A TLA+ Encoding of DEVS Models

Maximiliano Cristiá*
U. N. de Rosario - U. N. de Córdoba
Flowgate Security Consulting
Email: mcristia@fceia.unr.edu.ar

***Abstract*— Control Theory researchers have been using DEVS models to formalize discrete event systems for a long time [1] but, despite such systems are one of the main targets of Software Engineers, the DEVS formalism has not been used and it is hardly known by the formal methods community of Computer Science. This paper is an attempt to close the gap between these communities by setting some of the basic rules to translate DEVS models into TLA+ specifications. TLA+ [2] is a widely known formalism, used by formal methods researchers and practitioners, to specify –hardware or software based– reactive or concurrent systems. The paper includes some theoretical insights, some modeling design decisions and an example of the encoding we propose.**

***Index Terms*— DEVS, TLA+**

I. Introduction

Modeling critical, mission critical or embedded reactive systems is nowadays an accepted practice both in academia and industry [3], [4], [5], [6], [7]. It is perhaps the application domain that most rapidly has accepted that using formal techniques in earlier phases of the development process worth the (apparently) added costs. DEVS is a formal modeling technique and notation originally developed by Bernard P. Zeigler in the early '70 [1]. It has been routinely and successfully used, researched and expanded by researchers and practitioners belonging to the Control Theory or Automation communities. However, DEVS and its success has not been recognized as such by the Formal Method community of Computer Science; remarkably, one of the most complete and referenced on-line resources of this community [8] does not even list DEVS as a formal notation or method.

As an enthusiastic of the application of (Computer Science's) Formal Methods to software development, the author of this paper finds this fact quite interesting and deserving of some attention. One way to start investigating why DEVS is not part of the standard modeling toolkit of Software Engineers, and how to make it so is by researching on the relationships between DEVS and some formal methods listed in [8] –it is worth to say that this approach has been followed by people working from the other community [9]. In this paper we start to show how DEVS can be encoded in a standard, widely known formal method for Software Engineers called Temporal Logic of Actions Plus (TLA+) [2]. In fact, TLA+ was envisioned by its designer as a formal language (and latter a tool set) to be used by Software Engineers to model and verify complex, software or hardware based, reactive, real-time, concurrent systems. Since real-time, reactive systems are event driven systems with time requisites, and, on the other hand, being DEVS a formalism for the specification of discrete event systems, then it looks like both TLA+ and DEVS target the same class of problems, thus making them good candidates to compare each other.

Due to space restrictions and because this paper represents an ongoing research, it includes only shallow but broad issues regarding the relationship between DEVS and TLA+, with particular emphasis in encoding DEVS atomic models in TLA+ specifications. Particularly we restrict our present work to the use of DEVS to model discrete event systems and we left unattended its application to continuous systems [10], [11]. Besides, this paper does not introduce DEVS nor TLA+ beyond the features that are part of the comparison we report – readers may consult [1] and [2] in order to get a deeper insight of both formalisms.

The paper is structured as follows. In section II we discuss the scope of both languages, how their semantics are given and how them can help each other in the task of modeling and verifying event driven systems. Section III includes some of the rules to write a TLA+ specification from a DEVS model and the next section presents the application of these rules to a simple case study. Sections V and VI reports our future work and conclusions about this subject matter.

II. Semantics and Theoretical Issues

In this section we want to attack four issues –scope, semantics, liveness properties and initial states– regarding the theoretical differences and similarities between DEVS and TLA+, making some emphasis on their respective semantics models.

