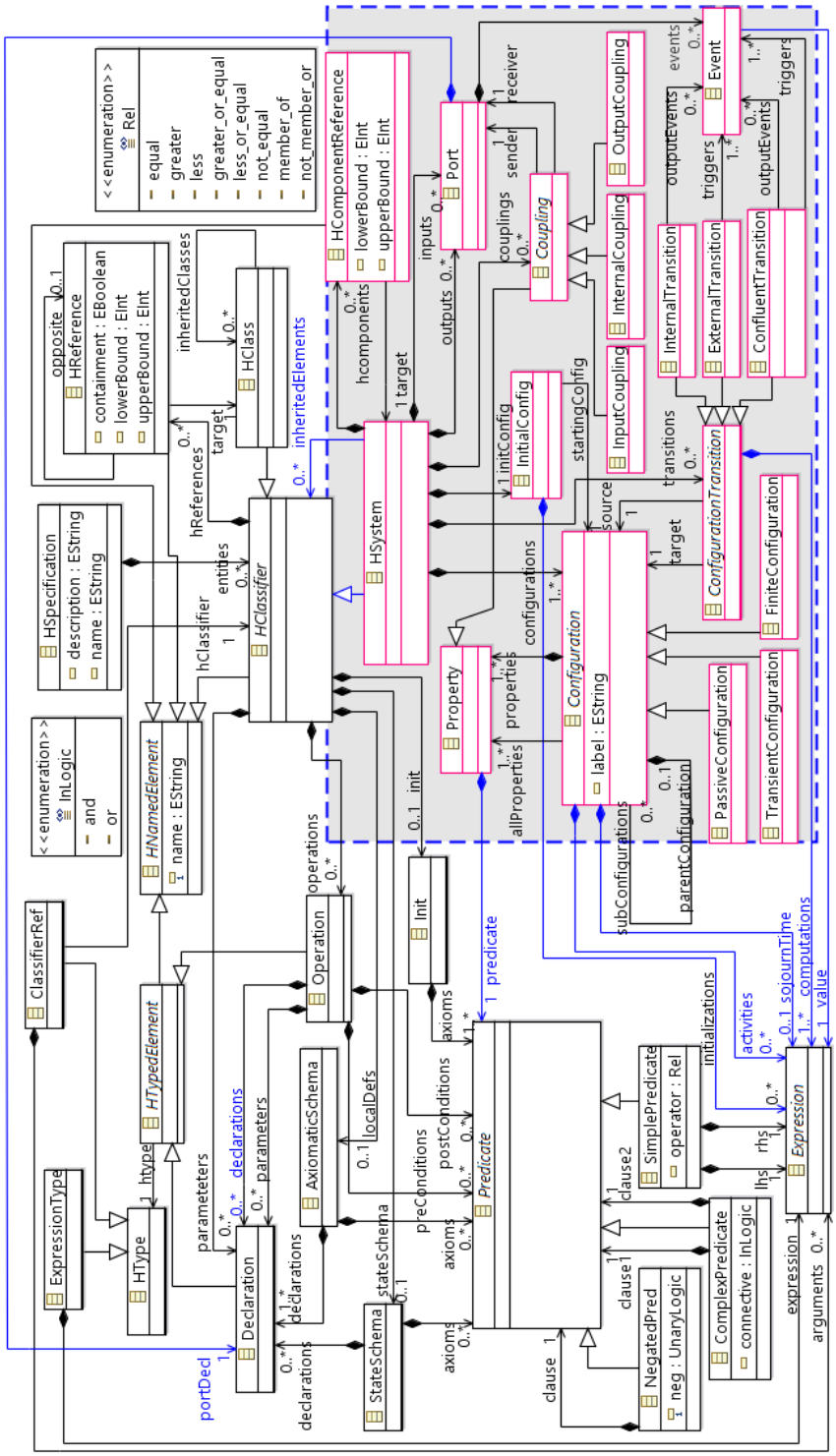


Fig. 3. Abstract syntax of HiLLS.

a *StateSchema* in which state variables are declared with possible constraints, an *AxiomaticSchema* that defines global parameters, and *Operations* that manipulate the system's variables and parameters.

The system's behavior is described by a finite set of configurations and transitions between them. A *Configuration* is a cluster of all states satisfying a unique *property* defined on the state variables. The *sojournTime* of a configuration is the duration for which it may be assumed before a scheduled transition occurs. A configuration is regarded as *Passive*, *Transient* or *Finite* if its *sojournTime* is positive infinity ($+\infty$), zero (0) or positive real number greater than zero ($\mathbb{R}^+$), respectively. A *ConfigurationTransition* belongs to one of three categories: an *internal* transition occurs at the expiration of the *sojournTime* of the current configuration, an external transition occurs whenever an *input* is received before the end of the *sojournTime* while a confluent transition occurs when the reception of an input coincides with the expiration of the *sojournTime*. A transition is accompanied by a sequence of *computations* that manipulate the state variables to satisfy the *property* of the target configuration; it may also involve sending *events* to some output ports.

Coupling describes the relations between the ports of the components of a system. Systems influence one another by exchanging *events* through their input and output ports. Therefore, a coupling is a property that establishes a relation between a source port (*sender*) and a target port (*receiver*) for the exchange of events. *InputCoupling*, *InternalCoupling* and *OutputCoupling* have the same definitions as DEVS' *EIC*, *IC* and *EOC*, respectively. In addition to the amenability of Z to logical reasoning, the segment of the meta-model outside the dashed-box are reused for the refinement of the system-theoretic concepts through their associations with them; examples are the associations between the following pairs of components: (*Property*, *Predicate*), (*Port*, *Declaration*) and (*Event*, *Expression*).

The class *HClass* describes objects/entities that have attributes and operations but no behavioral specifications. This is similar to *Class* in UML and its purpose is to enable HiLLS' users to model complex attributes, ports and message types.

4.2. *Concrete syntax*

The concrete notations to express HiLLS' concepts are described in Figs. 4(a)–4(h). *HClass* (a) is denoted by a box with three compartments similar to the UML class symbol. The first compartment contains the HClass' name and parameters if any. The second compartment houses the state and axiomatic schema if any. We adopt the notations of the state schema and axiomatic schema as used in Z. The third and last compartment houses the definitions of the class' operations if any. An operation is similar to the state schema but with additional information indicated on its top side. The top bears the name attribute of the operation, the list of parameter declarations (if any) and the type of the operation. Similarly, an empty type

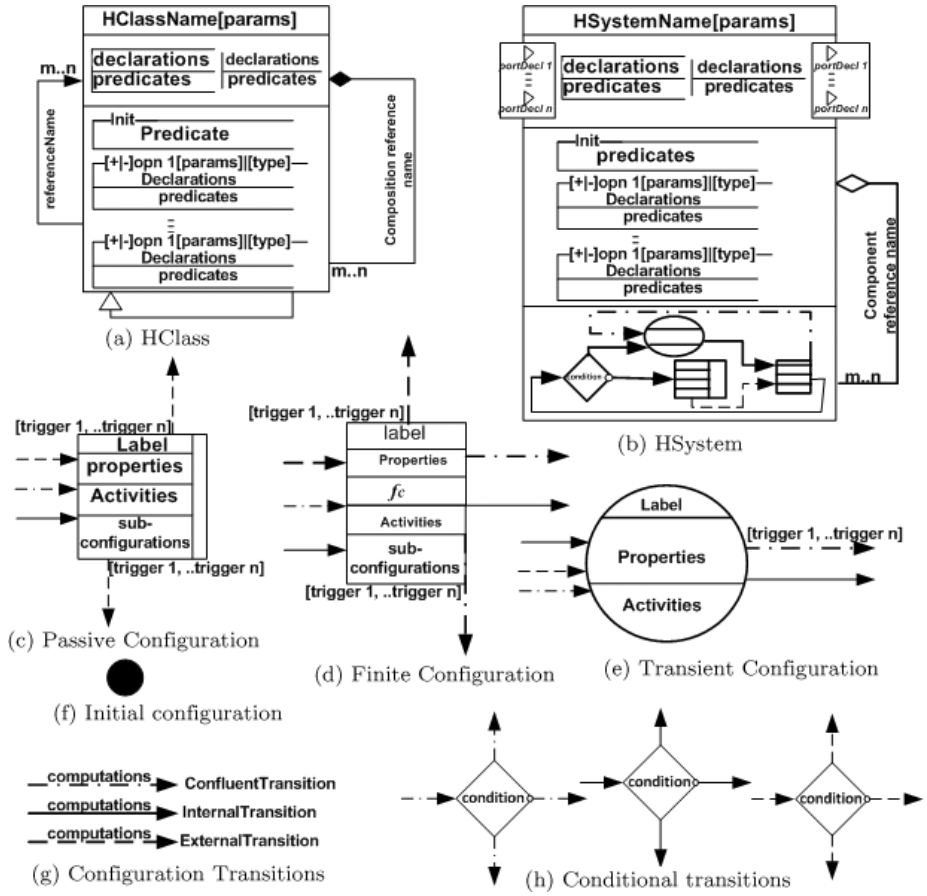


Fig. 4. Concrete syntax of HiLLS.

bracket denotes an operation that does not produce any output. Associations such as inherited class, composition and class reference use the corresponding notations in as used in the UML class diagram.

