Integrating a Neutral Action Language in a DEVS Modelling Environment

RESEARCH · SEPTEMBER 2015

DOI: 10.13140/RG.2.1.4591.4086

Bruno Barroca
McGill University
25 PUBLICATIONS **52** CITATIONS
SEE PROFILE

Sadaf Mustafiz
McGill University
22 PUBLICATIONS **138** CITATIONS
SEE PROFILE

Hans Vangheluwe
McGill University
159 PUBLICATIONS **1,539** CITATIONS
SEE PROFILE

Integrating a Neutral Action Language in a DEVS Modelling Environment

Bruno Barroca
McGill University
bbarroca@cs.mcgill.ca

Sadaf Mustafiz
McGill University
sadaf@cs.mcgill.ca

Simon Van Mierlo
University of Antwerp
simon.vanmierlo@uantwerpen.be

Hans Vangheluwe
University of Antwerp
hans.vangheluwe@uantwerpen.be

ABSTRACT

Visual environments for the modelling and simulation of complex, software-intensive systems are increasingly popular. While visual languages have many advantages, they may not be appropriate to render all details of a Discrete EVent system Specification (DEVS) model. Textual may be more appropriate, both to completely describe all details of a DEVS model (i.e., the content of transition and output functions), and to make the specification independent of the implementation platform (i.e., simulation implementation language).

In this paper, we propose two textual notations that are used as part of an integrated modelling and simulation environment for the Parallel DEVS formalism. Both notations allow the specification of DEVS functions by means of neutral action code. **DEVSPro** uses Python-like textual syntax and supports the full power of Parallel DEVS. From this neutral specification, simulator-specific code is synthesized. **DEVSLang** supports blended textual/visual modelling. It is more restricted in expressiveness to match the limited expressiveness of visual notations. For example, the sequential states in an Atomic model must be explicitly enumerated.

Visual DEVSLang models are transformed to their textual form in order to carry out syntactic and semantic checks. Possible detected errors are fed back to the visual modelling environment allowing the modeller to make changes directly in the source model. DEVSLang models are further translated automatically to DEVSPro models to allow for possible combination with DEVSPro models and subsequent analysis and simulation.

Keywords

Parallel DEVS; textual syntax; neutral action code

1. INTRODUCTION

As modern systems become more complex, they also become more and more difficult to describe and analyze. This is partly due to the high diversity in existing modelling and simulation (M&S) languages and environments. Examples of this diversity range from models (of systems and their environment), over (textual and/or visual) modelling tools, to efficient simulators. The success of M&S endeavours is increasingly dependent on our ability to re-use modelling and simulation solutions in a modular way.

Zeigler proposed the elegant DEVS formalism [21]. Models in different formalisms can be mapped onto it, making it a universal simulation "assembly language" [17]. As such, it enables the modular combination of different modelling languages, as well as their simulators. DEVS models are typically encoded in a programming language or at least using an API that embodies the essential aspects of DEVS models and their simulation semantics. Examples include ADEVS [9] which uses C++ and Python Parallel DEVS (PyPDEVS) [16].

Various solutions to standardize and support DEVS modelling and simulation interoperability have been proposed [8, 13, 3]. We postulate that modern M&S frameworks should be based on a usable and appropriate DEVS formalism with neutral (i.e., implementation platform/language-independent) action code. But what does this neutrality property really mean? Does it mean that its design should be in close relation with existing programming languages, so that it would be easier to reverse-engineer existing code and port it across platforms? Or should it instead be in close relation with Zeigler's DEVS formalism?

In this paper, we address these questions by building an Integrated M&S Environment (IM&SE) that integrates two distinct textual notations for the DEVS formalism, DEVSLang and DEVSPro, as presented in Figure 1. Each notation is designed for a different purpose, hence, each notation linguistically conforms to a different linguistic type model [1], simply referred metamodel throughout this paper.

While DEVSLang has restricted expressiveness in order to be generally fully renderable in a visual modelling environment, DEVSPro provides the full expressiveness of DEVS. On the one hand, modellers can then reuse their Statechart-like models (e.g., UML StateCharts or even timed automata) by easily porting them to DEVSLang, and then visually editing/debugging them in AToMPM. On the other hand, modellers can reuse existing libraries of DEVS models (e.g., ADEVS models) by also easily porting them to DEVSPro. By automatically translating DEVSLang models into DEVSPro models, our IM&SE enables the composition and integration of all the ported models in the development of new ones.

Similar to other solutions [19, 10], these languages use a neutral action code in order to express functions that we know by construction are indeed DEVS transition or output functions. Users can define several types of DEVS functions in order to not only reuse functionality but also to guarantee correctness. For instance, it is guaranteed that all of the defined output functions are only reading atomic component states, and cannot be composed with (i.e., internally call) other modifying functions, such as internal transition functions.

.

In order to study the integration of DEVSPro with DEVSLang and in order to develop a DEVS-IM&S environment, we built two meta-models, one for each formalism. The relationship between both languages is implemented by means of automated model transformation.

We demonstrate our approach by building a DEVS IM&SE: not only by (conceptually) integrating/merging our two formalisms, but also (in practice) combining an existing visual DEVS modelling environment built in our AToMPM (A Tool for Multi-Paradigm Modelling [12]), with an existing Parallel DEVS Simulator: PyPDEVS. In this IM&SE, modellers are able to visually define DEVS models while using embedded textual action code in order to specify each of the DEVS transition/output functions. The consistency and conformance of the defined DEVS models can then be analyzed using our DEVS syntax and static-semantics analyzer. The DEVS models can then also be debugged and simulated within the same environment [15].

The rest of this paper is structured as follows: Section 2 gives a brief theoretical background on Parallel DEVS. Section 3 introduces both formalisms while describing the pertinence and pragmatics of their textual notations (DEVSLang and DEVSPro). Section 4 discusses the static-semantics and contextual analysis that is required in order to produce models that are semantically conforming to the DEVS theory. Tool support for the analysis of the action code is described in Section 5. Finally, Section 6 presents a comparison with other related work, and Section 7 concludes by highlighting our contributions and discussing future work.

2. BACKGROUND

The DEVS formalism was introduced by Zeigler [21, 22], as a rigorous basis for the compositional modelling and simulation of discrete event systems. This section briefly presents the Parallel DEVS formalism, a variant of the original Classic DEVS.

DEVS allows modelling at two distinct levels of abstraction: *atomic* and *coupled* components. An atomic component model describes the discrete-event autonomous and reactive timed behaviour of system.