A. Scope

The first issue concerns the scope of both formalisms. The initial problem that Zeigler tried to solve was the formal modeling of *real world* systems and their simulation on a *computer*. On the other hand, Lamport wanted to give Software Engineers a tool to specify the functional properties of *concurrent computer* systems. Initially we are tempted to see a difference in scope of both formalisms but a deeper look should make us see that, in fact, this difference is somewhat apparent. This difference is not so relevant because, first, in

*This paper was written while the author was spending an invitation period at LSIS - UMR CNRS 6168 - University Paul Cézanne - France.

practice DEVS shortens its scope by prescribing a framework on which to model discrete event systems with complex timing requirements (for instance you cannot model the dynamics of the Solar system); and, second, because TLA+ is built over sufficiently general and abstract mathematical and logical theories that enable it to describe that class of systems. TLA+ has built-in modules describing axiomatic theories for the natural, integer and real numbers, it is based on a well-formed set theory (that sets the fundamentals for defining functions and operators) and its logical meaning comes from temporal logic. For example, there is no built-in concept of time in TLA+: if an engineer needs to specify a timing requirement, then she or he must include one or more variables representing time within the definition of the state of the system being modeled; but if there is no need for time, then the model will be simpler. At least in practice, and with scope in event driven systems, there is no limit on what can be specified with these tools –we hope that the encoding of a typical DEVS model in TLA+ in section IV will help to convince the reader. In fact, we use TLA+ to specify complex systems [12] following the methodology set by Jackson, Zave and others in [13], [14], [15]. This methodology, called WRSPM, prescribes to model part of the *domain knowledge* what implies to model so called *natural* properties of part of the *real world*, i.e. properties that are valid independently of the existence of any computer system.

B. Semantics

The semantics models of these formalisms is the second point that we want to talk about. DEVS semantics belongs to the class named operative semantics because DEVS models are understood by interpreting them with a simulation algorithm –others formalisms with a similar scope describe their semantics in a similar fashion [16]. In other words, the meaning of a DEVS model is given by the rules to simulate it on a computer or manually. On the other hand, TLA+ semantics is of the class known as logic semantics since a TLA+ specification gets its meaning from a temporal logic model. Each approach has its own advantages and disadvantages. The operative approach has a better learning curve, it is more intuitive, suggests an implementation and enables for mechanization or, as with DEVS, automatic simulation; but operative descriptions tend to be less formal impeding formal verification of models and opening the door to ambiguities and inconsistencies. A logic description of the semantics of a language sometimes is harder to learn because it is less intuitive, sometimes does not show a clear implementation, and full mechanization is mostly beyond scope. However, at the same time, a logic description is completely formal enabling formal verification and proof and model checking. Hence, if there is a chance to give a logic semantics of DEVS models, then it would be possible to get the best of both approaches. One way to do such a job is by translating DEVS models into TLA+ specifications. It is crucial that such a translation be *semantics preserving* in the sense that the set of valid properties given by the standard DEVS semantics is the same to the set derivable from the TLA+ semantics. But this is possible only if both semantics are completely formal which is not the present case. However, by looking at the problem from a slightly different perspective, an encoding of DEVS models in TLA+ specifications is a way to formalize the DEVS semantics. Then, by finding this encoding we are doing two things at the same time: first, DEVS semantics is formalized, and second, formal verification, model checking and other nice possibilities are enabled for DEVS models. Furthermore, an added value we obtain by encoding in TLA+ is that we place DEVS directly into the Alpern-Schneider framework for safety and liveness properties which is the dominant framework for specification in Computer Science [17].

C. Liveness Properties

Despite liveness properties are often considered less important than safety properties [2], according to Alpern and Schneider these properties complements the specification of any concurrent system. In fact, the functional description of any system is the result of intersecting a safety property with a liveness property [17]. Since DEVS models are a kind of generalized state machines, safety properties are described in the usual way –i.e. by giving the allowed transitions only. It is, thus, interesting to show how DEVS describes liveness properties and to compare this approach to the one prescribed in TLA+.

A liveness property prescribes that some states of interest must be reached eventually. DEVS uses the internal transition function to specify liveness properties. If it is necessary or natural that the system performs some action not initiated by its environment, then a DEVS model includes an internal transition that will be activated and executed at an *specified* point in time. Clearly, this schema is very general but this generality can be troublesome since, as Abadi and Lamport have shown [18], arbitrary liveness properties may lead to a system description that restricts some transitions that the safety property does not. TLA+ specifications avoid this situation intersecting a safety property with a *fairness* property. Fairness properties are a subclass of liveness properties that do not restrict transitions. In summary, DEVS generality for writing the liveness part of the system model, might lead to a wrong model.