The *HSystem* (b) notation extends that of HClass; it has four compartments with the first three serving similar functions as in HClass while the fourth contains the transition diagram that describes the system's behavior. The input and output interfaces are denoted by windows attached to the left and right sides respectively of the second compartment. In each rectangular window, a port is denoted by a small arrowhead labeled with the port declaration.

The notation for a finite configuration (d) is a box with five compartments for label, properties, sojournTime, activities and sub-configurations respectively from top to bottom. Passive configuration (c) is similar to finite configuration except that the compartment for sojournTime is not represented; a vertical stripe is attached to its right side as an indication of its infinite sojournTime.

Transient configuration (e) is denoted by a circle with three compartments for its label, properties and activities if any. Its shape depicts its zero sojournTime.

Configuration transitions are represented by arrow-ended lines (g) from source to target configurations with associated computations as textual labels. A conditional statement in the path of a transition may initiate a choice between one of two targets. In such cases, the condition is enclosed within a diamond (h). To disambiguate the flow of the computations, the transition arrow flows into the left corner of the diamond and flows out from a small circle attached to one corner if the condition is true; otherwise, it flows out from either the remaining two corners. To disambiguate the flow of the computations, the transition arrow flows into the left corner of the diamond and flows out from a circle attached to the right corner if the condition is true; otherwise, it flows out from either the top or bottom corner.

5. Simulation Semantics of HiLLS

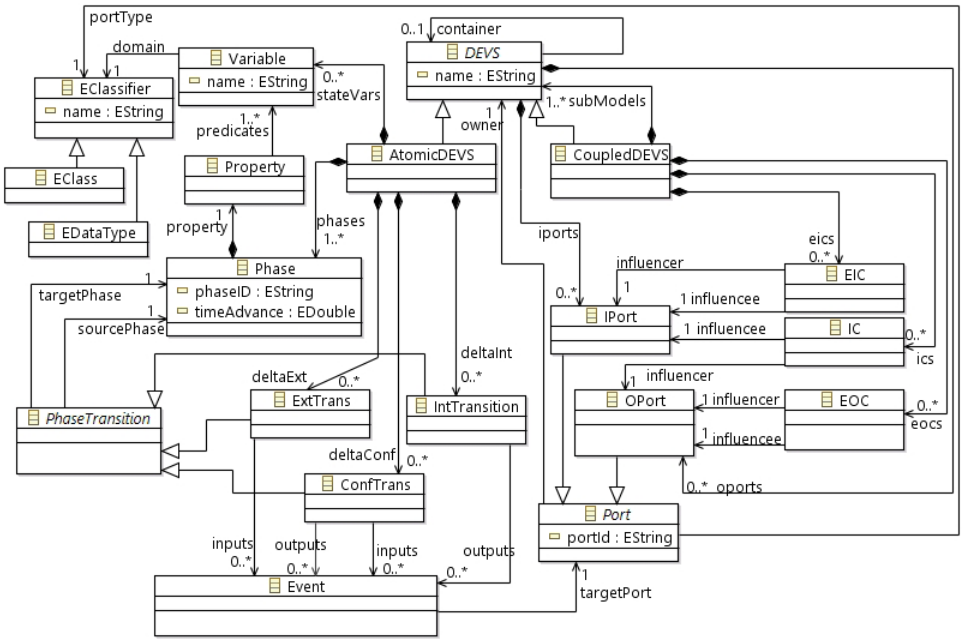
This section presents the mapping of HiLLS concepts to the domain of DEVS for simulation. We have provided a metamodel of HiLLS in the previous section, it would be reasonable to also provide a metamodel of DEVS concepts so that the relations between the two domains can be clearly done in the same technical space.

5.1. *DEVS metamodel*

Figure 5(a) presents a metamodel of the DEVS concepts presented in Eqs. (1) and (2) (see Sec. 3).

As shown by the two sub-types of the abstract *DEVS* class in Fig. 5(a), DEVS describes a DES as either an *AtomicDEVS* or *CoupledDEVS*. A DEVS model may have zero or more input ports, *iports*, and/or zero or more output ports, *oports*; a port is defined by a name, *portid*, and a type, *portType*, which may be a Class or primitive DataType.

An *AtomicDEVS* defines state variables, *stateVars* and a finite set of *phases* where a *Phase* is an abstraction of a unique combination of values of (or predicate on) the state variables. Technically, the phases constitute disjointed subsets of the state space. A phase is characterized by a *timeAdvance*. An *AtomicDEVS* may also define sets *deltaInt*, *deltaExt* and *deltaConf* of internal, external and confluent phase transitions, respectively. Each *IntTransition* and *ConfTransition* may be accompanied by a bag, *outputs*, of events while every *ExtTransition* and *ConfTransition* is triggered by a bag, *intputs*, of events. Additional information on phase transitions are provided the OCL (Object Constraint Language) constraints in Fig. 5(b). The constraints define some restrictions on exceptional cases in which each of the transitions may not occur. A *CoupledDEVS* defines a set, A *subModels*, of at least one component(s) of the A *container* model. It also defines the sets *eics*, *ics* and *eocs*



(a) DEVS metamodel

```
1  import 'DEVS.ecore'
2  package devs
3  context PhaseTransition
4      def:zeroTime:Real = 0.0
5      def:inf:Real = UnlimitedNatural
6      inv delta_Int_constraint('no internal transition from a passive state'):
7          oclIsKindOf(IntTransition) implies sourcePhase.timeAdvance < inf
8      inv delta_Conf_constraint('no confluent transition from a passive state'):
9          oclIsKindOf(ConfTrans) implies sourcePhase.timeAdvance < inf
10     inv delta_Ext_constraint('no external transition from a transient state'):
11         oclIsKindOf(ExtTrans) implies sourcePhase.timeAdvance > zeroTime
12 context EIC
13     inv EIC_components_constraint('EIC is between a coupled model and its sub-model'):
14         influencer.owner.oclAsType(CoupledDEVS).subModels->includes(influencee.owner)
15     inv EIC_ports_constriant('EIC must be between two input ports'):
16         influencee.oclIsKindOf(IPort) and influencer.oclIsKindOf(IPort)
17 context IC
18     inv IC_components_constraint('IC is between peer components of a coupled model'):
19         influencer.owner.container = influencee.owner.container
20     inv IC_ports_constriant('IC is from an output port to an input port'):
21         influencer.oclIsKindOf(OPort) and influencee.oclIsKindOf(IPort)
22     inv IC_ports_constriant('Feedback loop is not allowed in DEVS'):
23         influencer.owner <> influencee.owner
24 context EOC
25     inv EOC_components_constraint('EOC must be between a sub-model and its container'):
26         influencee.owner.oclAsType(CoupledDEVS).subModels->includes(influencer.owner)
27     inv EIC_ports_constriant('EOC must be between two output ports'):
28         influencee.oclIsKindOf(OPort) and influencer.oclIsKindOf(OPort)
29 endpackage
```

(b) DEVS transition and coupling constraints

Fig. 5. DEVS metamodel and static semantics.

of couplings between the components of the model. Figure 5(b) also provides additional information and restrictions on the three kinds of couplings in accordance to the DEVS formalism.