An atomic model is formally defined as,

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

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

A coupled component model describes the composition of several concurrent submodels which can be either atomic or coupled. Submodels have *ports*, which are connected by channels defining a *transfer function* to translate output to input events. Ports and channels allow a component model to receive/send events from/to other models. A coupled DEVS model is closed under coupling, which means its meaning can be given by "flattening" to a behaviourally equivalent atomic DEVS model.

An abstract simulator for Parallel DEVS computes the next state of the system (a "step") until its end condition is satisfied. Each step consists of the following phases:

1. Compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components).
2. Execute the output function for each imminent component, causing events to be generated on the output ports.
3. Route events from output ports to input ports, translating them in the process by executing the transfer functions.
4. Determine the type of transition to execute for the atomic DEVS model, depending on it being imminent and/or receiving input.
5. Execute, in parallel, all enabled internal, external, and confluent transition functions.
6. Compute, for each atomic DEVS model, the time of its next internal transition (specified by its time advance function).

3. DEVS IM&S ENVIRONMENT

Our integrated DEVS modelling environment uses the Modelverse [14] as its model repository and model management facility. The Modelverse is an infrastructure for scalable and efficient model-management. It includes a built-in meta-modelled action language to give basic execution support to various model operations, such as conformance checks performed on models w.r.t. metamodels. Based on such analysis, it produces model instances. It also supports rule-based model transformations. The notion of time is not present in the currently supported action code. Therefore, the integration of time-aware formalisms such as DEVS in the Modelverse is desirable.

In our work, we make use of a visual modelling, simulation, and debugging environment for Parallel DEVS that is already available in AToMPM [12] (A Tool for Multi-Paradigm Modelling), a generic visual front-end for the Modelverse.

Our initial task is to allow models, visually edited in AToMPM and stored in the Modelverse, to be retrieved, edited textually in a textual shell (and possibly imported back into the AToMPM visual view). For this purpose, we propose a language, DEVSLang, with an expressiveness equal to that of DEVS models that can be visually represented. Note that not all DEVS models can be explicitly modelled visually as we can only draw a finite number of sequential states or external transition functions.

The second task is to be able to reuse the parsing and analysis features of the textual shell for the checking of the transition-, time advance- and output functions of a DEVS model.

The third task is to translate all DEVS models automatically for use in existing simulators. As an example, we compile DEVS models into PyPDEVS [16], which is a DEVS-based language grafted onto 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, etc.). As PyPDEVS combines the full expressiveness of the Parallel DEVS formalism with that of the programming language Python, a language is needed at this expressiveness level too. For this purpose, we propose the language DEVSPro, from which PythonPDEVS code will be synthesized. We must ensure that DEVSPro *(i)* has an action code part that is as expressive as Python but yet *(ii)* enables a strictly verifiable conformance with DEVS.

Our complete workflow is described in Figure 2 in the form of a FTG+PM model (Formalism Transformation Graph and its complement, the Process Model) [7]. Figure 2 charts various activities in the MDE lifecycle, from requirements development to code syn-

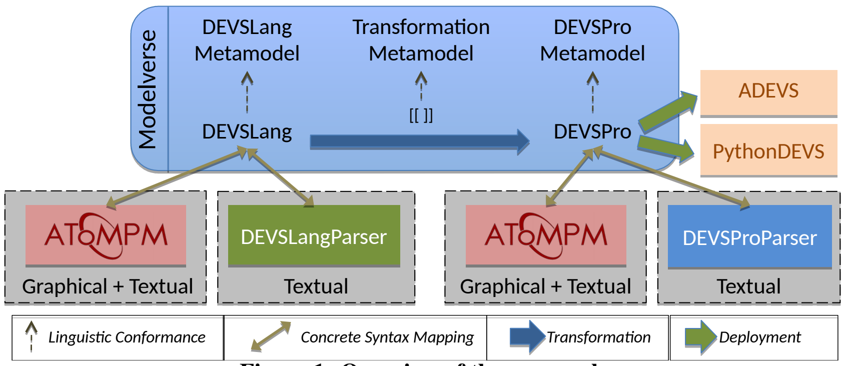


Figure 1: Overview of the approach.

thesis, that are part of our DEVS IM&SE. The FTG describes in an explicit and precise way, formalisms, and their relationships as transformations between formalisms. The PM defines an actual Model-Based Systems Engineering process using these formalisms and transformations. As depicted in the FTG+PM, DEVSPro models are transformed to PyPDEVS for simulation purposes. All DEVSLang models need to be transformed to DEVSPro in order to allow for the synthesis of the PyPDEVS code. This transformation to DEVSPro also allows models to be ported to different target platforms, and makes it possible for advanced users to adapt existing DEVS models in an unrestricted manner: for instance, without imposing the definition of a state enumeration on each Atomic component.

3.1 DEVSPro

DEVSPro is based on DEVS theory and is at an abstraction level closer to programming languages to allow expert users to exploit the full power of DEVS while maintaining a close conformance with the DEVS theory. The formalism is conceptually identical to PyPDEVS with a specific abstract syntax and semantics.

With regards to the tool integration aspect, the semantics of DEVSPro is given by mapping it to PyPDEVS. The formalism is composed of two kinds of syntactic structures: syntactic structures expressing action code in DEVS functions and syntactic structures expressing the structure of DEVS models (i.e., the atomic and coupled components).

DEVSPro is rooted in the Modelverse by means of an action language. The textual shell allows modellers to read a file module and load it. The ‘import’ feature allow modellers to refer to and load new class definitions. In the example below, ‘statedef’ is a class being imported as a means to define the ‘ProcessorState’, that is then instantiated in the atomic component ‘Processor’ as ‘idle’.

```
top:
  import statedef
...
statedef ProcessorState:
    constructor(name, job=None):
        self.name = name
        self.job = job
...
atomic Processor:
    constructor():
        self.state = ProcessorState('idle')
        self.outports = {'p_out'}
        self.inports = {'p_in'}
...
```

Listing 1: Snippet of the Producer-Consumer Example in DEVSPro

In the context of the example above, the notion of constructor (with a Python-style syntax) enables a given atomic component (in this case ‘Processor’) to bind, at instantiation phase, with a given state definition (in this case ‘ProcessorState’). The reserved word ‘state’ is used in order to refer to the concept state S, but its type is, by default, left underspecified. In the example, it is bound to an instance of ProcessorState of type statedef. The reserved words ‘inports’ and ‘outports’ are used to define the names of the input and output ports, respectively. Each port allows interface connections and logical dependencies with other components. The chosen names for the ports typically reflect the communication usage from the point of view of the component (i.e., from an OO perspective). In our example, if it was a producer of proteins, the processor’s input port would be simply named ‘protein’ which, when assigned becomes a reserved word in the atomic component known as a type of input port.

