An Integrated Modelling and Analysis Environment for Parallel DEVS

Bruno Barroca
McGill University, Canada
bbarroca@cs.mcgill.ca

Sadaf Mustafiz
McGill University, Canada
sadaf@cs.mcgill.ca

Hans Vangheluwe
Univ. of Antwerp, Belgium
McGill University, Canada
hv@cs.mcgill.ca

Simon Van Mierlo
Univ. of Antwerp, Belgium
simon.vanmierlo@uantwerpen.be

Ernesto Posse
Zeligsoft, Canada
eposse@zeligsoft.com

ABSTRACT

Visual modelling environments are becoming increasingly popular in the modelling and simulation domain. While the advantages of graphical languages are evident, the limitations restricting model analysis, portability, standardization also need to be addressed. Textual languages can prove to be beneficial in such cases giving users the opportunity of deploying formal verification means if desired, or to serve as a bridge to connect to external simulation or analysis environments.

In this paper, we introduce an integrated DEVS modelling, simulation, and analysis environment for modelling discrete event systems (DEVS), by using two textual notations: DEVSLang and DEVSPro. On the one hand, these languages facilitate both horizontal exchange of models (i.e., between the various existing DEVS environments), and vertical (i.e., between the tools within the integrated environment). On the other hand, they enable the complete verification of the action code included in DEVS models by means of syntactic checking, and static-semantics analysis.

Author Keywords

Parallel DEVS; HUTN; textual concrete syntax; analysis;

INTRODUCTION

Discrete EVent System Specification (DEVS) environments that allow both visual and textual modelling are highly desirable, with textual modelling allowing compact expressiveness at the level of the DEVS functions and expressions. Moreover, if the goal is to have an integrated DEVS environment that supports not only modelling, but also analysis and simulation, then textual notations can yet take a bigger role.

The use of textual concrete syntax to meet interoperability requirements has been advocated in [2]. As is evident from recent efforts [15, 7] in DEVS standardization, textual representations in a neutral language allow DEVS models to be easily ported between modelling and simulation environments (horizontal interoperability), or to integrate new tools within the environment (vertical interoperability).

In this work, we integrate editors, static analyzers and simulators in an integrated DEVS environment, to support modelling, analysis, and simulation of DEVS models. For this purpose, we introduce two textual languages in our DEVS framework (see Figure 1). Our framework is built on a graphical modelling and simulation environment for DEVS in AToMPM [10] which uses Python Parallel DEVS (PyPDEVS) [13] as the simulation engine. We have addressed the need for a textual concrete syntax with the respective *required* analysis support for PyPDEVS. The main contributions of this paper are outlined below.

- DEVSLang: We introduce a new restricted variant of the DEVS formalism that can be used to express DEVS models using a one-dimensional state representation. DEVSLang is expressive but compact which often means that it is also intuitive to use. It is to be adopted for the integration of a DEVS modelling environment with existing analysis mechanisms, and for interchange with other DEVS environments. We have integrated a neutral action code language in DEVSLang which is to be used to specify DEVS functions in a kind of pseudo-code, i.e., independently from any programming language environment.
- DEVSPro: We introduce a DEVS language based on DEVS theory allowing users to go beyond DEVSLang's one-dimensional state automatons to model atomic components in DEVS, while also using the same neutral action code language. This is especially needed in order to exchange existing DEVS models with other DEVS modeling environments, that (for instance) can enable further kinds of simulation and analysis.
- Static analysis of DEVS models: We built a DEVS static analyzer in order to verify the syntax and static-semantics of both DEVSLang and DEVSPro models.
- Integrated environment: We demonstrate our approach by building a DEVS modelling environment that integrates AToMPM (here used as a graphical modelling front-end)

SpringSim'15, April 13-16, 2015, Alexandria, VA

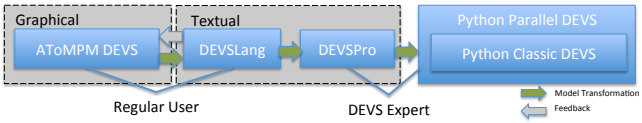

Figure 1: DEVS Formalisms

with several kinds of tools (ranging from analysis to simulation). The result is a hybrid DEVS modelling environment, where the users can use AToMPM in order to visually define DEVS Models while using embedded textual action code to specify the functions' behavior. The consistency and conformance of the defined DEVS models can then be analyzed using our DEVS static analyzer.

The rest of this paper is structured as follows: Following a brief background on Parallel DEVS, we introduce the DEVS formalisms used and proposed in our environment. We then elaborate on the concrete-syntax/grammar of the proposed formalisms (DEVSLang and DEVSPro) which is followed by a detailed section on the proposed analysis mechanisms (its aims and means). Finally, we present a comparison with related work in order to clarify our contribution, and discuss the future research trends and immediate developments in our integrated modeling and analysis environment.

