CREST - A Continuous, REactive SysTems DSL

Stefan Klikovits Alban Linard Didier Buchs
University of Geneva, Geneva, Switzerland
firstname.lastname@unige.ch

***Abstract*—The advance of cyber-physical systems in everyday life requires powerful modeling capabilities. Existing formalisms often have severe limitations and require complicated notations. In this paper we introduce CREST, a domain-specific language for modeling entity behavior and resource transfers in CPS. CREST aims to support CPS architects through clarity, comprehensiveness and analyzability.**

1. Introduction & Motivation

The advance of cyber-physical systems (CPS), unions of hardware sensors and actuators and software controllers, emphasizes the importance of analysis, validation and verification of such systems. The complexity of CPS often lies in their systems-of-systems architecture and mutual subsystem influences. The analysis and development of CPS require a clear and comprehensive means of modeling.

In this paper we introduce CREST, a domain-specific language (DSL) for continuous and reactive systems. CREST focuses on the specification of CPS entities and resource influences between them. The language features state-based entity behavior and continuous resource transfers. Through CREST's homogeneous entity definition, the language encourages modular system-of-systems designs.

We will introduce CREST's features using a plant growing system. This setup consists of several components (growing lamp, plant) and different kinds of resources (light, heat, electricity) that are transferred. Figure 1 shows a schematic representation of this scenario. Within we find hierarchical components, continuous value updates, discrete behavior and conversions between physical units. For example, the lamp's output, measured in lumen, influences the plant's light input, which is measured in lux. The calculation of the light impact requires various parameters such as distance, illuminated surface and illumination angle.

While usually a CPS model's purpose is simulation, CREST additionally aims to provide various other usage possibilities. These include the validation of CPS using the mathematical equations that describe physical processes, the efficient scheduling of inputs to the system, the synthesis of software controllers and the formal verification of CPS. CREST aims to facilitate CPS modeling using features such as continuous evaluation of variable values, reactive resource/data passing and a system-of-systems architecture.

The rest of this paper is organized as follows: Section 2 reflects on the current state of the art in CPS modeling.

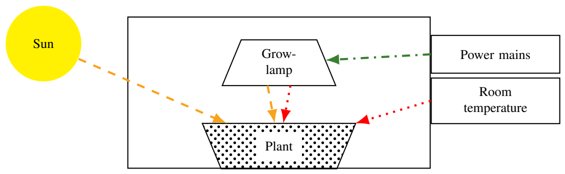

Figure 1: Schematic representation of the plant growing study. Arrows represent resource influences:
light - - ▸, heat ·····▸, electricity -·-▸

Section 3 introduces CREST and its graphical language. Section 4 outlines rules for exchange and re-use of CREST components. Section 5 formalizes CREST's syntax. Section 6 explains how to use CREST for validation, simulation, planning and controller synthesis and Section 7 concludes.

2. State of the Art

Systems modeling has been an active field of study for decades. Our research touches many areas of the modeling domain for instance concurrency, transaction and composition [1], [2]. We give an excerpt of related works.

While in the software area UML 2 [3] or SysML [4] are commonly used, the modeling of hardware behavior requires additional information. MARTE [5] has been used to model real-time behavior, timing constraints and similar embedded systems concepts. MARTE is a powerful means to model hardware and software platforms but features a very broad spectrum of concepts and diagrams.

Architecture Description Languages (ADLs) have been used to describe the connections of components using connectors and interfaces. There are several well-known representatives for ADLs similar to CREST, which have been employed for CPS. AADL [6] is very powerful and allows analyses for feasibility, optimization, timing of compositions of hard- and software. π-ADL [7] is based on the π-calculus and introduces typed ports and their verification. CREST differs from most ADLs by providing execution semantics that allow verification and simulation, as opposed to a pure architectural description.