Another minor issue regarding liveness properties in DEVS is that such properties are always tied to a timing constraint. In other words, if you want to specify a liveness property you must talk about time; you need to specify a particular future moment at which the transitions must be executed. TLA+ specifications allows you to say that a transition must happen in the future without saying exactly when it must happen. It would be nice to be able to write an initial system's abstract model without timing constraints.

D. Initial States

Despite that it would be erroneous to define an initial state for some abstract DEVS models, there are others for which to define their initial states is not only possible but essential – for instance, so called "experimental descriptions". One benefit from the encoding proposed below is that TLA+ specifications have a standard way to define the initial states of a system if it is necessary.

III. How to Encode DEVS Models into TLA+ Specifications

In this section we introduce some basic rules and design decisions to translate a DEVS atomic model into a TLA+ specification; these are not meant to be a complete set. Since TLA+ specifications have no explicit treatment of time and DEVS models do, we need to explain how time might be included in a TLA+ specification. On the other hand, to encode DEVS in TLA+ it is necessary to show how each element of a DEVS model has to be written in TLA+.

A. Real-time in TLA+

In [2] Lamport suggests a way to write real-time requirements in TLA+ specifications. In TLA+ timing requirements are set for transitions and not for states, and there is a state variable, named now, that records the time of the universe, i.e. the *real time*. Then, to specify a real-time requirement for transition T of a system with a state described by variable v, following predicate is used:

$$RTnow(v) \wedge RTBound(T, v, \delta, \epsilon) \tag{1}$$

which asserts, informally, that[1]:

- If now advances, then v remains unchanged (in $RTnow$).
- Transition T changes the value of v (in $RTBound$).
- Transition T cannot occur until T has been continuously enabled for at least δ time units since the last occurrence of event T, or since the initial state (in $RTBound$).
- Transition T can be continuously enabled for at most ϵ time units before a T step occurs (in $RTBound$).

In other words, the predicate says that when time advances nothing else changes and that T must happen at some time t in $[\delta, \epsilon]$ where both limits are taken since the last execution of T –or since the initial state. Hence, if for instance a requirement asks for the alarm to be activated 2 time units after it has been enabled, and $OnAlarm$ is the predicate describing the activation of the alarm (regardless of time), and a is the state variable describing the state of the alarm, then the formalization of the real-time requirement is:

$$OnAlarm \wedge RTnow(a) \wedge RTBound(OnAlarm, a, 2, 2)$$

Note that functional requirements ($OnAlarm$) and real-time requisites are described in separate formulas. This is good since there is a clear separation of concerns.

All of these definitions are assembled together in a TLA+ module named $RealTime$. Since DEVS models always contain real-time restrictions, then module $RealTime$ must always be included in the resulting TLA+ specification.

[1]This predicate is fully described and formalized in [2].

B. Mapping the Components of a DEVS Model into TLA+ Expressions

A DEVS model M is composed of seven elements:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{2}$$

We assume the reader is familiar with DEVS models so we will not explain them but we will map each of them into TLA+ expressions. Since DEVS semantics is not fully formal, part of this mapping entitles some degree of interpretation and subjectivity. However, at the same time, this interpretation gives a particular formalization of DEVS semantics; others should be possible and should be investigated. This first attempt is as follows.

M The model itself should be represented in a module structure as:

```
──────────── module M ────────────
extends RealTime
...
──────────────────────────────────
```

As we have said, it is always necessary to include $RealTime$ in order to be able to specify real-time requirements.

X These are the input values. In DEVS input values are (a) actual values and (b) external events. For instance, an input value might be the ASCII code of a character received by the system or the event $GoTo$ to move a robot arm to some position.