BACKGROUND

The DEVS formalism was introduced to develop a rigorous basis for the compositional modelling and simulation of discrete event systems [17]. We briefly present the Parallel DEVS formalism, a variant of Classic DEVS, in this section.

A DEVS model is composed of both *atomic* and *coupled* components. An atomic component model describes the behavior of a reactive system. A coupled component model is the composition of several submodels which can be atomic or coupled. Submodels have *ports*, which are connected by channels defining a *transfer function* to translate output to input messages. Ports and channels allow a component model to receive and send signals (events) from and to other models. Parallel DEVS is closed under coupling, which means that coupled models can be nested to arbitrary depth.

An atomic model is formally defined as,

$$M =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

where *input set X* denotes the set of admissible inputs of the model such that $X = \times_{i=1}^{m} X_i$ with X_i denoting the admissible inputs on port i; *output set Y* denotes the set of admissible outputs of the model such that $Y = \times_{i=1}^{l} Y_i$ with Y_i denoting the admissible outputs on port i; *state set S* is the set of admissible sequential states; *internal transition function* δ_{int} defines the next sequential state depending on the current state, $\delta_{int} : S \to S$; *output function* λ maps the sequential state set onto an output bag, $\lambda : S \to Y^b$; *external transition function* δ_{ext} gets called whenever an *external input* ($\in X$) is received in the model, $\delta_{ext} : Q \times X^b \to S$ with $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ where e is the elapsed time; *time advance function ta* defines the simulation time the system remains in the current state before triggering its *internal transition function* such that $ta : S \to \mathbb{R}^{+}_{0,+\infty}$; and finally *confluent transition function* is called if both an internal and external transition collide at the same simulation time, replacing both functions such that $\delta_{conf} : S \times X^b \to S$.

THE MODELVERSE DEVS ENVIRONMENT

Our integrated modelling and analysis DEVS environment uses the Modelverse [12] as its model repository and model management facility. The Modelverse focuses on scalable and efficient model-management activities. It includes a built-in meta-modelled action language to give basic execution support to back-end operations as well as to allow symbolic transformation and analysis of equations and action code. The AToMPM models are stored in the Modelverse and can then be retrieved and edited by means of a human usable textual notation (HUTN). The HUTN [12] is a textual front-end for the Modelverse. It is a language with high reflexive powers such as the capability of dynamically (in memory) defining grammars and strong type systems (such as meta-models) upon which the syntax of models (such as the ones expressed in the user-defined AToMPM DEVS models) can be checked for syntax errors.

DEVS Formalisms

The proposed environment is composed of formalisms at four different levels:

- **AToMPM DEVS** A simulation environment for Parallel DEVS visual modelling, simulation, and debugging in AToMPM (A Tool for Multi Paradigm Modelling) [10]. The DEVS formalism requires each atomic DEVS component to be explicitly modelled with states;
- **DEVSLang** A user-friendly textual syntax conceptually similar to the AToMPM DEVS formalism: it enforces the usage of one-dimensional states (here called modes);
- **DEVSPro** A textual notation based on DEVS theory and at an abstraction level closer to programming languages to allow expert users to exploit the full power of DEVS;
- **Python Parallel DEVS** (PyPDEVS) [13] is a DEVS language grafted on the Python language with a matching simulator. It supports various features (such as tracing, checkpointing, and real-time simulation) and formalisms (classic DEVS, parallel DEVS, and dynamic structure DEVS).

Method

The AToMPM model (example model of a Producer-Consumer system is shown in Figure 2), along with the embedded action code, is first exported to DEVSLang to carry out syntax checks and static-semantic analysis. The mapping onto PyPDEVS is then facilitated using a code generator. The DEVSLang models need to be instrumented for visualization, in order to send feedback to and animate the source model at the AToMPM front-end. This also requires the graphical details of the model elements to be stored in the Modelverse and in the Python PDEVS models. The DEVSLang models can also be transformed to a DEVSPro model, if required, to enable models to be ported to different environments, or to make it possible for advanced users to adapt the models in

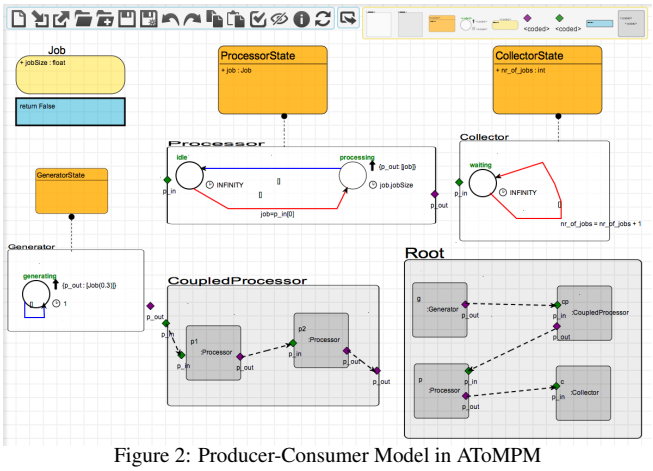

Figure 2: Producer-Consumer Model in AToMPM

an unrestricted manner (i.e., without having to use the one-dimensional state representation).