5.2. *Mapping of HiLLS concepts to DEVS*

This subsection presents the mapping between concepts described in the HiLLS and DEVS metamodels. The mapping rules have been specified with the ATLAS Transformation Language (ATL).[37] Figure 6(a) shows the mapping rules to obtain an *AtomicDEVS* from a given *HSystem* with an empty *hComponents*. The elements of *AtomicDEVS* as described in the DEVS metamodel are shown on the left-hand

```
rule HSystem2AtomicDEVS {
 from--HSystem without components -> Atomic DEVS
  hsystem : HiLLS!HSystem(hsystem.hcomponents->isEmpty())
 to
  atomicDEVS : DEVS!AtomicDEVS (
    name <- hsystem.name,
    iports <- hsystem.inputs->collect(p|thisModule.HiLLSPort2DEVSInput(p)),
    oports <- hsystem.outputs->collect(q|thisModule.HiLLSPort2DEVSOutput(q)),
    stateVars <- hsystem.stateSchema.declarations->collect(v|thisModule.HiLLSVar2DEVSVar(v)),
    phases <- hsystem.configurations->collect(ph|thisModule.HiLLSConfig2DEVS_Phase(ph)),
    deltaInt <- hsystem.transitions->collect(dInt|thisModule.HiLLSTrans2DEVSdeltaInt(dInt)),
    deltaExt <-hsystem.transitions->collect(dExt|thisModule.HiLLSTrans2DEVSdeltaExt(dExt)),
    deltaConf <-hsystem.transitions->collect(dConf|thisModule.HiLLSTran2DEVSdeltaConf(dConf))
  )
}
```

(a) Mapping a HiLLS HSystem without components to DEVS Atomic Model

```
lazy rule HiLLSPort2DEVSInput {
  from
    hillsPort : HiLLS!Port
  to  --HiLLS port -> DEVS input port
    devs_input : DEVS!IPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
  )
}
```

(b) HiLLS port → DEVS IPort

```
lazy rule HiLLSPort2DEVSOutput {
  from
    hillsPort : HiLLS!Port
  to --HiLLS port -> DEVS output port
    devs_output : DEVS!OPort (
    portId <- hillsPort.portDecl.name,
    portType <- hillsPort.portDecl.htype
 )
}
```

(c) HiLLS port → DEVS OPort

```
lazy rule HiLLSVar2DEVSVar {
  from
    hillsVar : HiLLS!Declaration
  to --HiLLS Declaration->DEVS state vars
    devsVar : DEVS!Variable (
    name <- hillsVar.name,
    domain <- hillsVar.htype
 )
}
```

(d) HiLLS declaration → DEVS variable

```
lazy rule HiLLSConfig2DEVS_Phase{
  from
    hillsConfig : HiLLS!Configuration
  to --HiLLS configuration -> DEVS phase
    devsPhase : DEVS!Phase (
    stateID <- hillsConfig.label,
    property <- hillsConfig.properties,
    timeAdvance <- hillsConfig.sojournTime
 )
}
```

(e) HiLLS configuration → DEVS phase

Fig. 6. Mapping rules of HiLLS concepts to Atomic DEVS concepts.

side of the rule with the corresponding *HSystem* elements on the right-hand side. The "lazy rules" in Figs. 6(b) and 6(c) provide the rules mapping individual HiLLS input and output ports to DEVS input and output ports with HiLLS port declaration name and type mapping to DEVS port id and type, respectively. These "lazy" rules are invoked from the *AtomicDEVS* and *CoupledDEVS* rules to obtain their input and output ports. In Figs. 6(d) and 6(e), we show the mapping rules to obtain DEVS state variables and phases from HiLLS state variables and configurations, respectively.

Similarly, individual HiLLS' *InternalTransition*, *ExternalTransition* and *ConfluentTransition* are mapped to DEVS' *DeltaInt*, *DeltaExt* and *DeltaConf* respectively by the rules in Fig. 7. It is important to state here that the imperative HiLLS computations that accompanying the transitions cannot be explicitly accounted for in this declarative mapping. A more complete Model-to-Text transformation technique may be used to provide the rules to generate such codes for a target DEVS-based implementation in a programming language.

In Fig. 8, we show the correspondences between an *HSystem* with nonempty *hComponents* and *CoupledDEVS* concepts. While Fig. 8(a) provides the rules to obtain the different sets, Figs. 8(b)–8(d) show the rules for obtaining individual EIC, IC and EOC, respectively. The rules for obtaining individual input and output ports have been presented previously in Figs. 6(b) and 6(c).

```
lazy rule HiLLSTrans2DEVSdeltaInt {
  from --InternalTransition-> DeltaInt
    intTrans: HiLLS!InternalTransition
  to
    deltaInt : DEVS!DeltaInt (
    sourcePhase <- intTrans.source,
    targetPhase <- intTrans.target,
    outputs <- intTrans.outputEvents
  )
}
```

(a) Mapping internal transitions

```
lazy rule HiLLSTrans2DEVSdeltaExt {
  from --ExternalTransition-> DeltaExt
    outTrans: HiLLS!ExternalTransition
  to
    deltaInt : DEVS!DeltaExt (
    sourcePhase <-  outTrans.source,
    targetPhase <-  outTrans.target,
    inputs <- outTrans.triggers
  )
}
```

(b) Mapping external transitions

```
lazy rule HiLLSTrans2DEVSdeltaConf {
  from --ConfluentTransition-> DeltaConf
    confTrans: HiLLS!ConfluentTransition
  to
    deltaInt : DEVS!DeltaConf (
    sourcePhase <- confTrans.source,
    targetPhase <- confTrans.target,
    inputs <- confTrans.triggers,
    outputs <- confTrans.outputEvents
  )
}
```

(c) Mapping confluent transitions

Fig. 7. Mapping HiLLS configuration transitions to DEVS phase transitions.

```
rule HSystem2CoupledDEVS {
  from  --An HSystem with components -> Coupled DEVS
    hsystem : HiLLS!HSystem(hsystem.hcomponents->notEmpty())
  to
    coupledDEVS : DEVS!CoupledDEVS (
      name <- hsystem.name,
      iports <- hsystem.inputs->collect(p|thisModule.HiLLSPort2DEVSInput(p)),
      oports <- hsystem.outputs->collect(q|thisModule.HiLLSPort2DEVSOutput(q)),
      subModels <- hsystem.hcomponents->collect(comp | comp.target.name),--model references
      eics <- hsystem.couplings->collect(eic|thisModule.HiLLSInputCoupling2DEVS_EIC(eic)),
      ics <- hsystem.couplings->collect(ic|thisModule.HiLLSInternalCoupling2DEVS_IC(ic)),
      eocs <- hsystem.couplings->collect(eoc|thisModule.HiLLSOutputCoupling2DEVS_EOC(eoc))
  )
}
```

(a) Mapping of HiLLS HSystem to DEVS Coupled Model

```
lazy rule HiLLSInputCoupling2DEVS_EIC {
  from --HiLLS Inputcoupling -> DEVS EIC
    inCoupling: HiLLS!InputCoupling
  to
    devsEIC : DEVS!EIC (
    influencer <- inCoupling.sender,
    influencee <- inCoupling.receiver
  )
}
```

(b) HiLLS InputCoupling to DEVS EIC

```
lazy rule HiLLSInternalCoupling2DEVS_IC {
  from --HiLLS InternalCoupling -> DEVS IC
    peerCoupling: HiLLS!InternalCoupling
  to
    devsEIC : DEVS!IC (
    influencer <- peerCoupling.sender,
    influencee <- peerCoupling.receiver
  )
}
```

(c) HiLLS InternalCoupling to DEVS EI

```
lazy rule HiLLSOutputCoupling2DEVS_EOC {
  from --HiLLS OutputCoupling -> DEVS EIC
    outCoupling: HiLLS!OutputCoupling
  to
    devsEOC : DEVS!EOC (
    influencer <- outCoupling.sender,
    influencee <- outCoupling.receiver
  )
}
```

(d) HiLLS OutputCoupling to DEVS EOC

Fig. 8. Mapping rules of HiLLS concepts to Coupled DEVS concepts part.

6. Case Study: The Alternating Bit Protocol

The Alternating Bit Protocol (ABP)[38] is a communication protocol to transmit messages over unreliable communication channels and ensure they are delivered in the correct orders in spite of disturbances in the channels. Figure 9 presents the components of ABP and communications between them; it consists of a *sender*, a *receiver*, a *message transmission channel* and an *acknowledgement*(*ack*) *channel*. The basic principle of the protocol is that it maintains a *control bit* (0 or 1) that is used as a tag to identify successive messages as they travel from sender to receiver. Each channel is assumed to treat messages based on First In First Out (FIFO) principle. The instantaneous reliability of a channel depends on its congestion levels and the presence of disturbances from the environment. The time to send a message

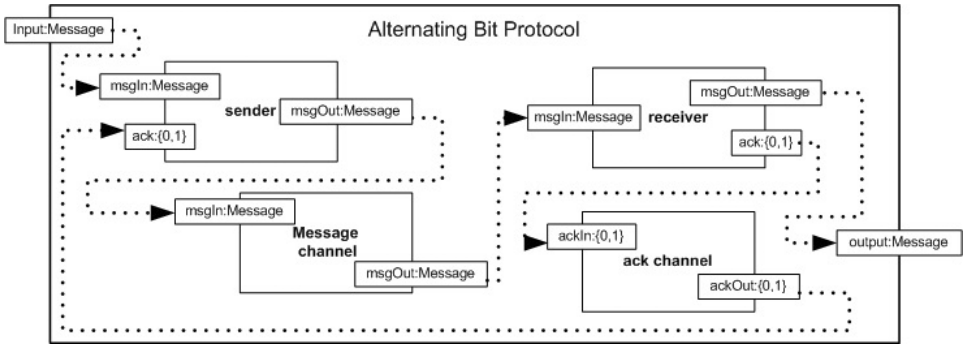


Fig. 9. Block diagram of the ABP.

or an ack to its destination depends on the channel's condition. It may also be duplicated or lost within the channel. The sender accepts a message from a device and forwards it with a control bit (i.e., 0 or 1) via the message channel to the receiver. It then enters a waiting session expecting an ack containing the control bit of the last message sent. If ack is not received after a specified waiting period, it resends the message with the same control bit on the assumption that the message is lost in the channel. This process is repeated until the expected ack is received when the next message in the buffer (if any) is sent with a control bit which is the binary complement of that of the last message sent.