For (a) TLA+ offers two alternatives: (i) a variable defined in the VARIABLES section should be used (this variable should take values in the appropriate set); or (ii) some external transition depending on this value should have a parameter. The choice between these alternatives depends to a great extent on the way the engineer wants to build the model. External events in (b) must be encoded as TLA+ transitions or actions. In all cases all these definitions are included inside the module. The aforementioned examples are as follows:

```
────────── module Example1 ──────────
extends RealTime
VARIABLES code
TypeInv ≜ code ∈ 0..255
─────────────────────────────────────
GoTo(pos) ≜ pos > 0 ...
─────────────────────────────────────
```

S These are the state values. Usually a DEVS definition of S is given by means of a Cartesian product between all the sets describing the attributes of the *phases* plus their lifetimes. In our TLA+ encoding one or more state variables, ranging over appropriate sets, must be defined for each of the attributes of the phase. The lifetime will be encoded in now, the definition of external and internal transitions, and a new state variable as we will show. For example, the state of the conveyor belt can be defined by state variable $belt$, and the name of a job being executed by some operating system can be another state variable named job. More pictorially:

```
──────── module Example2 ────────
CONSTANTS JOBID
VARIABLES belt, job
TypeInv ≜ ∧ belt ∈ {"running", "quiet"}
          ∧ job ∈ JOBID
─────────────────────────────────
```

Note that the set $JOBID$ (of job names) was defined in the CONSTANTS section.

Y Output values should be encoded as values of output variables. The rest of the output definition is described in λ.

δ_{int} This is the internal transition function. Internal transitions are executed by the system when the lifetime of the current state is reached. Since internal transitions are a subclass of transitions, we use TLA+ actions to codify them. To say that an internal action must be executed when the lifetime of the current state is reached, we will use a real-time formula as described in section III-A.

Let us see a small example. Say there is just one internal transition defined from any state as follows: when the lifetime of the state is reached the system must transition to a particular state with an infinite lifetime. Then, first we need a name for this transition, let us call it $Error$. Second, we need to identify the erroneous state; say this state is reached when the state variable st equals "err". Then, a *first* TLA+ encoding is as follows.

```
──────── module Example3 ────────
extends RealTime
VARIABLES st
├───────────────────────────────┤
Init  ≜ ...
Error ≜ st ≠ "err" ∧ st' = "err"
Next  ≜ Error ∨ ...
Spec  ≜ ∧ Init ∧ [□Next]_st ∧ RTnow(st)
        ∧ RTbound(Error, st, ta[st], ta[st])
─────────────────────────────────
```

where $Init$ formalizes the initial state, $Next$ may contain another external transitions, and ta is defined below.

However, the execution of an internal transition might enable, in particular, some external transitions. Due to some reasons that we will explain below, we need to include a new state variable, called cpt, to hold the value of now right after the execution of any transition. Hence, the *final* encoding of $Error$ is as follows[2]:

$$
\begin{aligned}
Error \;\triangleq\; & \land\; status \neq \text{"err"} \\
& \land\; status' = \text{"err"} \\
& \land\; cpt' = now
\end{aligned}
$$

Note how the functional aspect of $Error$ is separated from its timing requirements. Also, since internal transitions do not depend on inputs, then the corresponding TLA+ action's specification should not contain predicates depending on input variables.

δ_{ext} External transitions can be fired only if the lifetime of the current state has not been reached yet; and also they will not happen necessarily. Then, on one hand, we have a sort of timing requirement; and on the other hand, we cannot add a liveness condition to an external transition. Moreover, by the way TLA+ describes real-time requirements, there is no way to know the elapsed time since the last transition. Hence, to encode DEVS' external transitions in TLA+ we need to introduce a new state variable to count the time elapsed since the last transition; we call this variable *check-point time* or cpt. cpt has to be "updated" by every (external and internal) transition with $cpt' = now$. Then, every external action must include a precondition of the form $now - cpt \leq ta[vars]$ where $vars$ is the tuple of all state variables. It is worth to note that this last condition formalizes a rule of the DEVS semantics. DEVS external transitions depend on input values. Our TLA+ encoding codify this dependency explicitly. For instance, assume that a robot's arm must be moved upwardly p space units when the external event $MoveUp$ is received but it can be moved only if it does not go beyond its maximum position. Hence, the TLA+ specification is as follows:

```
──────── module Example4 ────────
extends Naturals, RealTime
CONSTANTS MAXUP
ASSUME MAXUP ∈ Nat ∧ MAXUP > 0
VARIABLES cpt, pos
├───────────────────────────────┤
MoveUp(p) ≜ ∧ pos + p ≤ MAXUP
            ∧ now − cpt ≤ ta[vars]
            ∧ pos' = pos + p ∧ cpt' = now
─────────────────────────────────
```

See more on the encoding of external transitions in section III-D below.