THE TEXTUAL LANGUAGES: DEVSLANG AND DEVSPRO

This section describes the representation of DEVS models using both DEVSLang and DEVSPro, while contrasting with their respective design goals.

DEVSLang

The DEVSLang textual syntax should primarily allow the specification of composite structures defined with coupled DEVS, and the specification of each atomic DEVS component along with its associated time advances, output function, transition functions, transfer functions, confluent functions, and port definitions. The events and state definitions should also be included as syntactic constructs in the textual model.

We now detail the proposed concrete syntax for DEVSLang by partially showing (due to space constraints), the corresponding DEVSLang model for the Producer-Consumer model (Figure 2). This is presented in Listing 1. An exporter in AToMPM is able to generate the DEVSLang textual models from the visual DEVS models.

Listing 1: Snippet of the Producer Consumer example in DEVSLang.

```
1  # State definitions
   ...
   statedef CollectorState(name, nr_of_jobs)
   ...
5  # Event definitions
   event Job(jobSize)
   ...
   atomic Processor():
           inports p_in
10         outports p_out
           mode ProcessorState('idle', job):
                   any -> ProcessorState('processing', p_in[0])
                   after infinity -> any
                   out nothing
15         mode ProcessorState('processing', job):
                   after job.jobSize -> ProcessorState('idle')
                   out {p_out: [job]}
           initial Processor('idle')
   ...
20 ## component Root needs to be defined
   coupled ProducerConsumer():
           instances:
                   g = Generator(a,b)
                   cp = CoupledProcessor()
25                 p = Processor()
                   c = Collector()
           connections:
                   from g.p_out to cp.p_in
                   from cp.p_out to p.p_in
30                 from p.p_out to c.p_in

   ...
   bottom:
35         sim = ProducerConsumer()
           sim.setVerbose(None)
           sim.setTerminationTime(100)
           sim.simulate()
```

A DEVSLang specification includes definitions for state definitions (as in line 3), events (as in line 6), atomic components (as from lines 8 to line 18), coupled components (as from line 21 to line 30), and two possible action blocks: an action block to be executed at the beginning of the simulation (i.e., the *top* block—not shown in Listing 1), and an action block to be executed at the end (i.e., the *bottom* block as shown from line 33).

As we can see inside each atomic component definition (line 11 to 14), DEVSLang mimics the notion of one-dimensional states as used in the AToMPM's visual syntax, with the notion of *mode*. Modes are connected by means of either external transitions (see line 12) or internal transitions (see line 13), in a grammar-like fashion: each transition is defined by means of a rule in the form $<left-hand\ side> \rightarrow <right-hand\ side>$, where the left-hand side is matching some condition on the component's non-modal state variables, and the right-hand side creates a new mode instance while allowing the creation and modification of the non-modal state variables, and passing them through that mode instance. In this example (see line 13), the external transition starts with the *any* keyword, which means that any event coming from any of the defined *inports* in the $Processor$ component will trigger this transition.

It is very easy to recognize an internal transition since they always start with the *after* (line 13) keyword, but do notice that the value *infinity* is actually the result of a partial definition of time advance function implicitly defined for the $Processor$ component. In DEVSLang, the time advance function for a given atomic component is implicitly defined by the collection of all the after conditions from all of the defined modes in that component. For instance, in a given component with two specified modes a and b, where mode a defines an internal transition in the form: $after\ c \rightarrow b$; and mode b defines a transition in the form: $after\ d \rightarrow a$; then it is mapped to the following time advance function for that component (in Python):

```
def time_advance:
        if(mode == a):
                return c
        if(mode == b):
                return d
```

where both c and d are of type $Float$ or $Infinity$.

The action code meta-model used in both DEVSLang and DEVSPro is shown in Figure 4. It demonstrates the high expressiveness of the action code language, bringing additional problems on a straight forward translation to timed automata.

DEVSPro

We now briefly present the DEVSPro language by means of a small example. The type model for DEVSPro (not shown here for space reasons) is very much like the class diagram of PyPDEVS, without the simulation-specific classes. The meta-model is based on DEVS theory, and hence the notion of *modes* (in the DEVSLang metamodel) is not present here.