However, in the case of complex atomic component initializations, the map of typed (input/output) port names could be changed during simulation. Given that these port names can be changed in the arrival/departure of an event, we could have a conflict between out-dated instances with the subsequent modification of the ports. For now, we avoid discussions about the correct scheduling in order to avoid or mitigate these conflicts that could only happen in more complex conditions.

Notice that although we present a deployment to PyPDEVS as an example of DEVSPro semantics, we only show it in that particular platform, not with its full semantics, which is to be by definition platform-neutral. Following Zeigler’s formalism, the full semantics remains largely underspecified in what matters to concrete semantics and representations, although complete in what matters to internal consistency rules. This is enforced by construction on any DEVSPro type system, conforming with the DEVSPro’s type model.

The textual shell allows the user to define functions in many different ways. Functions are considered to be type definitions. In this case, a function call is just an instance of a Function type. However, DEVSPro Atomic components (in the example above atomic Processor) are forced to redefine a set of Function types, each of them conforming to a particular DEVSPro Function Meta-Type. From the time advance function ta to the output function λ, each of them are semantically typed according to Zeigler’s formalism. The correct analysis of each of these function types is, of course, limited by our analysis algorithms, but if we impose semantic type-restrictions in the way the modeller composes DEVSPro models, it may be possible to assure and verify the preservation of these semantic types—i.e., the consistency rules inherent from each of the functions’ requirements, such as, for instance, the return type of the internal function δ_{int} *versus* the return type of the time advance function ta.

Notice in the example, that ‘outputFunc’ (i.e., λ) outputs a value from the current state variable (namely ‘job’) into output port p_out. This is possible, because in an output function, the state variable S can be read (although not modified). In this DEVSPro model, the output port is implicitly typed with the type of variable ‘job’ (which

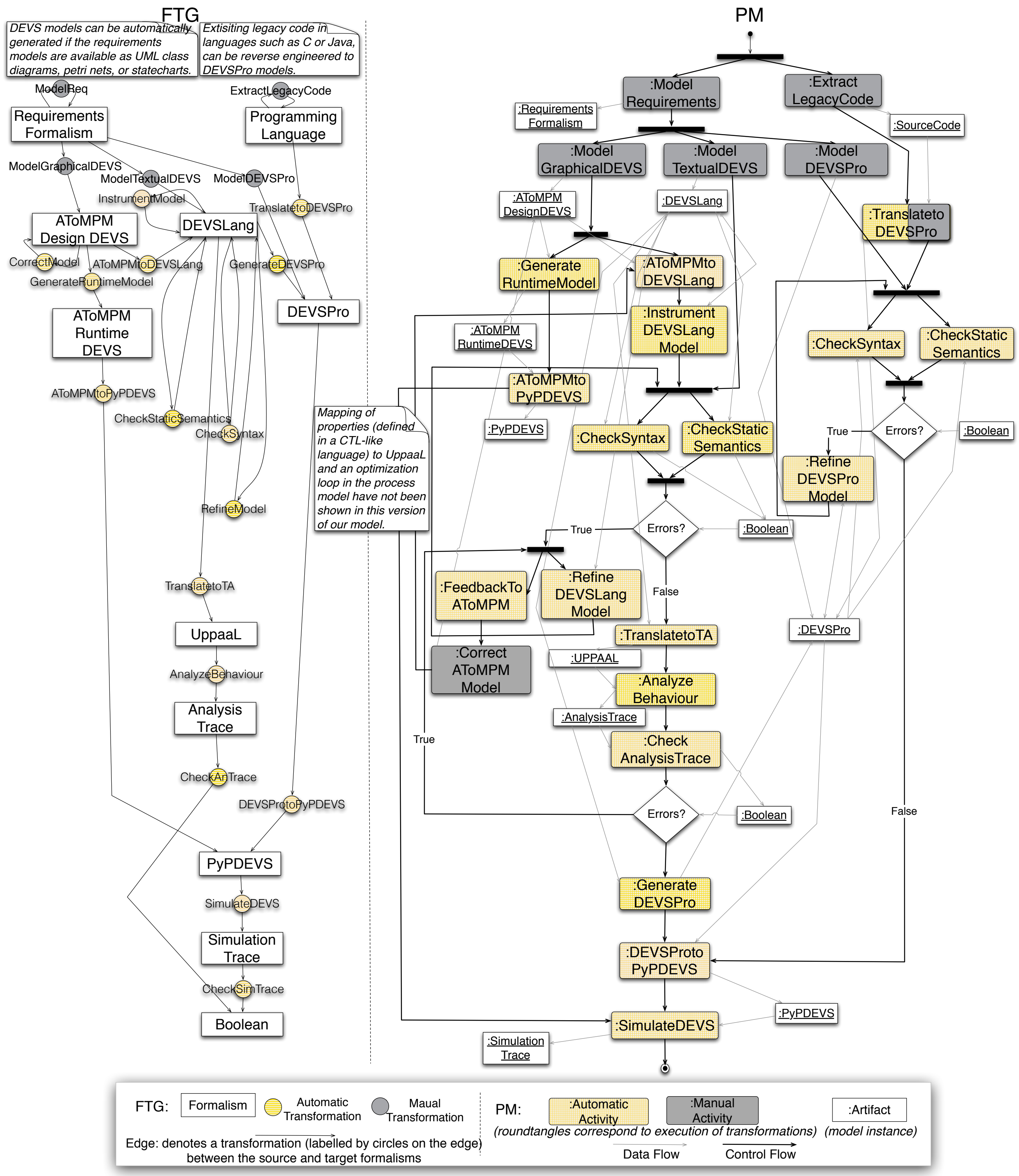

FTG
PM
DEVS models can be automatically generated if the requirements models are available as UML class diagrams, petri nets, or statecharts.
Extisiting legacy code in languages such as C or Java, can be reverse engineered to DEVSPro models.
Mapping of properties (defined in a CTL-like language) to UppaaL and an optimization loop in the process model have not been shown in this version of our model.
Requirements Formalism
Programming Language
AToMPM Design DEVS
DEVSLang
DEVSPro
AToMPM Runtime DEVS
UppaaL
Analysis Trace
PyPDEVS
Simulation Trace
Boolean
:Model Requirements
:Extract LegacyCode
:Model GraphicalDEVS
:Model TextualDEVS
:Model DEVSPro
:Translateto DEVSPro
:Generate RuntimeModel
:AToMPMto DEVSLang
:Instrument DEVSLang Model
:AToMPMto PyPDEVS
:CheckSyntax
:CheckStatic Semantics
Errors?
True
False
:Refine DEVSPro Model
:FeedbackTo AToMPM
:Refine DEVSLang Model
:Correct AToMPM Model
:TranslatetoTA
:Analyze Behaviour
:Check AnalysisTrace
:Generate DEVSPro
:DEVSProto PyPDEVS
:SimulateDEVS
FTG: Formalism Automatic Transformation Maual Transformation
Edge: denotes a transformation (labelled by circles on the edge) between the source and target formalisms
PM: :Automatic Activity :Manual Activity :Artifact
(roundtangles correspond to execution of transformations) (model instance)
Data Flow Control Flow

Figure 2: A Formalism Transformation Graph and Process Model (FTG+PM) for DEVS: the PM is depicted as an activity diagram (on the right) and the FTG is depicted as a hypergraph (on the left) with *languages* as nodes and *transformations* as edges).

in the example can be just an Integer).