λ This is a function from the set of states of the system onto the set of system outputs. DEVS semantics says that if an internal transition is executed from state s then the system outputs $\lambda(s)$. External transitions do not produce output. There is no special gesture or convention in TLA+ for modeling output. Usually some variables are defined within the module representing output devices, channels or whatever. In other words, module variables are no only for capturing the

[2]We include just $Error$'s definition, the rest remains the same as in module $Example3$.

state of the system but also to represent the state of its environment.

Then, our encoding prescribes to model the output function λ explicitly in each internal transition definition. For example, if *Error* in module *Example*3 should also output the error number through channel *out*, then we change that module as follows:

module *Example5*

extends *RealTime*
VARIABLES *st, out*
$vars \triangleq \langle st, out\rangle$

$Error \triangleq \land\ status \neq \text{“err”} \land status' = \text{“err”}$
$\qquad\qquad \land\ cpt' = now \land out' = 2$

ta This function represents the lifetime of each state. Since it is a constant value, in the sense that its definition cannot be changed by internal nor external transitions, we will represent *ta* as one of the module's constants and in the ASSUME section we will give its definition. For example

module *Example6*

CONSTANT *ta*
ASSUME
$\land\ ta \in [T_1\times,\ldots,\times T_n \rightarrow \Re_0^+]$
$\land\ \forall x_1 \in T_1,\ldots,x_n \in T_n :$
$\quad ta[x_1,\ldots,x_n] =$
$\qquad \textbf{case}\ x_3 = \ldots \qquad \rightarrow \infty$
$\qquad\qquad x_2 \in \ldots \land x_5 \leq \ldots \rightarrow 0$

where T_i is the "type" of phase variable s_i; here only variables representing phase attributes are considered. Then the value of *ta* in a particular state is just $ta[\langle s_1,\ldots,s_n\rangle]$ –since the tag VARIABLES imposes no order over the variables being defined we must take care to apply *ta* consistently with respect to its definition.

This finishes the mapping of a DEVS model onto TLA+ expressions. Despite, the mapping presented so far is not completely formal, we think that the rules given so far are sufficiently clear to allow an engineer to translate his or her DEVS models.

C. A Note on Distinguishing Internal and External Transitions

By reading the TLA+ encoding of a DEVS model it is no always easy to know which are internal and external transitions. Internal transitions are controlled or initiated by the system; external transitions are controlled or initiated by the environment. We propose, by following [15], [13], to complement the TLA+ formal description with a so called *control table* in which each action is classified as internal (machine controlled) or external (environment controlled).

D. A Note on the Interpretation of External Transitions

In a DEVS model the reception of an external event is always enabled since the model cannot control when the environment will produce the event. Our encoding transforms the reception of an external event in the execution of a corresponding (external) action. For instance, the reception of event MoveUp in a DEVS model, is encoded as the execution of action *MoveUp* in the TLA+ specification (cf. module *Example*4 above). Due to the semantics of a TLA+ specification an action such as *MoveUp* is enabled only if its preconditions are met. Hence, it may appear that it is possible for the DEVS model to receive event MoveUp in more states that it is possible to execute action *MoveUp*. However, despite the event can be received in any state, the DEVS model only prescribe a meaningful behavior when the event is sensed in a subset of those states; in the other cases, the model transitions to a new state that differs only in its lifetime – i.e. $\delta_{ext}(s,\sigma,e,x) = (s,\sigma - e)$. Our encoding describes only those meaningful behaviors.

IV. A Case Study

In this section we will show an application of the encoding presented so far. First, we will show a very simple DEVS model, and then we will apply the rules to encode it as a TLA+ specification.