When a message arrives at the receiver, its (message's) control bit is extracted and sent in an ack via the ack channel to the sender while the message itself is delivered to the appropriate destination device. In addition, the receiver keeps the control bit of the last message received and compares it with those of subsequent messages until a message with its binary complement arrives; any message that arrives with the same control bit as the previous is assumed to be a duplicate; hence, it is only acknowledged but not delivered.

The HiLLS model in Fig. 10 presents the hierarchical composition of the ABP's components as well as their I/O interfaces. *ABProtocol* has four components: *sender* and *receiver* are instance of HSystems Sender and Receiver, respectively, while *msgChannel* and *ackChannel* are instances of HSystem CommLine[T] with T as *Message* and *Integer*, respectively. Each of sender and receiver has a complex attribute, *buffer*, which is a queue of instances of HClass Message.

Figure 10 is a black box view of the specification, we present the internal details of each component in the following subsections.

6.1. *Receiver and message*

The Receiver HSystem and Message HClass are as shown in Fig. 11. Message declares two attributes, *header* and *content* to store the message's control bit and content, respectively. Five operations are also defined for manipulating and

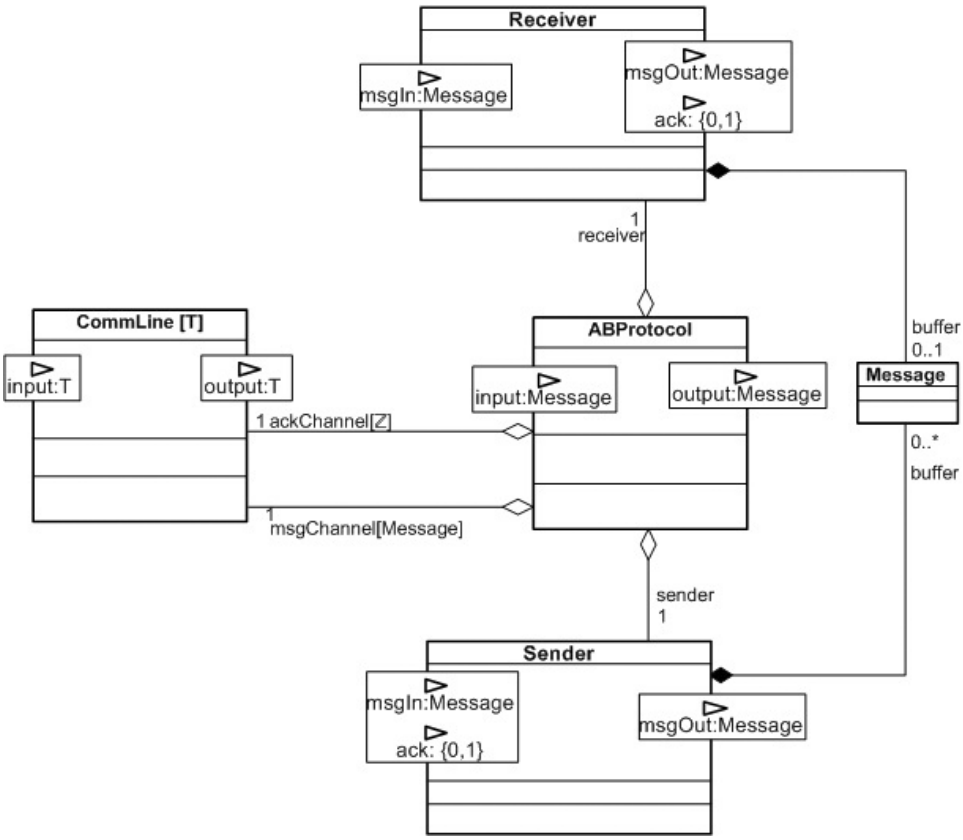


Fig. 10. HiLLS' structural hierarchies of the ABP.

assessing the attributes. It is important to state here that every operation with a defined type has an implicit output variable, *out!*, having same type as the operation. The *out!* variable serves a similar function as the *return* statement in Java and C++.

Receiver has a variable *buffer* which is a queue of messages with maximum length of 1 as depicted by the containment reference from Receiver to Message. It has a second state variable, *flag* with domain {0,1} as prescribed by the constraints. The global variable, *indicator* models the light indicator as an activity that manifests the instantaneous states of the system to an observer in real time. Receiver has an input port, *msgIn* of type Message and two output ports, *ack* and *msgOut* of types Integer and Message, respectively. Messages are received in *msgIn*, acknowledged through *ack* and delivered (without duplicates) to the target device through *msgOut*.

The operations defined are as shown in the third compartment. *setFlag* extracts the control bit of the message in the buffer and assigns it to the flag variable

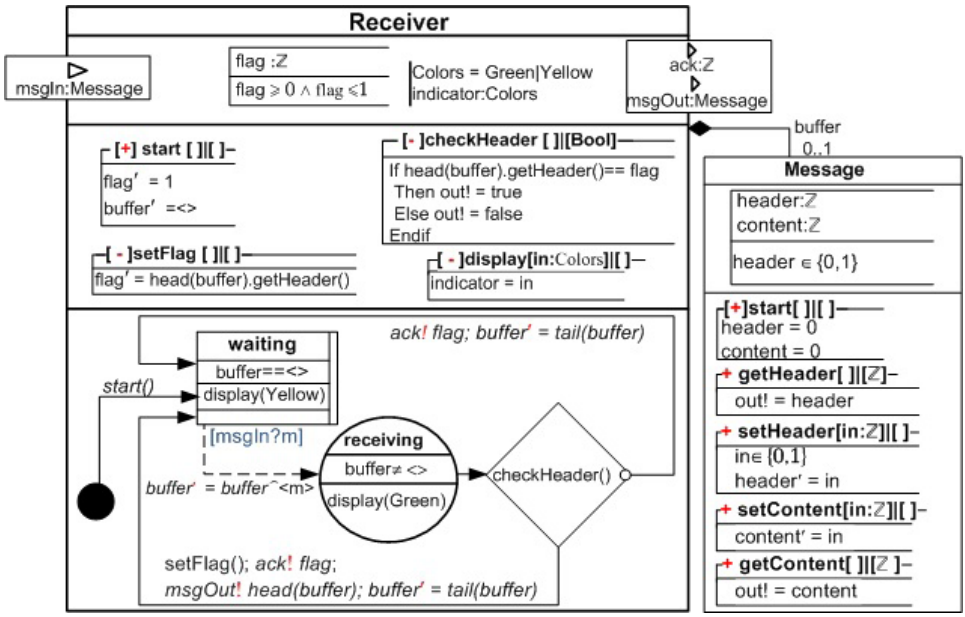


Fig. 11. Receiver HSystem.

checkHeader checks whether the control bit of the message in the buffer is equal to the current value of flag or not.

Receiver's behavior is modeled by the configuration transition diagram in the fourth compartment. It has two configurations: *waiting* and *receiving*; recall from Sec. 4.2 (Fig. 4) that a passive configuration is denoted by a four-compartment rectangle with a vertical strip on its right edge and a transient configuration is denoted by an oval shape. Configuration *waiting* is assumed when $buffer == \langle\rangle$; otherwise (i.e. when $buffer \neq \langle\rangle$), the *receiving* configuration is assumed. From the specification, the initial configuration of any object of Receiver is *waiting*. Upon assumption of each configuration, the activity function invokes the display operation which displays the specified color of light indicator. For example, the yellow indicator is displayed during the *waiting* configuration.

Since *waiting* is a passive configuration, the system remains in this state until a message m received at the input port *msgIn* triggers an external transition to the *receiving* configuration. The computation accompanying the transition adds the received message to buffer thereby satisfying the property of the target configuration. The prime, ´, decoration on buffer in the computation indicates its final state after the transition. The *receiving* configuration is transient, hence, an internal transition occurs automatically; the first computation invokes the *checkHeader* operation to determine whether the received message, m (i.e. the content of *buffer*), has the same *header* as the current value of *flag*. Recall that the *flag* stores the header bit of the last message received, so if *checkHeader* returns *true*, it means m

is a duplicate of the previous one; thus, only acknowledgment is sent to sender by taking the path described by the upper arrow to *waiting* configuration. If *checkHeader* returns *false*, then m in a new message; therefore, it is acknowledged and delivered as described by the operations accompanying the lower path to the *waiting* configuration. Note that delivery is achieved with the output on the *msgOut* port i.e., $msgOut!head(buffer)$.