This means that this component, when connected, since the atomic component Processor will have port p_out with event values of type Integer, the other DEVS component must be reading events from the corresponding input port of the same type Integer. However, there can be several outcomes programmed in a given output function λ, all of them must be a map of port names with their respective typed variables or values. The composition of all of the possible types for a given port type (e.g., p_out) is given by an ordered union of constraints that have priority over others: be-

```
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
...
```

Listing 2: Snippet of the Producer-Consumer Example in DEVSPro

cause action code is a sequence of statements. This means that in a sequence of statements, each 'if' or 'while' imposes a constraint that has more priority than the constraint on the next statement. The constraints are added to the return types on each branch of the dataflow diagram imposed on any give action code execution. It is possible, however, to override this implicit constraint-priorization by using logical connectors such as 'and', in order to assign the same priority, However in the case of 'or', we must assure that both operands are logically disjoint, otherwise we would have more type-check ambiguity (and in particular non-determinism) in the port assignments while evaluating coupled components.

```
...
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'])
...
```

Listing 3: Snippet of the Producer-Consumer Example in DEVSPro

In the listing above, we show a coupled DEVS model named 'ProducerConsumer' that instantiates its internal DEVS components during its construction, and places them in the reserved variable 'submodels'. Notice that all of the instances are typed by construction using the user-defined DEVS component types 'Generator', 'Processor', etc. Instantiating the connectPorts function enables the binding of input with output ports from all of the instantiated internal DEVS components: i.e., the channels. Not shown but, it is also possible to pass a user-defined transfer function as a parameter of the connectPorts built-in. A suitable transfer function will have exactly one parameter—i.e., a value from some source port in a connection—, and will return a value, which at simulation-time is passed to the target port on that connection. In either case, the type checker has to ensure that the type of the events on the input port, of a given component, is consistent with the type of the events on the output port, of the other component.

Finally, we present the 'bottom' function. Notice that parameterless functions (as mere action code labels) are always executed and cannot be used as types. At the end of the DEVSPro module file, we instantiate one of the coupled models (namely the 'ProducerConsumer') and set some attributes by instantiating pre-defined simulation Function types (verbose and termination time). These must be defined for a given platform (in this case PyPDEVS). Notice that, since the textual shell supports dynamic type operations on instances, it is possible at any moment in time to extend a given ProducerConsumer instance with a new platform-dependent functionality. Finally, we run the simulate function in order to produce simulation traces (not shown here).

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

Listing 4: Snippet of the Producer-Consumer Example in DEVSPro

3.2 DEVSLang

DEVSLang is a language whose expressiveness is identical to that of the DEVS formalism encoded for the visual modelling in AToMPM—i.e., the AToMPM Design DEVS in Figure 2. This means that the abstract syntax of textual DEVSLang is the same as that of the visual DEVSLang language in AToMPM. In our environment, the formalism has been associated with two different concrete syntaxes: *(i)* a 100% textual concrete syntax and *(ii)* a blended visual and textual concrete syntax for modelling in AToMPM. The textual concrete syntax here refers to DEVSLang's neutral action language.

The DEVSLang textual syntax primarily allows 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 are by default included as syntactic constructs in the textual model.

We now detail the concrete syntax for DEVSLang by partially showing (due to space constraints) the corresponding DEVSLang model (Listing 5) for the Producer-Consumer example presented in Section 3.1. An exporter in AToMPM is able to generate the DEVSLang textual models from the visual DEVS models (Figure 8a).

In comparison with the DEVSPro model of the ProducerConsumer shown before, this example reveals that its representation in DEVSLang becomes much more compact. This is mostly because it builds on an enumeration of states that is implicitly defined in each atomic component. For instance, in the 'Processor' component, the used 'ProcessorState' will be a tuple containing an enumeration, with only two possible values (*idle* and *processing*), and a variable *job*. Notice that in DEVSLang, each state definition 'statedef' only defines regular state variables: for instance, 'ProcessorState' defines *job* as a state variable.

In the case of DEVSLang, the types of parameters are restricted to basic datatypes (String, Boolean, Integer, Float, etc.). DEVSLang includes pre-defined constructs for state definition (as in line 3), events (as in line 6), atomic components (as from lines 8 to line 24), and coupled components (as from line 27 to line 36).

As we can see inside each atomic component definition, DEVSLang mimics the notion of enumerated states as used in AToMPM's visual syntax, with the notion of *states*. *States* are connected by

```
1      # State definitions
2      ...
3      statedef ProcessorState(job)
4      ...
5      # Event definitions
6      event Job(jobSize)
7      ...
8      atomic Processor() with ProcessorState:
9        inports p_in
10       outports p_out
11
12       initial idle
13       state idle:
14         time advance:
15           return infinity
16         external -> processing:
17           action:
18             return {job: p_in[0]}
19       state processing:
20         time advance:
21           return job.jobSize
22         internal -> idle
23         output:
24           return {p_out: [job]}
25     ...
26     ## component Root needs to be defined
27     coupled ProducerConsumer():
28       instances:
29         g = Generator(a,b)
30         cp = CoupledProcessor()
31         p = Processor()
32         c = Collector()
33       connections:
34         from g.p_out to cp.p_in
35         from cp.p_out to p.p_in
36         from p.p_out to c.p_in
37     ...
```

Listing 5: Snippet of the Producer-Consumer Example in DEVSLang

means of either 'external' transitions or 'internal' transitions. Each transition function points to a given state name (which will be the name of the state as defined in type 'statedef'), and returns a map of parameters which must always conform to a user-defined 'statedef' type instance. The conforming implication of a given atomic component with a given 'statedef' type instance is set explicitly with the keyword 'with' (see line 8).

3.3 Mapping and Merging

In order to illustrate the commonalities between both languages, and also to compare their expressiveness, Table 1 provides a mapping from DEVSLang to DEVSPro constructs. The *OO* translation function is applied to all action *statements* in order to make the expressions consistent with object-orientation navigation features. Similarly, the *Cond* translation function is applied to an action statement block converting it into a DEVSPro condition function type with *return type Boolean*, which is then directly instantiated in a given *if statement's condition*.