A. The DEVS Model

The following DEVS model represents a robot's arm that has to be moved from a conveyor belt to a press. If the external event *ToPress* is received and the arm is at the belt, then it starts to go to the press. When 5 time units have elapsed since the arm started to go from the belt to the press, it must be stopped and the signal *Stopped* must be outputted –we assume that the arm remains at the press for ever. In this way, we have one external transition, one internal transition, one output, and one real-time requirement which is quite representative of the kind of systems modeled with DEVS.

$$
\begin{aligned}
&Arm = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta\rangle\\
&X = \{ToPress\}\\
&Y = \{Stopped\}\\
&S = \{AtBelt, Moving, AtPress\} \times \Re_0^+\\
&\delta_{int}(s,\sigma) = (AtPress, \infty),\ \ s = Moving\\
&\delta_{ext}((s,\sigma), e, ToPress) =\\
&\quad \begin{cases} (Moving, 5), & s = AtBelt\\ (s, \sigma - e), & s \in \{Moving, AtPress\}\end{cases}\\
&\lambda(s,\sigma) = Stopped,\ \ s = Moving\\
&ta(s,\sigma) = \sigma
\end{aligned}
$$

B. The TLA+ Specification

The TLA+ specification that encodes DEVS model *Arm* by following the rules introduced in section III-B is as follows.

module *Arm*

extends *RealTime, Reals*
CONSTANTS *ta, S,* $\Re_0^+$
ASSUME
$\land\ S = \{\text{“AtBelt”}, \text{“Moving”}, \text{“AtPress”}\}$
$\land\ ta \in [S \rightarrow \Re_0^+]$

$$\begin{aligned}&\land\ \forall s \in S:\\ &\qquad ta[s] = \textbf{case}\ s \in \{\text{“AtBelt”}, \text{“AtPress”}\} \ \rightarrow\ \infty\\ &\qquad\qquad\qquad\quad s = \text{“Moving”} \qquad\qquad\quad \rightarrow\ 5\end{aligned}$$

$$\begin{aligned}&\text{VARIABLES}\ \mathit{cpt, pos, out}\\ &\mathit{vars}\ \triangleq\ \langle cpt, pos, out\rangle\\ &\mathit{NoVal}\ \triangleq\ \text{CHOOSE}\ x \neq \text{“Stopped”}\\ &\mathit{TypeInv}\ \triangleq\ \land\ pos \in S \land out \in \{\text{“Stopped”}, NoVal\}\\ &\qquad\qquad\quad \land\ cpt \in \Re_0^+\end{aligned}$$

$$\begin{aligned}&\mathit{Init}\ \triangleq\ cpt = now \land pos = \text{“AtBelt”} \land out = NoVal\\ &\mathit{ToPress}\ \triangleq\ \land\ pos = \text{“AtBelt”} \land now - cpt \leq ta[pos]\\ &\qquad\qquad\quad \land\ pos' = \text{“Moving”} \land cpt' = now\\ &\qquad\qquad\quad \land\ out' = out\\ &\mathit{Stop}\ \triangleq\ \land\ pos = \text{“Moving”} \land pos' = \text{“AtPress”}\\ &\qquad\quad \land\ out' = \text{“Stopped”} \land cpt' = now\\ &\mathit{Next}\ \triangleq\ ToPress \lor Stop\\ &\mathit{Spec}\ \triangleq\ \land\ Init \land [\Box Next]_{vars} \land RTnow(vars)\\ &\qquad\quad \land\ RTBound(Stop, vars, ta[pos], ta[pos])\end{aligned}$$

As we propose ta is defined as a constant function depending only on the phase. State variable cpt is introduced to count the time elapsed since the last transition, initially equals now and it is updated in the post-condition of every transition. The system outputs on state variable out following the encoding for λ; besides, internal transitions such as $Stop$ are the only ones which can modify the value of this variable. The initial state –actually we should say the initial phase– is defined in predicate $Init$ as is customary in TLA+ specifications.