6.2. *Communication line* (*CommLine[T]*)

The communication channel is shown in Fig. 12. It is a generic HSystem with parameter T denoting the type of message that may pass through it as indicated by the types of the input and output ports and state variable *buffer*. Therefore, the message and ack channels reuse this generic specification by substituting T with the types Message and Integer, respectively (see Fig. 10).

CommLine has two state variables: *buffer*, a sequence of T and *c_level*, an integer with constraints as shown in the figure. *c_level* holds the instantaneous coefficient of the level of congestion in the channel; we assume that its value may be between 1 and 4 and is determined by the external factors of the channel's environment, so we do not give further details about how the values are obtained in this model for simplicity.

CommLine starts in the initial configuration *waiting*; it remains in this configuration until an input, *msg*, is received on port *input* which triggers and external transition to the *sending* configuration. The *computation* accompanying the transition invokes the *receive* operation with *msg*; the operation simply adds *msg* to the channel's *buffer*.

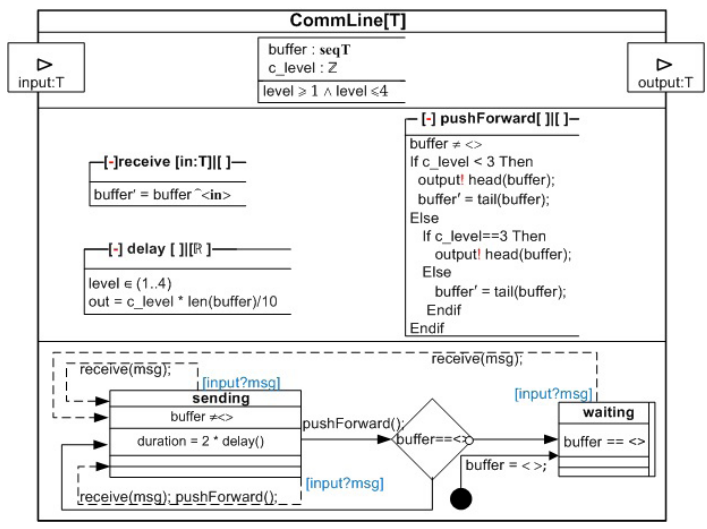


Fig. 12. Channel HSystem.

Table 2. The pushForward operation of CommLine.

Congestion level (c_level)	Computations	Results
$c\_level < 3$	1. Output head of buffer 2. Remove head of buffer	Successful delivery. No duplicate
$c\_level = 3$	Output head of buffer	Successful delivery with Duplicate message
$c\_level > 3$	Remove head of buffer	No delivery. Duplicate message

The time to push a message (or ack) through a channel is not deterministic; it depends on the level of congestion in the channel. therefore, the *sojournTime* of the *sending* configuration is defined by the expression $duration = 2 * delay()$. The *delay* operation returns a real value that is calculated based on the instantaneous level of congestion and length of the buffer. At the expiration of the generated sojourn time, an internal configuration transition occurs which invokes the *pushForward* operation. Operation *pushForward* is explained in Table 2; depending on the level of congestion, *c_level*, the message at the head of the channel's buffer is correctly delivered (without duplicate) to its destination, lost or duplicated. After the execution of *pushForward*, a target configuration is chosen between *waiting* and *sending* depending on whether *buffer* is empty or not. If an input is received just when the sojourn time expires, an confluent transition occurs (see bottom of sending configuration) which first adds the received message to the buffer before invoking *pushForward* and the target configuration will be *sending*. If the input is received before sojourn time expires, an external transition occurs as described on top of the sending configuration.

6.3. *Sender*

Figure 13 presents the HiLLS specification of the ABP's sender. We believe that the explanations provided previously for *Reveriver* and *CommLine* is sufficient to help the reader understand substantial part of the *Sender* specification. Thus, we provide details on only those features that are not yet encountered in the previous specifications.

The configuration *active* is a composite configuration with two sub-configurations: *sending* and *waiting*. The property, $buffer \neq \langle\rangle$ of *active* depicts the predicate that is common to all its sub-configurations. Therefore, the complete predicate properties of *sending* is $buffer \neq \langle\rangle \wedge head(buffer).getHeader() \neq flag$ while that of *waiting* is $buffer \neq \langle\rangle \wedge head(buffer).getHeader() = flag$. The sojourn time specification, $duration = \eta$, of *active* depicts that its actual sojourn time at any instant is that of its active sub-configuration. It is important to note that a composite configuration is nothing more than a logical clustering of configurations with some common properties.

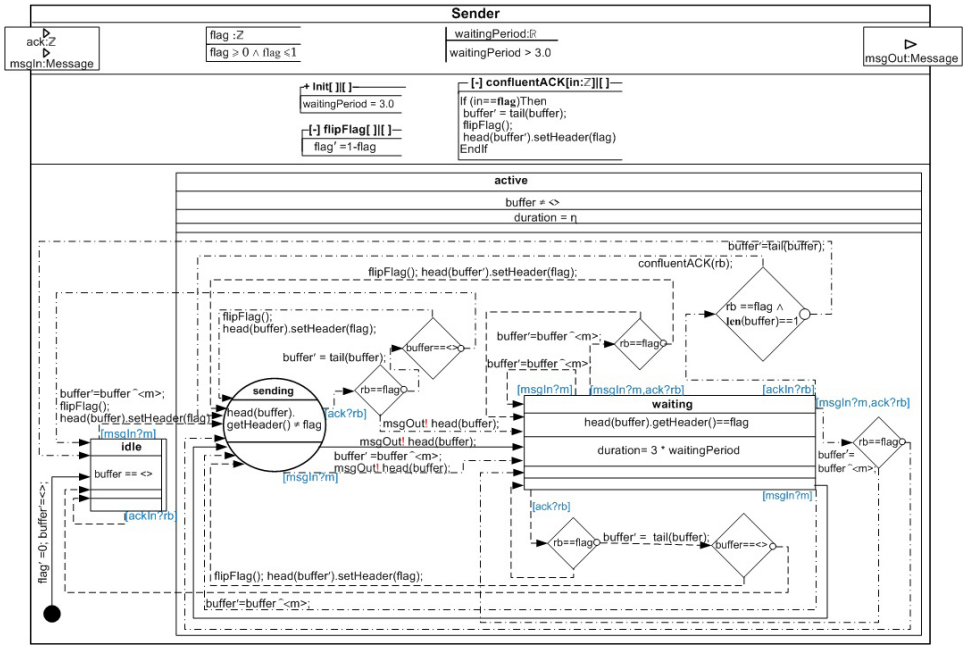


Fig. 13. Sender HSystem.

6.4. *ABProtocol*

The ABProtocol is described in Fig. 14. It defines an input and an output port each of type Message. The state variables are described by the *hcomponent* references *sender*, *receiver*, *ackChannel* and *msgChannel* as shown in Fig. 10. The predicate part of the state schema defines the invariants for permanently coupled ports. The couplings are realized by invoking the *connect*() operation during the initialization of the starting configuration. The first coupling specification, $sender \cdot msgIn =$

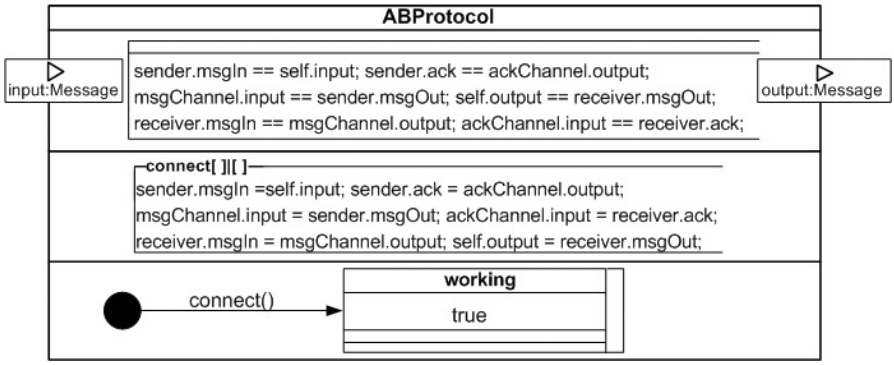


Fig. 14. ABProtocol HSystem.

$self \cdot input$, is an external coupling that connects an input port, *input*, of *ABProtocol* (as source) to an input port, *msgIn* of component *sender* (as target). Similarly, couplings $sender \cdot ack = ackChannel \cdot output$, $msgChannel \cdot input = sender \cdot msgOut$, $receiver \cdot msgIn = msgChannel \cdot output$ and a $ckChannel \cdot input = receiver \cdot ack$ are all internal couplings between peer components of ABProtocol. Only one passive configuration, *working*, is specified which does not change. This implies that the composed model does not add any extra behavior to that which is defined by the interactions between its components. The case is, however, slightly different in dynamic structured system where the modeler can specify multiple configurations with different coupling relations as their properties. The dynamic structure feature is not discussed further in this paper, we refer the reader to Ref. 14 for more details on modeling dynamic structure systems with HiLLS.