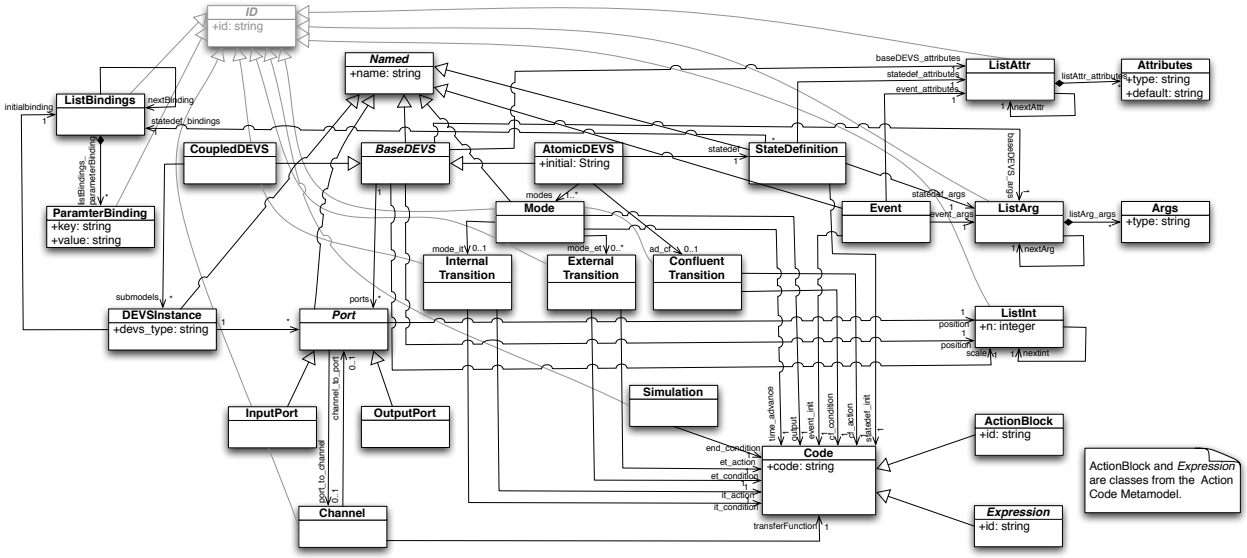

Figure 3: DEVSLang Metamodel

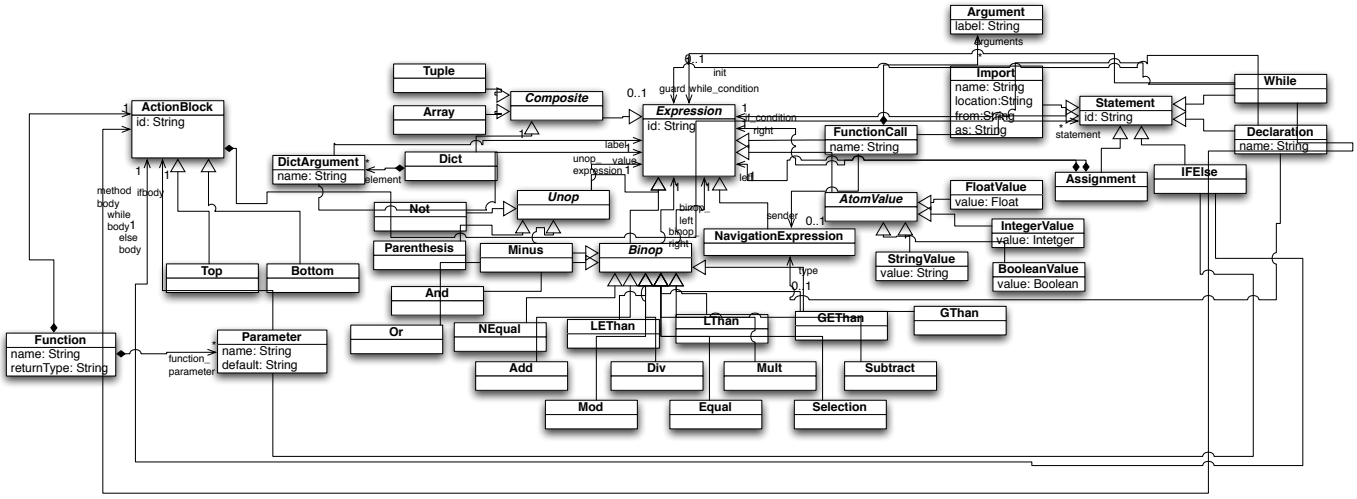

Figure 4: HUTN Action Code Metamodel used in both DEVSLang and DEVSPro.