As we propose, the external transition due to the reception of input $ToPress$ is codified as an action with the same name. The alternative guarded by $s \in \{Moving, AtPress\}$ is not explicitly codified although it can be done as:

$$\begin{aligned}&ToPressOk\ \triangleq\ as\ ToPress\ in\ module\ Arm\\ &ToPressE\ \triangleq\ pos \neq \text{“AtBelt”} \land \text{UNCHANGED}\ vars\\ &ToPress\ \triangleq\ ToPressOk \lor ToPressE\end{aligned}$$

Finally, the "functional" aspects (i.e. *what*) of the internal transition are described in its definition, while the timing conditions for its execution (i.e. *when*) are codified in predicates $RTnow$ and $RTbound$ as we suggest.

V. Future Work

This paper is the result of an ongoing research, then there is a lot of work to do in the future. First, we need to formalize the encoding given so far as the first step towards the construction of a translation tool. Then, we need to investigate whether all DEVS model can be transtaled or not, and in this case under what conditions the encoding works. Particularly, we would like to study how DEVS coupled models can be encoded in TLA+. Then, continuous systems might be considered but we do not foresee to much future in this direction.

VI. Conclusions

The main conclusion of this work is that DEVS models describing discrete event systems can be easily translated into TLA+ specifications. This translation is not only possible but beneficial for DEVS since it lays the basis for a formal semantics of this powerful modeling language. Having a TLA+ specification of a DEVS model enables for formal verification of the model or to model-check it with the tools already available for TLA+ specifications. In fact, our encoding has made explicit some important rules to simulate a DEVS model. However, the resulting TLA+ specification shows some inconveniences that should be either tolerated or improved with future research –for example, the TLA+ specification looses the clear distinction between internal and external transitions enforced in the DEVS model.

Acknowledgments

Norbert (Giambiasi) thanks a lot for encouraging me to write this article. Claudia (Frydman), hope sometime I can give you something in return for inviting me several times to LSIS. Hernán (Dacharry) thanks for helping me to understand DEVS. Thanks to the referees for their comments.

References

[1] B. P. Zeigler, *Theory of Modelling and Simulation*. Robert F. Krieger Publishing, 1976.

[2] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[3] D. Craigen, S. Gerhart, and T. Ralston, "An international survey of industrial applications of formal methods," National Institute of Standards and Technology, Tech. Rep. NIST GCR 93/626-V1 & NIST GCR 93-626-V2, 1993.

[4] J. Bowen, "Formal methods in safety-critical standards," in *Proc. Software Engineering Standards Symposium (SESS'93)*. Brighton, UK,: IEEE Computer Society Press, Sept. 1993, pp. 168–177.

[5] M. Hinchey and J. Bowen, *Industrial-Strength Formal Methods in Practice*, ser. Formal Approaches to Computing and Information Technology. Springer-Verlag, 1999.

[6] E. Clarke and J. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 18, no. 4, pp. 626–643, Dec. 1996.

[7] P. E. Ross, "The exterminators," *IEEE Spectrum*, vol. 42, no. 9, pp. 30–35, Sept. 2005.

[8] J. Bowen, "Formal methods," http://vl.fmnet.info/.

[9] H. P. Dacharry and N. Giambiasi, "From Timed Automata to DEVS models: Formal verification," in *SMC 2005 Spring Simulation Multiconference*, 2005.

[10] E. Kofman, "Quantized-state control. a method for discret event control of continuous systems," in *Latin American Applied Research*, 2001.

[11] N. Giambiasi, B. Escude, and S. Gosh, "GDEVS: A generalized discrete event specification for accurate modeling of dynamic systems," *Transactions of SCS*, vol. 17, no. 3, pp. 120–134, 2000.

[12] M. Cristiá, "Material de la asignatura Análisis de Sistemas, LCC, FCEIA, UNR," http://www.fceia.unr.edu.ar/asist.

[13] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 1, Jan. 1997.

[14] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Software*, vol. 17, no. 3, pp. 37–43, May 2000.

[15] M. Jackson, *Software Requirements and Specifications*. ACM Press, 1995.

[16] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[17] B. Alpern and F. B. Schneider, "Defining livennes," *Information Processing Letters*, vol. 21, no. 7, pp. 181–185, Oct. 1985.

[18] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, 1991.