6.5. *Mathematical specification of the Message HClass*

The subsequent subsections of this section present the DEVS models derived from the HiLLS specification of the ABP presented in the last subsection. Since the models have a lot of artifacts in common, we think it would be sufficient to show the derivation of an atomic DEVS and a coupled DEVS.

DEVS does not provide an explicit syntax for specifying objects mathematically. For the sake of clarity, we represent Message as a mathematical object in the form:

$$Message = \langle V, F \rangle$$

with V and F as sets of variables and operations, respectively. Therefore, from the specification of Message in Fig. 11, we derive V and F as:

$V = \{(header, \mathbb{Z}), (content, \mathbb{Z}) \,|\, header \in \{0, 1\}\}$ and
$F = \{getHeader, setHeader, getContent, setContent\}$.
$getHeader : Message \to \mathbb{Z}$ returns the header/control bit of a message,
$setHeader : \mathbb{Z} \to Message$ sets the header/control bit of a message,
$getContent : Message \to \mathbb{Z}$ returns the content of a message and
$setContent : \mathbb{Z} \to Message$ sets the content of a message.

6.6. *Derived DEVS Model from receiver*

The Receiver in Fig. 11 has no hcomponent, hence it maps to an Atomic DEVS.

$$Rceiver_{DEVS} = \langle X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{conf}}, \lambda, \text{ta} \rangle.$$

$X = \{(msgIn, Message)\}$, $Y = \{(ack, \{0, 1\}), (msgOut, Message)\}$,
$S = \{((buffer,\ seq\, Message), (flag, \{0, 1\}), (phase, \{waiting,\ sending\})) \,|\, phase = waiting \Leftrightarrow buffer = \langle\rangle, phase = sending \Leftrightarrow buffer \neq \langle\rangle\}$.

The state variables $flag$ and *buffer* form a subset of S in DEVS and the complement is provided by the variable *phase* whose domain is the set of configuration

names specified in the system's behavior (see Fig. 11). The collection of the properties of each configuration (i.e., as specified in HiLLS) translates to the predicate part of S i.e., $phase = waiting \Leftrightarrow buffer = \langle\rangle$ implies that phase *waiting* is active if and only if predicate $buffer = \langle\rangle$ is true. Similarly, $phase = sending \Leftrightarrow buffer \neq \langle\rangle$ implies that phase *sending* is active if and only if predicate $buffer \neq \langle\rangle$ is true.

6.6.1. *Internal transition function*

The internal transition function $\delta_{\text{int}} : S \to S$ is:

$$\delta_{\text{int}}(sending, buffer, flag) = \begin{cases} (waiting, tail(buffer), flag) & \text{if } \theta_1 = flag, \\ (waiting, tail(buffer), \theta_1) & \text{if } \theta_1 \neq flag, \end{cases}$$

where $\theta_1 = head(buffer) \cdot getHeader()$.

Starting from a state in which $phase = sending$, the internal transition has a target in which $phase = waiting$ but the final value of variable $flag$ depends on the condition and if condition $\theta_1 = flag$. If the condition is satisfied, the value of $flag$ is same as before the transition (i.e., unchanged), otherwise, the final value is $flag' = \theta_1$. This piecewise equation is derived from the two paths from *sending* to *waiting* configuration in Fig. 11 by extracting all computations along the path except those meant for sending output events to some ports.

6.6.2. *External transition function*

The external transition function $\delta_{\text{ext}} : S \times \mathbb{R}^+ \times X^b \to S$ is:

$$\delta_{\text{ext}}((waiting, buffer, flag), e, m \in Message) = (sending, buffer\hat{a}\langle m\rangle, flag).$$

From a state in which $phase = waiting$ an input event $m \in Message$ on port $msgIn$ will trigger an external state transition into a state in which $phase = sending$. This equation is derived from Fig. 11 from the external transition $waiting \dashrightarrow sending$; the input event m in δ_{ext} is obtained from the trigger $[msgIn?m]$ in Fig. 11 while the domain (Message) of m is obtained from the type of port $msgIn$ in $[msgIn?m]$.

6.6.3. *Confluent transition function*

$\delta_{\text{conf}} : S \times X^b \to S$.
No confluent configuration is specified in Fig. 11, hence $\forall s \in S,\ \delta_{\text{conf}}(s) = \phi$.

6.6.4. *Output function*

The output function $\lambda : S \to Y^b$:

$$\lambda(sending, buffer, flag) = \begin{cases} \{(ack, flag), (msgOut, head(buffer))\} & if\ \theta_1 = flag, \\ \{(ack, flag)\} & if\ \theta_1 \neq flag, \end{cases}$$

where $\theta_1 = head(buffer) \cdot getHeader()$.

The output function is derived by traversing the internal and confluent configuration transitions in the HiLLS specification to extract their associated output expressions (if any). In this example, only the internal transition $sending \mapsto waiting$ (Fig. 11) may specify some output events. This transition specifies two alternative paths depending on whether condition "$\theta_1 = flag$" is satisfied or not; the corresponding output operations are derived in the λ function. There are two output expressions, $ack! = flag$ and $msgOut! = head(buffer)$ which translate into $(ack, flag)$ and $(msgOut, head(buffer))$, respectively.

6.6.5. *Time advance function*

$\text{ta} : S \rightarrow R^+ \cup +\infty$
$\forall buffer \in seqMessage \wedge flag \in \{0,1\} : \text{ta}(waiting, buffer, flag) = +\infty$ and $\text{ta}(sending, buffer, flag) = 0$. The time advance function is derived by mapping the label of each configuration to its sojourn time. The two configurations defined in Fig. 11 have pre-defined sojourn times denoted by their concrete representation as explained previously in Sec. 4.2; the passive configuration waiting has a pre-defined sojourn time of positive infinity while the transient configuration sending has a pre-defined sojourn time of zero.

6.7. *Derived DEVS model from ABProtocol*

ABProtocol (Fig. 14) translates to a Coupled DEVS model because its set of *hcomponents* is not empty.

$$ABProtocol_{DEVS} = \langle X, Y, D, \{M_d\}_{d \in D}, EIC, IC, EOC \rangle.$$

6.7.1. *Ports and sub-models*

$X = \{(input, Message)\}$,
$Y = \{(output, Message)\}$ and
$D = \{sender, msgChannel, ackChannel, receiver\}$.
X and Y are derived from the input and output interfaces respectively of Fig. 14. Set D is built from Fig. 10 by extracting the names of all *hComponent* references having ABPRotocol as source while $\{M_d\}_{d \in D}$ is the set of DEVS equivalents of the targets of such references. We have provided the DEVS equivalent of the HSystem referenced by $receiver \in D$ i.e., $Receiver_{DEVS}$. Therefore, $M_{receiver} = Receiver_{DEVS}$. Similarly, if the detailed specifications of other HSystems in Fig. 10 were given, then we would have $M_{sender} = Sender_{DEVS}$, $M_{ackChannel} = CommLine[\mathbb{Z}]_{DEVS}$ and $M_{msgChannel} = CommLine[Message]_{DEVS}$.

6.7.2. *Coupling relations*

We presented an overview of DEVS' meta coupling relations in Sec. 3.1. From Fig. 14, the DEVS coupling relations can be derived from the coupling predicates specified in the *connect*() operation as follows: The RHS of a coupling predicate

translates to the influencer system and port of the DEVS coupling expression while the LHS translate to the influenced system and port. For example, considering the first coupling predicate $sender \cdot msgIn = self \cdot input$; $LHS = sender \cdot msgIn$ and $RHS = self \cdot input$, the system reference of the LHS belongs to the group $d \in D$. The system reference of the RHS is $self$ (i.e., ABProtocol itself) and its port reference is also an input port; therefore, this coupling predicate translates to an element of EIC as shown below. The remaining five coupling predicates can be translated in the same manner to derive the elements of sets EIC, EOC and IC as:
$EIC = \{((self, input), (sender, msgIn))\}$,
$EOC = \{((receiver, msgOut), (self, output))\}$ and
$IC = \{((sender, msgIn), (msgChannel, input)), ((msgChannel, output), (receiver, msgIn)), ((receiver, ack), (ackChannel, input)), ((ackChannel, output), (sender, ack))\}$.

7. Discussions

We have classified existing DEVS-based visual modeling languages and tools, in Sec. 2, into three categories: UML-based, SysML-based and DSL-based. A common motivation of the UML-based and SysML-based categories is the possibility to advertise DEVS using modeling notations that are already established and accepted by a large community of users. In contrast, the DSL-based category is motivated by the flexibility offered by the chance of defining notations that are considered more "original" for DEVS concepts that may not be sufficiently described by the universal languages or the purposes for which they were created. In this context, HiLLS finds its place between the two extremes where existing concepts and notations are adopted (and augmented) where possible and new ones are introduced where necessary.