The example model shown in Listing 1 (in parts) is mapped to DEVSPro here.

```
...
statedef CollectorState:
    constructor(name, nr_of_jobs=0):
        self.name = name
        self.nr_of_jobs = nr_of_jobs
...

atomic Processor:
    constructor():
        self.state = ProcessorState('idle')
        self.outports = {'p_out'}
        self.inports = {'p_in'}

    timeAdvance():
        if self.state.name == 'idle':
            return INFINITY
        else:
            if self.state.name == 'processing':
                return self.state.job.jobSize

    outputFnc():
        if self.state.name == 'idle':
            return {}
        else:
            if self.state.name == 'processing':
                return {self.outports['p_out']: [self.state.job]}

    intTransition():
        if self.state.name == 'processing':
            return ProcessorState('idle')

    extTransition(inputs):
        if self.state.name == 'idle':
            return ProcessorState('processing', job=inputs[self.inports['p_in']][0])

    confTransition(inputs):
        pass
...
coupled ProducerConsumer:
    constructor():
        self.outports = {}
        self.inports = {}
        self.submodels = {'g':Generator(), 'p':Processor(), /
                'cp':CoupledProcessor(), 'collector':Collector()}
        self.connectPorts(self.submodels['g'].outports['p_out'], /
                self.submodels['p'].inports['p_in'])
        self.connectPorts(self.submodels['p'].outports['p_out'], /
                self.submodels['cp'].inports['p_in'])
        self.connectPorts(self.submodels['cp'].outports['p_out'], /
                self.submodels['collector'].inports['p_in'])

bottom:
    sim = ProducerConsumer()
    sim.setVerbose(None)
    sim.setTerminationTime(100)
    sim.simulate()
```

ANALYSIS

The grammar specifications for each of the languages DEVSLang and DEVSPro enabled the generation of parsers that are able to recognize their sentences and assert its syntactic correctness. For instance, issue syntactic errors when some grammar rule is violated. However, these syntactic rules are not expressive enough to ensure consistency of the specified DEVS models, and more importantly their conformance to the original Parallel DEVS theory (presented in Section Background).

We will now observe the potential that both of the defined textual notations have to provide powerful mechanisms for consistency analysis, and describe our first steps in our integrated modeling and analysis environment to carry out analysis of DEVS models.

First of all, we need to distinguish between the three different kinds of analysis that are required in order to assure consistency of DEVS models as well as their conformance to the Parallel DEVS theory: contextual analysis, determinism analysis and symbolic state-space analysis. Each of these kinds of analysis require specialized data structures, and multiple passes over a given abstract syntax tree (AST) that represents a given parsed DEVS model.

Contextual Analysis in DEVSLang

The contextual analysis provided in the DEVSLang prototype is quite rich, and it builds on the syntactic constructs provided by the language, namely the type definitions, and explicit/syntactic division of the transition functions in pre- and post-conditions organized inside mode definitions.

Single state definitions in Atomic DEVS Component definitions. In order to maintain visual coherence with the AToMPM models, the DEVSLang users must specify state definition structures, and then reference them in the mode definitions of atomic components. The first contextual analysis check is to assure that only one state definition type is used per atomic component. This check is important so that we can safely instantiate DEVSLang models conforming to the DEVSLang metamodel which only allows one single state definition per component.

Communication between components in Coupled DEVS Component definitions. DEVSLang users while defining Atomic and Coupled components, are actually defining new types: $DevsType$. These types can then be used in order to be instantiated and connected in a given Coupled DEVS definition. This forms a tree of instantiations, where only the Coupled DEVS defined at the root level, is actually instantiated. The contextual analysis checks that every two connected instances belong to two existing types, and their respective $inports$ and $outports$ have compatible types. For instance, if a component A is outputting a vector of type T in $outport\ O_a$, and a component B is expecting a vector of type T' through $inport\ I_b$, and an instance of A is being connected to an instance of B by connecting O_a to I_b, then types T and T' should be the same (or at least compatible, if we are to support subtyping in the future). Notice however, that the original DEVS theory does not impose any restriction a-priori in the types of the events flowing through the ports. Therefore, with this restriction DEVSLang enforces a stronger type system, with less expressiveness but also more type safety, which actually means less possible programming errors, and hence produce DEVS models with more quality.

Variable references within a given scope. The DEVSLang users can define variables on particular scopes ranging from the state definitions, event definitions, function definitions, or components definitions (either atomic or coupled). Moreover, each variable is typed (basic types are supported such as Integers, Float, Strings and Booleans), and can be initialized with a specified default value. All of the scope definitions are globally available (e.g., a coupled component can refer and instantiate a given atomic component type). Also variables defined in state definitions and event definitions, or even functions inside components, are accessible outside a given scope definition using the dot notation. For instance, it is possible to access a given function named Z defined inside a component named B, simply by calling 'B.Z()'.

The contextual analysis forbids the access of both function and component parameters from outside their scope. Parameters are variables defined to be used internally in their own scopes. When the users define such variables, the variable remains accessible within its particular scope.

Consistent operation of user-defined functions and variables. The contextual analysis enforces the correct definition and calling of user-defined functions on general expressions, while computing the function return types and their composition within expression operations. For instance, in the expression `1 + c()`, the return type of function `c` is expected to be `integer` or at least `float` in order to be cast to `integer` and safely allow the + operation.