Although not mentioned in Table 1, the rule that is applied to the time advance function is also applied to the output function. The same is true for external transitions and both internal transitions and confluent transitions. Also not shown is the trivial mapping between the coupled component's graph: the *connections* between DEVSLang's *instances* (these ones mapped into the respective DEVSPro component's *submodels*) are mapped onto calls to the DEVSPro reserved method *connectPorts*.

Figure 3 presents the core of the DEVSLang metamodel. If we compare this metamodel with the DEVSPro metamodel, we see that in DEVSPro there is no fixed state definition structure, and therefore DEVS functions are directly defined under the atomic component definition (instead of being under a given state declaration). Also, we see that unlike DEVSLang, DEVSPro allows user-defined functions, and in particular parameterless functions. Finally, it is not shown, but following DEVS theory, the *elapsed* keyword is defined in both formalisms as an attribute that can be accessed inside a given atomic component.

The action code metamodel used in DEVSLang is shown in Figure 4. The action code metamodel of DEVSPro is a superset of

DEVSLang Concepts	DEVSPro Concepts
statedef SDName(pars..) ..	statedef SDName: constructor(pars..): self.par1 = xpto ..
atomic CName() with SDName: .. initial ISName ..	atomic CName: constructor(): self.state = SDName(ISName) ..
atomic CName(args..)	atomic CName: constructor(args..): ..
atomic CName() ..: .. inports p, q,.. outports r, s,.. ..	atomic CName: constructor(): self.inports = {'p', 'q', ..} self.outports = {'r', 's', ..} ..
atomic CName()..: .. state SName: timeAdvance: Statements.. ..	atomic CName: .. timeAdvance(): if self.state == 'SName': OO(Statements..) ..
atomic CName() with SDN: .. state SN: external $-$ $>$ SON: condition: Stms.. action: Stms'.. ..	atomic CName: .. extTransition(inputs): if self.state == 'SN' and Cond(Stms..): OO(Stms').. return SDN('SON', args) .. ..

Table 1: Reference mapping between DEVSLang Concepts and DEVSPro Concepts.

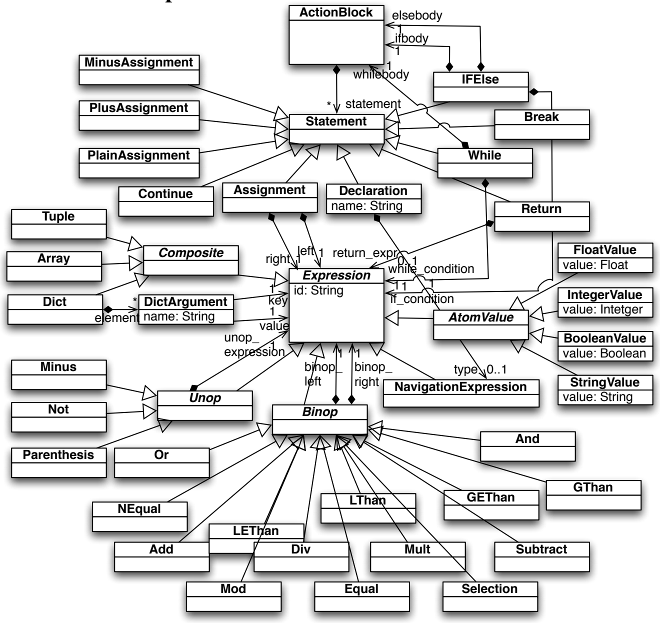


Figure 4: Neutral Action Code Metamodel used in both DEVSLang and DEVSPro. ***The DEVSPro's Method Definition construct is not shown since it only represents an instantiation possibility in DEVSPro models.***

the DEVSLang one including object-orientated constructs. Bear in mind that these figures only show the linguistic part of the DEVSLang and partially DEVSPro metamodels. The typing rules that constrain all sentences to a conformity to DEVS theory (presented in Section 2) are also represented in the Modelverse and accessible using the textual shell.

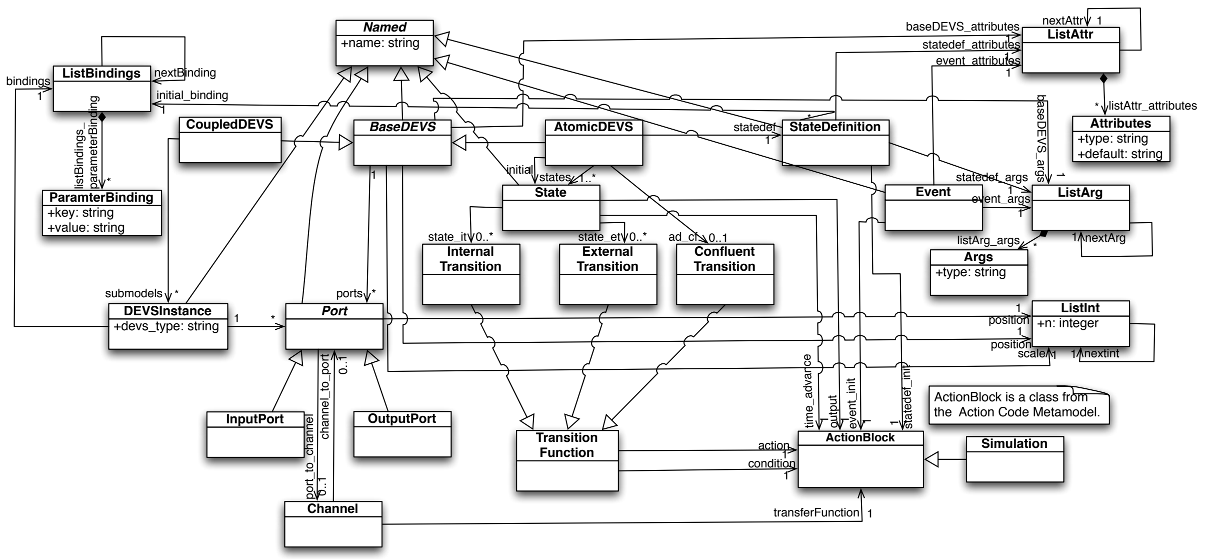


Figure 3: The DEVSLang Metamodel.

4. ANALYSIS

The grammar specifications for the languages DEVSLang and DEVSPro drives the automated generation of their respective parsers that as shown before are able to recognize their sentences and assert their syntactic correctness. In particular, they issue syntactic errors when some grammar rule is violated.