We compared, in Table 1 (see Sec. 2), how languages in the three categories model states and their time advances in atomic DEVS, their supports for reuse of component models in building hierarchical coupled DEVS models, and support for modeling complex input/output port and event types. Table 3 highlights some key features of HiLLS with regards to these four questions in comparison with related work; it also compares the kinds of model-based analysis methodologies to which models are amenable and the possibility of mapping the models to implementation tools in multiple programming platforms. The rest of this section is devoted to more detailed discussions of the points highlighted in the table.

7.1. *Support for reuse of specifications in coupled models*

HiLLS adopts the Software Engineering concept of composition between UML classes to model *hComponent* relationships between a coupled *HSystem* and its components. This allows to define only one specification for duplicate components in a coupled system or the referencing of a specification to model components in different coupled systems that is, an object of the *HSystem* is created as many

Table 3. Comparison of HiLLS with other DEVS-based visual modeling languages.

Formalisms	Model reuse	Time advance	State specification	Complex port type	Target solutions			Target platform
					Sim	FA	Enact	
eUDEVS	×	fixed	enumerated[a]	×	✓	×	×	generic
SysML-based	✓	fixed	enumerated[a]	×	✓	×	✓	generic
CD++	×	fixed	enumerated[a]	×	✓	×	×	C++
DEVS Diagram	×	fixed	structured[b]	×	✓	×	×	generic
DDML	×	fixed	structured[b]	×	✓	×	×	custom
MS4 Me	✓	fixed	structured[b]	✓	✓	×	×	Java
HiLLS	✓	dynamic	structured[b]	✓	✓	✓	✓	generic

Note: Sim ↦ Simulation, FA ↦ Formal Analysis, Enact ↦ Enactment.
[a]Finite set of arbitrary state identifiers called phases.
[b]State variables and a finite set of phases; a phase is a set of states that satisfy some conditions.

times as the modeler wants. This feature is typified in Fig. 10 with two references *msgChannel* and *ackChannel* to the *CommLine* to create the message channel and acknowledgment channel respectively of the *ABProtocol*. Both *msgChannel* and *ackChannel* have the same structure and behavior as specified in *CommLine*; the only difference is the types of messages being transmitted and this is specified by the passing the types *Message* and $\mathbb{Z}$ to the generic parameters of the former and latter, respectively.

A similar approach is adopted by formalisms in the SysML-based category where systems are described by *Block diagrams* and composition relationships are modeled by composition references between components with the sub-models as the targets. We, however, propose a different approach to describe the coupling between ports in a coupled system. While the SysML-based languages use a separate IBD solely for coupling specifications, we specify as part of the properties of the configurations.

MS4 Me also supports model reuse through the creation of multiple *instances* of an entity in an SES decomposition of a system. This is achieved by embedding an 'underscore' in the instance name with the entity's name at the right of the underscore. For example, in the example presented previously where *msgChannel* and *ackChannel* were created from *CommLine*, the instances would be described in MS4 Me's SES as *msgChannel_CommLine* and *ackChannel_CommLine*; the underscores in the names indicate that each of them is an instance of an existing entity *CommLine*. Another strength of the SES that is worthy of note here is that it allows for the specification of a family of hierarchical models rather than a single composition. It offers the concept of decomposition that breaks a complex system into simpler hierarchical modules called components that are coupled together through the specification of message flows between their I/O ports. The specialization feature allows for the specification of multiple kinds of certain components so that the modeler can explore different combinations of alternatives offered by the specialization through a process called pruning. These vital features are, however, available only in the textual scripting interface usually reserved for use by

the modeling expert; maybe it would also be beneficial to incorporate the features into the graphical modeling interface provided for the domain experts. We think the UML composition references adopted for this reuse purpose in HiLLS and the SysML-based formalisms is a well-known concept and does not require much efforts to learn or understand even by a novice.

7.2. *Time advance specification*

To our knowledge, in existing DEVS-based visual modeling languages, time advance is usually modeled as a constant real value associated with the state/phase. We think it would be reasonable to be able to define expressions in terms of some state variables for on-the-run generation of accurate time advance in some complex systems. For instance, let us consider the *CommLine* model (see Fig. 12) in the ABP example presented in the previous section; it would be difficult to specify a definite value of time advance for the *sending* configuration at modeling time because the time to transmit a message depends on the congestion level of the medium which varies with the number of messages in the channel and some other environmental variables. HiLLS allows the modeler to define an *expression* in terms of system's variables and parameters that may be executed to generate the sojourn time of a configuration during the execution of the simulation; hence, we were able to define the sojourn time of the configuration as $duration = 2 * delay()$ where $delay()$ is a call to an operation that generates the instantaneous value of the time to transmit a message from a source to destination through the medium.

7.3. *Support for structured state specification*

By structured state specification, we mean the description of a system's states in terms of the instantaneous values of state variables as against the enumeration of identifiers that do not explicitly depend on values of other variables as used in many DEVS-based formalisms. HiLLS supports the structured description of the *states* of complex systems through the concept of *configuration*, a high-level abstraction of a unique partition of the state space that defines a predicate (on state variables), activities and sojourn time (time advance) expressions that are fulfilled by all states in the partition. Other formalisms that provides means of structured state specification are DEVS Diagram, DDML and MS4 Me. An extra advantage offered by HiLLS in this regards is that the declaration of state variables and specification of configurations are done using concepts of *schema* and *predicate* adopted from Object-Z (see HiLLS' metamodel in Sec. 4) which naturally provides amenability to rigorous logical evaluation of the partitioning of the state space for consistency and mutual exclusiveness.

7.4. *Support for complex attributes and port and event types*

The capability to describe an *object* as an *HClass* is supported in the HiLLS' abstract and concrete syntax. In addition to primitive data structures, a modeler

can use the HiLLS' HClass to accurately describe complex attributes and input objects. This is also exemplified in the ABP example (Figs. 10–14) where the components have complex variables such as *buffer* which is a sequence of messages and I/O ports of type *Message*. We used the HiLLS' HClass to model the *Message* object with its attributes and operations to manipulate and access them. This feature is rarely supported in DEVS-based visual modeling languages, hence, most tools resort to programming languages to complete this part.

7.5. *Target solutions*

We stated in the beginning of this paper that HiLLS is a multi-semantics formalism with amenability to computational analysis using simulation, FM and enactment. Therefore, HiLLS seeks to make a unique contribution to model-driven system analysis in that models can be subjected to rigorous logical analysis to identify and resolve subtle logical issues in the specification before proceeding to simulation and/or enactment. This would help in the accreditation of models as well as boost user's confidence in the simulation and enactment traces. Another important advantage of making one model amenable to formal reasoning in multiple analysis contexts is that it will foster multi-disciplinary collaborations and tool integration using appropriate model-driven technologies.

7.6. *Target implementation platform*

Some of the DEVS-based formalisms discussed in this paper are strictly targeted at simulators and solvers built in specific programming platform. for instance, MS4 Me is specifically targeted at Java; in fact, the modeler must be conversant with the language as Java codes in tagged blocks must be embedded in the enhanced FDDEVS and SES specifications to implement the logics of transition and output functions and other miscellaneous functions in the model. Similarly, CD++ requires the modeler to have some programming skills in C++ in order to embed logic codes within the model to complete the C++ code generated from the model. DDML gives the modeler the flexibility to choose the target programming platform and then embed logic codes in the chosen language within the model for code synthesis.

A common argument in support of the need for embedding program codes within model specifications is that it offers the flexibility of completely describing complex algorithms. To pave the way for a more complete model engineering with DEVS, HiLLS captures the complexity of algorithms by combining first-order logic (Z schemas) and flowchart concepts (elements of the concrete syntax) to describe the dynamics of systems. Therefore, algorithms are described in a generic format — independent of any programming platform — so that program codes can be generated for any Object-Oriented programming language. The bases for refinement of Z specifications to program codes via intermediate pseudo-codes have been demonstrated by the authors of Refs. 39–41.

8. Conclusions

We have presented HiLLS, a graphical formalism for DESs. it combines concepts from System Theory and Software Engineering to provide an expressive syntax to model systems for exhaustive computational analysis through simulation, logical analysis and enactment. This paper is the first of the series to present HiLLS' suitability in the different domains of computational analysis; the present paper presents HiLLS in the simulation perspective. We presented the abstract and concrete syntax and its semantics in DEVS. Therefore, in this context, HiLLS is considered as a visual modeling language for DEVS with additional support for logical analysis and enactment to boost the simulationist's confidence in the simulation results.