Consistent operation of DEVS functions and variables. The contextual analysis goes deep over the called user-defined functions and analyzes conditional branches, in order to check the return and parameter types of the time-advance, internal/external/confluent and output functions. It checks the allowed Read/Write variable access under the above functions (e.g., the elapsed variable in the precondition of the external function). The action code defined on the left-hand side of each transition definition, is analyzed in order to verify that the component's variables are being just read: this means that a match cannot change the variables of a given component, or of any other defined component.

The complete set of access rules is presented on Table 1. Here we detail for each function, the expected types for the pre-conditions and post-condition parts. Not explicitly represented in Table 1 is the additional requirements that: *(i)* there can be only exactly one internal transition function defined per mode in atomic components (as specified directly in the DEVSLang metamodel); and *(ii)* that the type of its result (written on the right-hand side) must always belong to the same state definition, and only the state variables are accessible for reading and writing.

On each external transition's pre-condition, the elapsed variable, all the state variables, and all the defined $inport$ vari-

	Internal Function		External Function		Confluent	Output
	Time Advance (Precon.)	Postcond.	Precond.	Postcond.	Function	Function
Return Type	Float	State Def.	Boolean	State Def.	State Def.	Dictionary(*)
State variables	Read Only	Read/Write	Read Only	Read/Write	Read/Write	Read Only
Inport variables	N/A	N/A	Read Only	N/A	Read Only	N/A
Outport variables	N/A	N/A	N/A	N/A	N/A	Write Only
Elapsed	N/A	N/A	Read Only	N/A	Read Only	N/A

Table 1: Contextual Analysis Rules for DEVSLang models. (*) where all labels belong to defined Outports in given Atomic Component

ables are accessible for read-only. Although in the post-condition, only the state variables are accessible for reading and writing.

Finally, in the output function only the $outport$ variables are accessible for writing, although we can read all of the state variables. The contextual analysis checks that the return type of the output function is a dictionary where all of its labels are names of defined $outport$ variables on that component. Warnings are issued if the analysis is inconclusive, and errors are issued if an inconsistency is detected. For instance, in the following output function:

```
out { nr_of_jobs = 5; return nothing; }
```

the contextual analysis issues the following message back to the AToMPM front-end: 'Error at line 13: nr_of_jobs state variable is not accessible for writing in output functions.'.

Contextual Analysis in DEVSPro

The DEVSPro syntax for DEVS functions does not offer any explicit distinction between the pre-condition and post-condition parts. Besides the predefined DEVS functions both atomic and coupled component definitions also include the $self.inport$, $self.outports$ predefined as sets of Strings. All the atomic components predefine the $self.elapsed$ variable of type $Float + Infinity$. The coupled components predefine the $self.submodels$ variable typed as a dictionary of $String$ x $DevsType$, as well as the predefined function $connectPorts$ used to connect DEVS instances together (and possibly with the containing $self$ coupled component), using their ports.

We reuse the analysis made for DEVSLang (as detailed in Table 1) for DEVSPro, with the exceptions that here we apply both the rules for pre- and post-conditions; and in the $outputFunc$, the checker allows read access for all the variables except the $self.inport$ and $self.elapsed$ variables.

Notice that in DEVSPro, the time advance function has an explicit representation, and there will be exactly one of the DEVS functions per Atomic Component definition (i.e., external transition function, confluent transition function, etc.), which if not defined, the simulator engine will just assume that their results are 'pass' (or 'Infinity' in the case of the time advance function).

Determinism Analysis in DEVSLang

Since DEVSLang uses a notion of states as in the state charts formalism, it will inherit some of its advantages in terms of usability, but also its problems. One of such problems is the non-determinism that can occur on the outgoing transitions of a given state (i.e,. on external transitions). Non-determinism is usually not a problem in analyzing languages like state charts, because timed-automata model checkers such as UPPAAL are also able to deal with non-determinism; and also while translating to a particular simulation engine such as PythonPDEVS, any non-deterministic behavior is automatically resolved: typically the first defined external transitions take priority over the subsequent ones.

This automated resolution brings up however a PIM/PSM gap between the DEVSLang and the resulting PythonPDEVS Code. For instance, during simulation, the modeller might get unexpected or even missing behavior in the non-deterministic branches. In order to reduce such a gap, the modelers should be able to automatically detect non-deterministic behavior, and/or correct it directly at the model level.

The non-determinism check on languages such as state charts is decidable, as demonstrated on the analysis of UML statecharts (with similar expressiveness) [8]. The analysis performs a constraint solving of the conjunction of any two outgoing guards from any given state: If the solver returns true then it means non-deterministic, because there exists a non-empty intersection of values from both guard conditions that evaluate to True.