However, the 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. For instance, it is possible to write an output function that modifies existing state variables, which is incorrect under DEVS theory.

We will now observe how we can exploit the usage of textual notations to model DEVS and, most importantly, the usage of action code, in order to provide powerful mechanisms for consistency analysis.

4.1 Static Semantics

Since the information contained in context-free grammars is not sufficient to enable sophisticated evaluations of the parsed DEVSLang model, we need to provide additional information in order to capture all the semantic rules required for both our DEVS languages. In particular, the following rules have to be specified, and dealt separately in an evaluation procedure distinct from parsing.

- Each reference marked on the ‘with’ on each atomic component definition, must point to a valid declared state definition.
- Each atomic block must have exactly one initial definition.
- Each external, internal or confluent function, must have at most one condition block and one action block.
- The target pointed to by either an external, internal or confluent functions must be a valid defined state defined on the same atomic component.
- Function calls referenced in any expression in the action code must point to a valid Event declaration, and therefore must comply with its required parameters.
- On every two connected instances, the mapped written and read values on the respective connected $inports$ and $outports$ names must have compatible types.
- On the expressions, any specified operation must be consistent w.r.t. declared types of their operands (which can again be operations).

4.2 Checking Conformance with DEVS

The contextual analysis can go deep by analyzing conditional branches, in order to check the return and parameter types of the time-advance, internal/external/confluent transition and output functions. It can, for instance, check the only allowed read/write variable access within the above functions (e.g., the elapsed variable in the condition of an external function).

The action code defined in the condition part of each transition definition can be analyzed in order to check that the component’s variables are just being read: meaning that the evaluation of a condition cannot change the variables of a given component, or of any other defined component.

The complete set of access rules for conformance checking is presented in Table 2. It details for each function, the expected types for the condition and action blocks. Warnings are issued if the analysis is inconclusive, and errors are issued if an inconsistency is detected.

For the sake of brevity we omit the rules for the transfer functions on the coupled DEVS.

The DEVSPro syntax for DEVS functions does not offer any explicit distinction between the condition and action parts. Therefore, the resulting conformance checking will have to be necessarily weaker. However, we can reuse the analysis made for DEVSLang (as detailed in Table 2) for DEVSPro, with the exceptions that here we apply both the rules for conditions and actions. In the $outputFunc$, the checker allows read access for all the variables except the $self.inport$ and $self.elapsed$ variables.

5. ACTION CODE ANALYSIS

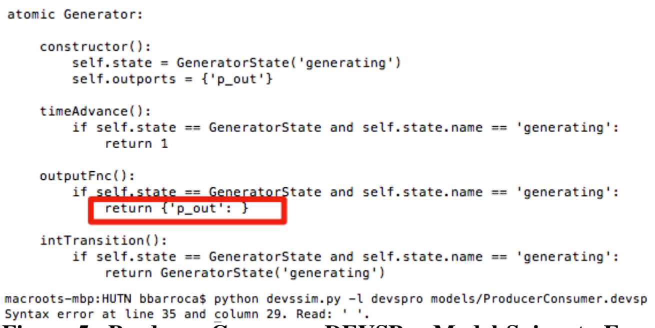
As outlined in the previous section, we have implemented a parser and analyzer to carry out syntactic and structural semantic checks of our DEVS models. In this section, we demonstrate these automated analyses implemented in our DEVS modelling environment by means of the Producer-Consumer example.

Action Code in DEVSPro. New or existing models can be analyzed by the parser/checker to identify syntactic and contextual errors. The Producer-Consumer example DEVSPro model presented in Section 3.1 (including errors this time) was analyzed for syntactic errors. Fig. 5 shows the results of the analysis (the first error detected).

Action Code in DEVSLang.The DEVSLang models including

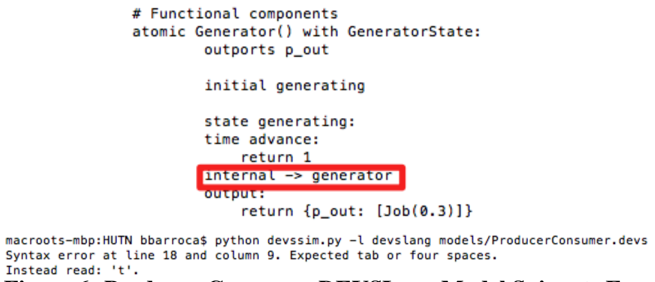
	Time Advance Function	Internal Function		External Function		Confluent Function		Output Function
		Condition	Action Code	Condition	Action Code	Condition	Action Code	
Return Type	Float	Boolean	Map	Boolean	Map	Boolean	Map	Map(*)
State variables	Read Only	Read Only	Read/Write	Read Only	Read/Write	Read Only	Read/Write	Read Only
Inport variables	N/A	N/A	N/A	Read Only	N/A	Read Only	N/A	N/A
Outport variables	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Write Only
Elapsed variable	N/A	N/A	N/A	Read Only	N/A	Read Only	N/A	N/A

Table 2: Contextual Analysis Rules for DEVSLang models. (*) where all labels belong to defined Outports in given Atomic Component



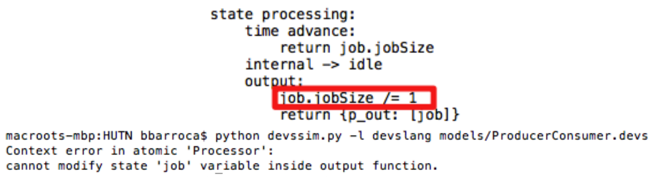
```
atomic Generator:

    constructor():
        self.state = GeneratorState('generating')
        self.outports = {'p_out'}

    timeAdvance():
        if self.state == GeneratorState and self.state.name == 'generating':
            return 1

    outputFnc():
        if self.state == GeneratorState and self.state.name == 'generating':
            return {'p_out': }

    intTransition():
        if self.state == GeneratorState and self.state.name == 'generating':
            return GeneratorState('generating')

macroots-mbp:HUTN bbarroca$ python devssim.py -l devspro models/ProducerConsumer.devsp
Syntax error at line 35 and column 29. Read: ' '.
```

Figure 5: Producer-Consumer DEVSPro Model Snippet: Example Error (incomplete return statement)



```
        # Functional components
        atomic Generator() with GeneratorState:
                outports p_out

                initial generating

                state generating:
                time advance:
                    return 1
                internal -> generator
                output:
                    return {p_out: [Job(0.3)]}

macroots-mbp:HUTN bbarroca$ python devssim.py -l devslang models/ProducerConsumer.devs
Syntax error at line 18 and column 9. Expected tab or four spaces.
Instead read: 't'.
```

Figure 6: Producer-Consumer DEVSLang Model Snippet: Example Error (incorrect state name specified)



```
                state processing:
                    time advance:
                        return job.jobSize
                    internal -> idle
                    output:
                        job.jobSize /= 1
                        return {p_out: [job]}
macroots-mbp:HUTN bbarroca$ python devssim.py -l devslang models/ProducerConsumer.devs
Context error in atomic 'Processor':
cannot modify state 'job' variable inside output function.
```

Figure 7: Contextual Check Example Error

the action code in the defined functions can be checked for syntactic errors, conformance errors, and for contextual errors. The action code in the Producer-Consumer DEVSLang model in Listing 5 was analyzed for syntactic and semantics errors. Fig. 6 shows the initial error returned.