We have shown that HiLLS is well positioned for model engineering with DEVS by comparing some of its features with the state-of-the-art in visual languages for DEVS. Notably, it supports the specification of structured states of complex systems, expressions for on-the-run generation of time advances that cannot be accurately determined at modeling time, model reuse, modeling of complex attributes and input/output port types, and suitability for integration with simulators built in multiple programming platforms. Some of these features are often supported only at programming level in most visual languages based on DEVS. A case study was provided in the paper to showcase system modeling with the language.

For further research, the construction of an editor for HiLLS is in progress; when it is completed, we expect to be able to automatically generate simulation codes for different DEVS-based simulation, FM and enactment tools. Our ambition is that, by making HiLLS a highly expressive, communicable and usable language for DESs in general, we can actually make the language the pivot to integrate disparate methodologies for model-based systems engineering such as simulation, logical analysis and enactment within one framework through MDE. One of the motivations for such integration of analysis techniques is that we can derive maximum utilities — exhaustive model analysis — from the efforts in building models. Moreover, we expect that the discussion of the state-of-the-art of visual languages for DEVS discussed in this paper will stimulate some research interests into providing more convenient means of model engineering with DEVS by harnessing modern Software Engineering techniques.

References

1. Bruel J.-M., Combemale B., Ober V., Raynal H., MDE in Practice for computational science, *Proc. International Conf. Computational Science*, Reykjavík, Iceland, pp. 669–669, 2015.
2. Carson II J. S., Introduction to modeling and simulation, *Proc. 36th Conf. Winter Simulation*, Washington, DC, USA, pp. 9–16, 2004.
3. Maria A., Introduction to modeling and simulation, *Proc. 29th Conf. Winter Simulation*, Washington, DC, USA, pp. 7–13, 1997.

4. Clarke E. M., Wing J. M., Formal methods: State of the art and future directions, *ACM Comput. Surv.* **28**(4):626–643, 1996.
5. Wing J. M., A specifier's introduction to formal methods, *Computer* **23**(9):8–22, 1990.
6. Woodcock J., Larsen P. G., Bicarregui J., Fitzgerald J., Formal methods: Practice and experience, *ACM Comput. Surv.* **41**(4):19, 2009.
7. Hoare C. A. R., Misra J., Leavens G. T., Shankar N., The verified software initiative: A manifesto, *ACM Comput. Surv.* **41**(4):22, 2009.
8. Fisher M. S., *Software Verification and Validation*: *An Engineering and Scientific Approach*, Springer Science & Business Media, USA, 2007.
9. Dowson M., Fernström C., Towards requirements for enactment mechanisms, in Warboys B. C. (ed.), *Software Process Technology*, Lecture Notes in Computer Science, Vol. 772, Springer Verlag, Berlin, 1994.
10. Aliyu H. O., Maïga O., Traoré, M. K., A framework for discrete event systems enactment, *Proc. 29th European Simulation and Modeling Conf. — ESM'2015*, Leicester, United Kingdom, pp. 149–156, 2015.
11. Rumbaugh J., Jacobson I., Booch G., *Unified Modeling Language Reference Manual, The 2nd Edition*, Pearson Higher Education, 2004.
12. Zeigler B. P., Praehofer H., Kim T. G, *Theory of Modeling and Simulation*: *Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press, San Diego, 2000.
13. Vangheluwe H. L., DEVS as a common denominator for multi-formalism hybrid systems modelling, *Proc. Computer-Aided Control System Design*, *CACSD 2000*, *IEEE Int. Symp.*, pp. 129–134, 2000.
14. Maïga O., Aliyu H. O., Traoré M. K., A new approach to modeling dynamic structure systems, *Proc. 29th European Simulation and Modeling Conf. — ESM'2015*, Leicester, United Kingdom, pp. 141–148, 2015.
15. Risco-Martín J. L., Jesús M., Mittal S., Zeigler B. P., eUDEVS: Executable UML with DEVS theory of modeling and simulation, *Simulation* **85**(11–12):pp. 750–777, 2009.
16. Zinoviev D., Mapping DEVS models onto UML models, *DEVS Symp.*, *Spring Simulation Multiconf.*, San Diego, CA, USA, pp. 101–106, 2005.
17. Nikolaidou M., Dalakas V., Kapos G. D., Mitsi L., Anagnostopoulos D., A UML2. 0 profile for DEVS: Providing code generation capabilities for simulation, *SEDE*, pp. 314–319, 2007.
18. Nikolaidou M., Dalakas V., Mitsi L., Kapos G. D., Anagnostopoulos D., A sysml profile for classical devs simulators, *Software Engineering Advances*, pp. 445–450, 2008.
19. Huang E., Ramamurthy R., McGinnis L. F., System and simulation modeling using SysML, *Proc. 39th Conf. Winter Simulation*, pp. 796–803, 2007.
20. Friedenthal S., Moore A., Steiner R., *TA Practical Guide to SysML*: *The Systems Modeling Language*, Morgan Kaufmann, 2014.
21. Wainer G., CD++: A toolkit to develop DEVS models, *Softw.*: *Pract. Exp.* **32**(13):1261–1306, 2002.
22. Wainer G., Christen G., Dobniewski A., Defining DEVS models with the CD++ toolkit, *Proc. ESS*, pp. 633–637, 2001.
23. Song H. S., Kim T. G., DEVS diagram revised: A structured approach for DEVS modeling, *Proc. European Simulation Conf.*, pp. 94–101, 2010.
24. Traoré M. K., A graphical notation for DEVS, *Proc. 2009 Spring Simulation Multiconf.*, Society for Computer Simulation International, Article No. 162, 2009.

25. Maïga O., Ighoroje U. B., Traoré M. K., DDML: A support for communication in M&S, *Proc. IEEE 21st Int. Workshop on Enabling Technologies*: *Infrastructure for Collaborative Enterprises* (*WETICE*), IEEE, pp. 234–243, 2012.
26. Ighoroje U. B., Maïga O., Traoré M. K., The DEVS-driven modeling language: Syntax and semantics definition by meta-modeling and graph transformation, *Proc. 2012 Symp. Theory of Modeling and Simulation — DEVS Integrative M&S Symposium*, Society for Computer Simulation International, Article No. 49, 2012.
27. Zeigler B. P., Sarjoughian H. S., Guide to modeling and simulation of systems of system, in *Simulation Foundations*, *Methods and Applications*, Springer London, 2012.
28. Seo C., Zeigler B. P., Coop R., Kim D., DEVS modeling and simulation methodology with MS4 Me software tool, *Proc. 2013 Symp. Theory of Modeling and Simulation-DEVS Integrative M&S Symp.*, Society for Computer Simulation International, Article No. 33, 2013.
29. Zeigler B. P., Seo C., Kim D., System entity structures for suites of simulation models, *Int. J. Model., Simulat., Sci. Comput.* **4**(3):1340006, 2013.
30. Mittal S., Douglass S. A., DEVSML 2.0: The language and the stack, *Proc. 2012 Symp. Theory of Modeling and Simulation — DEVS Integrative M&S Symp.*, Society for Computer Simulation International, Article No. 17, 2012.
31. Hollmann D. A., Cristiá M., Frydman C., CML-DEVS: A specification language for DEVS conceptual models, *Simulat. Model. Pract. Theor.* **57**:100–117, 2015.
32. Efftinge S., Völter M., oAW xText: A framework for textual DSLs, *Workshop on Modeling Symp. at Eclipse Summit*, p. 118, 2006.
33. Schülz S., Ewing T. C., Rozenblit J. W., Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling, *Proc. Seventh IEEE Int. Conf. Workshop on Engineering of Computer Based Systems*, pp. 308–316, 2000.
34. Chow A. C. H., Zeigler B. P., Parallel DEVS: A parallel, hierarchical, modular, modeling formalism, *Proc. 26th Conf. Winter Simulation*, pp. 716–722, 1994.
35. Smith G, The Object-Z specification language, in Hinchey M. (ed.), *Advances in Formal Methods*, Vol. 1, Springer USA, 2000.
36. Spivey J. M., *Understanding Z*: *A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.
37. Jouault F., Allilaire F., Bézivin J., Kurtev I., Valduriez P., ATL: A QVT-like transformation language, *Companion to the 21st ACM SIGPLAN Symp. Object-Oriented Programming Systems*, *Languages*, *and Applications*, pp. 719–720, 2006.
38. Bartlett K. A., Scantlebury R. A., Wilkinson P. T., A note on reliable full-duplex transmission over half-duplex links, *Commun. ACM* **12**(5):260–261, 1969.
39. Spivey J. M., An introduction to Z and formal specifications, *Softw. Eng. J.* **4**(1):40–50, 1989.
40. Woodcock J., Davies J., *Using Z*: *Specification*, *Refinement*, *and Proof*, Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1996.
41. Cavalcanti A., Woodcock J., ZRCa refinement calculus for Z, *Formal Aspects Comput.* **10**(3):267–289, 1998.