However, the above mentioned analysis does not consider the usage of action code such as the one we are using in DEVSLang. Therefore, in the context of an external precondition expression consisting of several function calls, the determinism analysis must unfold the resulting guard expression from each function call, not as a value (that would be dynamic analysis), but instead as the a logical expression that represents the logical guard of that external transition.

RELATED WORK

Over the years, several extensions and variants of the DEVS formalism have been proposed [4]. One of our primary concerns in this paper was to address the specification of the DEVS functions. As seen in the observed DEVS language extensions, they are typically expressed using a textual syntax strongly influenced by the existing underlying programming language on which the simulation environments are grafted into (e.g., as seen in [1]). The need to provide a platform independent specification, with the capability to specify platform-neutral code statements (i.e., action code) that could then be rendered in any given programming language has remained unaddressed in most projects.

Yet another concern was the *interoperability* of DEVS environments along with the need for making available an integrated environment for DEVS. A common adopted solution for this problem (e.g. in [6, 11]) caused by the need for integrating tools in integrated modelling environments,

is the usage of meta-modelling, model transformations, and their combination with textual formats such as XML, in order to rapidly interchange models between simulation systems. However, little reference is made to the difficulty of expressing models in XML by regular users: the common argument is that this is a problem to be solved by suitable visual modelling environments.

In Table 2, we present a comparison of existing environments and approaches for DEVS modelling and analysis based on several criteria (listed in the first column of the table). A summary of such textual DEVS languages proposed for standardization is given in [14]. While various graphical front-ends for modelling DEVS are available (such as [9]), in our survey we have focused on approaches that allow formal specifications (in some form of textual concrete syntax) of DEVS models. It is to be noted that *static analysis* in Table 2 refers to syntax checking and type checking including static semantics analysis, and *symbolic analysis* refers to model checking or behavioral analysis.

Based on the related work we have seen, we conclude that the main challenge is to tackle the need for an integrated modelling environment, which composes tools for modelling, analysis, simulation, optimization and execution. In our work, we have attempted to adopt the stack approach by interacting different tools and associated meta-modelled languages, along with a textual platform-neutral action code that serves as the most appropriate format for model interchange between these tools, as well as across different modelling environments.

In this stack approach, static analysis tools should be able to validate the consistency of the defined DEVS during the editing phase, by sending correctness feedback to the editing tool, before allowing any further advanced analysis (e.g., by translating it to timed automata). This feature has also been addressed in our work.

Finally, to provide such advanced analysis, we must first include a way to express a property language based on a branching time temporal logics such as Timed CTL, which is an extension of regular branching time temporal logics with clock constraints. The integration of both kinds of analysis—either by scenario/case simulation, or by model checking's exhaustive search—in the system's design/modelling process is probably the biggest challenge to achieve in a DEVS integrated modelling environment.

CONCLUSION

We have built an integrated DEVS environment with the main goal of carrying out analysis of DEVS models. For our purpose, we have introduced the neutral textual languages, DEVSLang and DEVSPro, that can be used to model discrete event systems at two levels of abstraction: *(i)* one which gives users the capability to model atomic DEVS as state-automatons using a restricted language with an easy-to-use and readable notation; *(ii)* a second which gives users the flexibility and power to model systems as they would using any programming language (with or without using the notion of states). Action code in a human usable textual notation (HUTN) has been integrated in both languages. The syntax for specification of the action code remains the same over both languages. The action code can be used to represent the transitions functions, time advances, and output functions in our visual DEVS modelling environment. Hence, the visual models can be seamlessly transformed to our textual models to further extend or adapt the models, or to carry out analysis of the models.

The exported models are instrumented (with visualization details) in order to allow feedback from the static analyzer back to the graphical front-end. Based on the error and warning feedback, the user is required to make the necessary modifications to the action code in order to continue with the simulation.

The introduced static analyzer supports the following.

- Syntax checking: The action code is checked for basic syntactic errors according to either the DEVSLang or DEVSPro grammar definitions.
- Static Semantics: The static analyzer goes beyond traditional type checking (from programming languages), by, for instance, forbidding write access to the component's state variables inside an output function.

We plan on defining a conservative analysis procedure in order to automatically decide if a given DEVSLang model is for sure translatable to timed automata, so that further behavioral analysis can be provided (e.g., reachability analysis) on model checking tools such as UPAAL.

Further language enhancements in the proposed modeling environment will involve the pre-visualization at design time of the results of the DEVSPro constructors in the AToMPM front-end. It is interesting to note that it is possible to define parameters on DEVS components, such that a coupled component that accepts as parameter a number (value of type Integer) would be able to programmatically create a coupled system with a completely connected graph with the available connections between predefined DEVS components instances.

ACKNOWLEDGEMENTS

This work was partly funded by the Automotive Partnership Canada (APC) in the NECSIS project.