Hardware modeling languages such as VHDL [8] or Verilog [9] mostly focus on low-level description of electronic systems. Bond graphs [10], state-space representations [11] and similar allow for the precise modeling of physical aspects within dynamic systems. They abstract away from

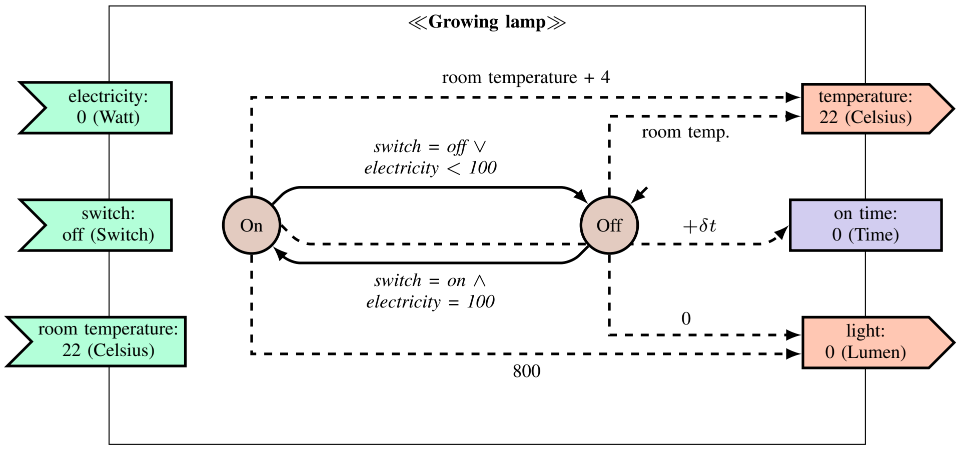

Figure 2: A basic CREST entity with three inputs, two outputs, two states and transitions, one local and update functions

real-world representation and describe pure physical system behavior. They however require a precise knowledge of the exact behavior. Further, it is difficult to represent real-world systems that are influenced by digital information or on/off switches for example.

There exists a plethora of tools for the design, simulation and analysis for embedded systems and models. Examples are Ptolemy II [12], 20-sim [13] and Simulink [14]. They support of a wide variety of different systems and offer large libraries of pre-defined entities and components. In our use-case they present themselves as potential simulation and state-space exploration platforms.

Finally we relate our approach to the Discrete EVent System specification (DEVS) [15]. DEVS can be used to model discrete event systems using time stamps. It features hierarchical composition and internally uses a state-based approach. Information between subsystems is passed through events and transitions fire according to an internal time advance function or external events. Due to its nature continuous value updates or internal events are not possible.

CREST uses specific terms such as*entities*, *influences* and *resources* for domain concepts. They might be related to concepts of other, more generic languages (e.g. *components*, *connections*, *types*). As CREST is a DSL however, we choose to use terminology that closely describes its purpose.

3. CREST Language

To satisfy the need for a continuous, reactive language we developed CREST. CREST has three main notions: *resources*, defining measurement units and the domains that values can take, *entities*, representing physical or logical objects, and *influences*, modeling relations between entities.

Resources are value types such as physical units or signals that can be transferred using influences. In our system we use several resources for modeling data transfers, such as Time (value domain: $\mathbb{R}^{+}$) for time measures in hours, Lumen ($\mathbb{N}$) for luminous flux, Switch ([On,Off]) for an electrical device's switch.

Entities are graphically depicted as CREST diagrams. The model for our system's growing lamp is presented in Figure 2. The lamp's interface is defined by its *inputs* (`switch` and `electricity`) and *outputs* (`temperature` and `light`), which respectively represent the sources and targets of resource or signal transfers between entities. Each of these so-called *ports* is linked to a resource and has a *value* from the resource's domain.

An entity's behavior is specified by an automaton (or a formalism that maps to automata, e.g. Petri nets to model concurrency), consisting of *states* and *transitions*. Our lamp has two states: `Off` and `On`, a short arrow points to the initial/current state. Transitions are represented as arrows between states and annotated with *guard* conditions (Boolean queries on port values). Transitions trigger as soon as their guards evaluate to *true*.

Internally the lamp defines a third type of port, namely a local variable, that can be used to store information. Locals can only be accessed by the entity itself. The lamp defines `on time` to measure the total amount of time it was turned on. Local port and output values can be changed using update functions (- -►). These functions are continuously evaluated if the automaton is in the corresponding state. In order to allow analysis and verification of CREST models, updates have access to a δt variable that represents the duration since the last call of the update. This permits simulation (value discretization by time steps) and verification (topological discovery of the significant behaviour times). This aspect of the modeling makes link between continous and discrete aspect of time.