In Figure 7, we show the result of a contextual check. In this case, a statement in an output function is trying to change the value of a state variable. A DEVS output function should however not change a component's state.

While this section only demonstrates validation of action code, our tool checks the entire model for syntactic errors.

Action Code in AToMPM. The DEVSLang models constructed in our visual front-end, AToMPM, includes transition functions specified as DEVSLang action code. As in many current tools, these functions were originally defined using a programming language (Python in our case), which was not validated for syntax or semantics. The exported Python code needed to be parsed and any changes made in the Python code were not updated in the original model (or it was updated manually which could lead to even more errors). Furthermore, the Python models were platform-dependent which restricted portability to different simulators. Using neutral action code alows the models to be validated in the front-end and also keeps the model consistent with the generated DEVSLang and DEVSPro models.

We have implemented a feedback loop such that the errors detected in the generated textual DEVSLang model can be fed back to the AToMPM visual modelling environment. This is based on traceability information. The user only needs to press one button (outlined in red in Fig. 8a) to validate the action code in the model. The model is then exported to textual DEVSLang and parsed, and the results are returned to AToMPM (shown in Fig. 8b). The erroneous functions are highlighted on screen with a message next to them elaborating on the kind and location of the error. Fig. 9 shows screen shots of two such syntactic errors detected and highlighted in the visual model.

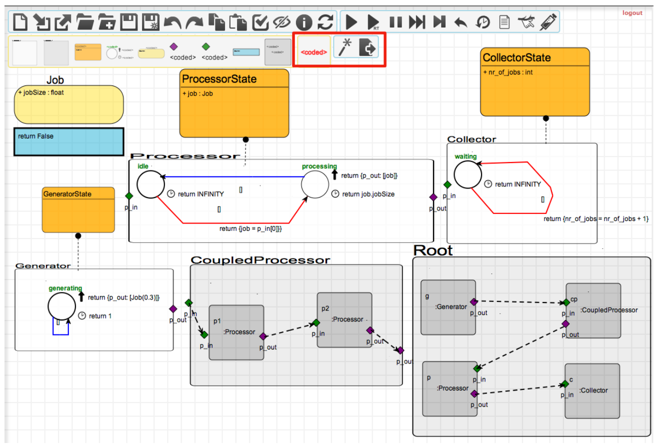
Users also have the option of exporting to textual DEVSLang with the press of a button (shown within red-outlined box in Fig. 8a) once the model is corrected.

6. RELATED WORK

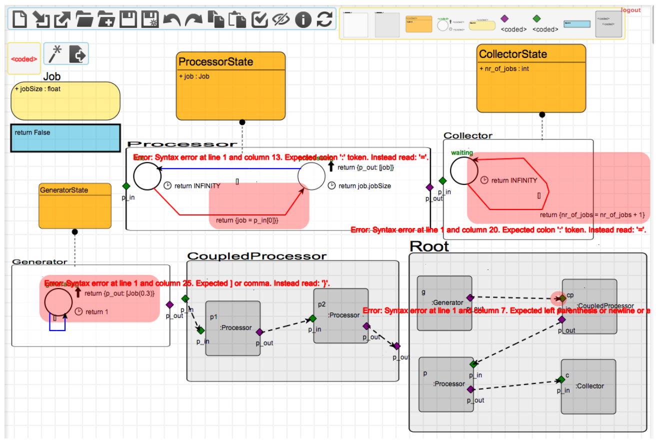
Over the years, several extensions and variants of the DEVS formalism have been proposed [5]. One of our primary concerns in this paper was to address the specification of the DEVS functions. In the DEVS language extensions described in the literature, functions are typically expressed using a textual syntax strongly influenced by (or identical to) the existing underlying programming language onto which the simulation environments are grafted (e.g., as seen in [2]). The need to provide a platform independent specification, with the capability to specify platform-neutral code statements (i.e., action code) that can then be translated to any given programming language has remained unaddressed in most related work.

Yet another concern was the *interoperability* of DEVS environments along with the need for making available an integrated environment for DEVS. A commonly adopted solution for this problem (e.g. in [8, 13, 3]), 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, although little support is then provided for action code editing.

In Table 3, 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 [18]. While various graphical front-ends for modelling DEVS are available (such as [11]), 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 anal-*



(a) Producer-Consumer Example DEVSLang Model


Error: Syntax error at line 1 and column 13. Expected colon ':' token. Instead read: '='.

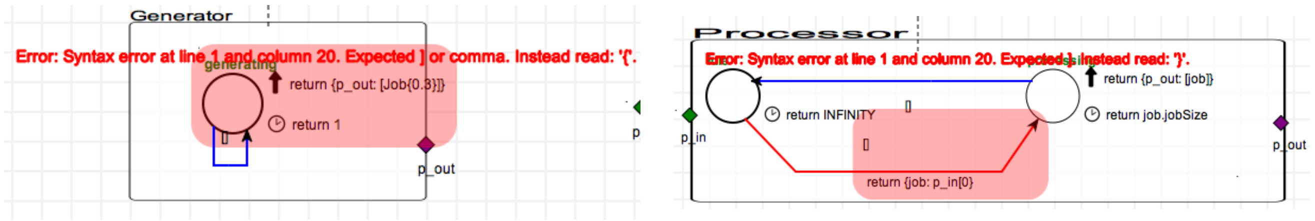
Error: Syntax error at line 1 and column 20. Expected colon ':' token. Instead read: '='.

Error: Syntax error at line 1 and column 25. Expected] or comma. Instead read: '}'.

Error: Syntax error at line 1 and column 7. Expected left parenthesis or newline or e

(b) Producer-Consumer Example: Syntax checking of Action Code

Figure 8: AToMPM DEVS Modelling Environment


Error: Syntax error at line 1 and column 20. Expected] or comma. Instead read: '{'.

Error: Syntax error at line 1 and column 20. Expected] Instead read: '}'.