REFERENCES

1. Franceschini, R., Bisgambiglia, P.-A., Bisgambiglia, P., and Hill, D. DEVS-ruby: A domain specific language for DEVS modeling and simulation (WIP). In *TMS-DEVS, SpringSim*, SCS (2014), 393–398.
2. Goldschmidt, T., Becker, S., and Uhl, A. Classification of concrete textual syntax mapping approaches. In *Model Driven Architecture - Foundations and Applications*, vol. 5095 of *LNCS*. Springer, 2008, 169–184.
3. Hong, K. J., and Kim, T. G. DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems. *Inf. Softw. Technol. 48*, 4 (Apr. 2006), 221–234.

	DEVSpecL [3]	DEVSML [5]	DEVSML 2.0 [6]	DEVSRuby [1]	DEVSW [16]	SimStudio [11]	Our DEVS Framework
DEVS (C^1/P^2)	C	C	C,P	C	C	P	P
Meta-modelling/MT	*X*	XSL Transformation	✓	*X*	*X*	✓	✓
DEVS Variant	Theory	Theory	Finite Deterministic DEVS	Theory	FSA^3	Theory	FSA^3 and Theory
DEVS Levels	✓	✓	✓	DSL only	*X*	✓	✓
Analysis							
Static	✓	*X*	✓	*X*	*X*	*X*	✓
Symbolic	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Modelling							
Graphical	✓	Limited support	*X*	*X*	*X*	✓	✓(AToMPM)
Textual	✓	✓(XML)	✓(NLDEVS, DEVSML)	✓	✓(XML)	✓(XML)	✓(DEVSLang, DEVSPro)
Multi-view Synch	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Integrated Environment	✓	*X*	✓	*X*	*X*	✓	✓
Action code	✓	Lisp-like	JAVAML	*X*	*X*	✓(neutral)	✓(neutral)
Property Language	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Simulation							
Debugging	*X*	*X*	*X*	*X*	*X*	*X*	✓
Execution	✓	*X*	✓	✓	✓	✓	✓
Target Platform	C++, Java	Various (PyDEVS, DEVSJava)	DEVSJava	Ruby, Ruby C	DEVS-Scheme	Multiple	Python
Portability	✓	✓	✓	*X*	✓	✓	✓

Table 2: DEVS Modelling and Analysis Environments: A Comparison ([1]C: Classic DEVS, [2]P: Parallel DEVS; [3]FSA: Finite State Automaton)

4. Hwang, M. H. Taxonomy of DEVS variants. In *TMS-DEVS, SpringSim* (2014), 445–450.
5. Janoušek, V., Polášek, P., and Slavíček, P. Towards DEVS Meta Language. In *ISC* (2006), 69–73.
6. Mittal, S., and Douglass, S. A. DEVSML 2.0: the language and the stack. In *TMS-DEVS, SpringSim* (2012), 17.
7. Sarjoughian, H. S., and Chen, Y. Standardizing DEVS models: An endogenous standpoint. In *TMS-DEVS, SpringSim*, SCS (2011), 266–273.
8. Schwarzl, C., and Peischl, B. Static and dynamic consistency analysis of UML state chart models. In *MoDELS*, vol. 6394 of *LNCS*. Springer, 2010, 151–165.
9. Song, H. Infrastructure for DEVS modelling and experimentation. Master's thesis, School of Computer Science, McGill University, 2006.
10. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations* (2013).
11. Touraille, L., Traoré, M. K., and Hill, D. R. C. A model-driven software environment for modeling, simulation and analysis of complex systems. In *TMS-DEVS, SpringSim*, SCS (2011), 229–237.
12. Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., and Kühne, T. Multi-level modelling in the modelverse. *MULTI 2014 – Multi-Level Modelling Workshop Proceedings 1286* (2014), 83–92.
13. Van Tendeloo, Y., and Vangheluwe, H. The modular architecture of the Python(P)DEVS simulation kernel. In *TMS-DEVS, SpringSim* (2014), 387–392.
14. Wainer, G. A., Al-Zoubi, K., Dalle, O., Hill, D. R., Mittal, S., Martín, J. R., Sarjoughian, H., Touraille, L., Traoré, M. K., and Zeigler, B. P. *Standardizing DEVS model representation*. CRC Press, 2010, ch. 17, 427–458.
15. Wainer, G. A., Al-Zoubi, K., Mittal, S., Risco-Martín, J. L., Sarjoughian, H., and Zeigler, B. P. *An Introduction to DEVS Standardization*. CRC Press, 2010, ch. 16, 393–425.
16. Wang, Y., and Wang, L. An XML-based DEVS modeling tool to enhance simulation interoperability. In *Proceeding 14th European Simulation Symposium* (2002).
17. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation, Second Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.