Figure 9: DEVSLang Syntax Check Example Errors

ysis in Table 3 refers to syntax checking and type checking including static semantics analysis, and *symbolic analysis* refers to model checking or behavioural analysis.

Based on the related work we have examined, we conclude that the main challenge is to tackle the need for an integrated modelling environment, which integrates tools for modelling, analysis, simulation, optimization and execution. In our work, we have attempted to integrate 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 such environments with multiple languages at different levels of abstraction, 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.

7. CONCLUSION

We first introduced two neutral textual languages, DEVSPro and DEVSLang, made for different purposes. Unlike other existing notations for DEVS, these also use a neutral action code to represent transition functions, time advances, and output functions in our visual DEVS modelling environment.

We then described a reference semantic metamodel (i.e., a set of mapping and type rules) in order to enable the semantic conformance of DEVS models (expressed on those two DEVS notations) with Zeigler's DEVS theory. This semantic metamodel was embedded in the internals of a DEVS Integrated Modelling & Simulation Environment (IM&SE), by means of: consistent automated mappings (model transformations, translators, and exporters); and consistent automated (linguistic) syntax and (semantic) conformance/contextual checkers.

The advantages of the such kind of integration are twofold. On the one hand, modellers now have the option to easily switch between a completely textual (in the textual shell) or a textual/visual (in AToMPM) model representation. On the other hand, the semantic conformance to DEVS can now be ensured on all of these representations, by means of automated syntax and conformance checking. This is crucial in order to be able to automatically guarantee (or check) that the deployed models (for instance into PythonPDEVS) are indeed valid DEVS models.

As future work, we intend to continue the integration of different kinds of engines in our IM&SE in order to explore existing interesting type systems (founded on DEVS), that may challenge the analyzability of its internal consistency.

8. ACKNOWLEDGEMENTS

This work was partly funded by the Automotive Partnership Canada (APC) in the NECSIS project, and by PhD fellowships from the Agency for Innovation by Science and Technology in Flanders (IWT) and the Research Foundation - Flanders (FWO). The authors also acknowledge Ernesto Posse for his early work on DEVSLang.

9. REFERENCES

[1] BARROCA, B., KÜHNE, T., AND VANGHELUWE, H. Integrating language and ontology engineering. In *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)* (2014).

	DEVSpecL [4]	DEVSML [6]	DEVSML 2.0 [8]	DEVSRuby [2]	DEVSW [20]	SimStudio [13]	Meta-DEVS [3]	**Our DEVS Framework**
DEVS (C^1/P^2)	C	C	C,P	C	C	P	C	P
Meta-modelling/MT	X	XSL Transformation	✓	X	X	✓	✓	✓
DEVS Variant	Theory	Theory	Finite Deterministic DEVS	Theory	FSA^3	Theory	FSA and Theory	FSA and Theory
DEVS Levels	✓	✓	✓	DSL only	X	✓	X	✓
Analysis								
Static	✓	X	✓	X	X	X	X	✓
Symbolic	X	X	X	X	X	X	X	X
Modelling								
Graphical	✓	Limited support	X	X	X	✓	✓	✓ (AToMPM)
Textual	✓	✓ (XML)	✓ (NLDEVS, DEVSML)	✓	✓ (XML)	✓ (XML)	✓ (XML)	✓ (DEVSLang, DEVSPro)
Integrated Environment	✓	X	✓	X	X	✓	✓	✓
Action code	✓	Lisp-like	JAVAML	X	X	✓ (neutral)	✓ (low expressiveness / rule-based)	✓ (neutral)
Simulation								
Debugging	X	X	X	X	X	X	X	✓
Execution	✓	X	✓	✓	✓	✓	✓	✓
Target Platform	C++, Java	Various (PyDEVS, DEVSJava)	DEVSJava	Ruby, Ruby C	DEVS-Scheme	Multiple	Java and Python	Python
Portability	✓	✓	✓	X	✓	✓	✓	✓

Table 3: DEVS Environments: A Comparison (^{1}C: Classic DEVS, ^{2}P: Parallel DEVS; ^{3}FSA: Finite State Automaton)

[2] 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* (2014), SCS, pp. 393–398.

[3] Garredu, S., Vittori, E., Santucci, J. F., and Bisgambiglia, P. From state-transition models to DEVS models - improving DEVS external interoperability using metadevs - A MDE approach. In *SIMULTECH 2013 - Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Reykjavík, Iceland, 29-31 July, 2013* (2013), pp. 186–196.

[4] 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.

[5] Hwang, M. H. Taxonomy of DEVS variants. In *TMS-DEVS, SpringSim* (2014), pp. 445–450.

[6] Janoušek, V., Polášek, P., and Slavíček, P. Towards DEVS Meta Language. In *ISC* (2006), pp. 69–73.

[7] Lucio, L., Mustafiz, S., Denil, J., Vangheluwe, H., and Jukss, M. FTG+PM: An integrated framework for investigating model transformation chains. In *SDL 2013: Model-Driven Dependability Engineering*, vol. 7916 of *LNCS*. Springer, 2013, pp. 182–202.

[8] Mittal, S., and Douglass, S. A. DEVSML 2.0: the language and the stack. In *TMS-DEVS, SpringSim* (2012), pp. 17:1 – 17:12.

[9] Nutaro, J. adevs. `http://sourceforge.net/projects/adevs/`.

[10] Sarjoughian, H. S., and Chen, Y. Standardizing DEVS models: An endogenous standpoint. In *TMS-DEVS, SpringSim* (2011), SCS, pp. 266–273.

[11] Song, H. Infrastructure for DEVS modelling and experimentation. Master's thesis, School of Computer Science, McGill University, 2006.

[12] 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).

[13] Touraille, L., Traoré, M., and Hill, D. A model-driven software environment for modeling, simulation and analysis of complex systems. In *TMS-DEVS, SpringSim* (2011), SCS, pp. 229–237.

[14] 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.

[15] Van Mierlo, S., Van Tendeloo, Y., Mustafiz, S., Barroca, B., and Vangheluwe, H. Explicit modelling of a Parallel DEVS experimentation environment. In *TMS-DEVS, SpringSim* (2015). Accepted for publication.

[16] Van Tendeloo, Y., and Vangheluwe, H. The modular architecture of the Python(P)DEVS simulation kernel. In *TMS-DEVS, SpringSim* (2014), pp. 387–392.

[17] Vangheluwe, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design* (September 2000), pp. 129–134.

[18] 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, pp. 427–458.

[19] 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, pp. 393–425.

[20] Wang, Y., and Wang, L. An XML-based DEVS modeling tool to enhance simulation interoperability. In *Proceeding 14th European Simulation Symposium* (2002).

[21] Zeigler, B. P. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[22] 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.