Entities can be grouped together within an enclosing entity. A system is thus a hierarchical system-of-systems and represented by one "parent"-entity. Figure 3 displays the plant growing system, an example for such a composition. A composed entity can, additionally to the already introduced features (ports, locals, automaton), use other entities as subentities. The subentities' interfaces are linked via *influences*. Influences link between child-entity interfaces, as well as between the parent's ports and the ports of its direct children. As influences denote the transfer of resources, it is

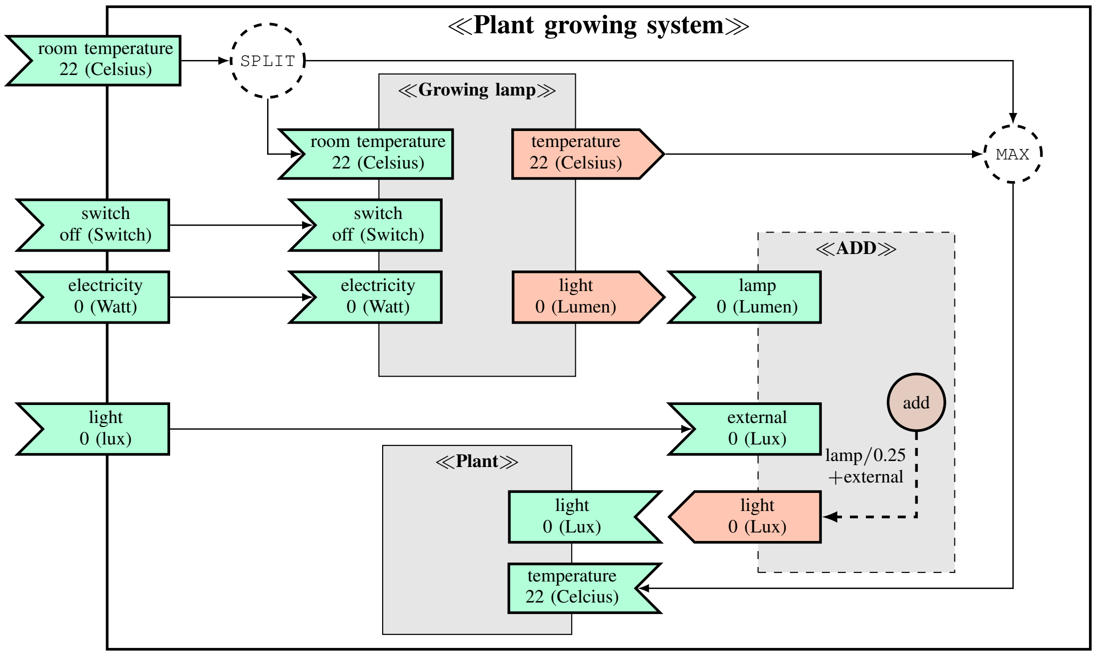

Figure 3: An entity that is composed of two subentities. To increase legibility the internal behavior of the Growing lamp, Plant, SPLIT and MAX was omitted.

the responsibility of the modeler to specify a transformation of the source's resource to the target's resource. Also note, that CREST only permits a maximum of one input and output per interface port. This is uncommon for modeling languages which usually define specific connectors (e.g. additive, averaging) for such cases. To overcome this restriction we define logical entities (entities without real-life counterpart) such as *splitters*, *adders* and similar to divide and combine resource signals. This constraint facilitates analyzability and verification of CREST models.

In the example in Figure 3, the growing lamp's light output is specified in lumen, but the plant's input in lux. The plant's total light exposure is the lamp's lumen value divided by the illuminated surface (0.25 m^2), added to the `external` light. The calculation is performed in a *logical entity* ≪`ADD`≫ (dashed edge). It features one state (`add`) and one update function that continuously modifies the output value. Figure 3 shows two more logical entities (`split` and `max`) that send signals to multiple targets and select the highest value, respectively. These are, due to spatial constraints, displayed in abstract form as dashed circles.

4. Entity Compatibility and Replacement

CREST's modularity and hierarchical specification concept facilitates the reuse, exchange and modification of systems. When it comes to replacing an entity with another component it is important to assert the interface compatibility.[1]

1. We do not discuss behavioral or semantic compatibility, as e.g. described in [16].

When it comes to compatibility of interfaces we require two consistency properties. Firstly, the number of required and provided ports needs to be consistent. This means that an entity can be replaced if it requires the same number or less input ports and if it provides at least the same outputs that the original entity provided. The reasoning is that in order to define a sound replacement a new component must not specify more requirements or provide less services than the original. This concept resembles a form of subtyping relationship that is well-known in the software engineering domain as *Liskov's substitution principle* [17].

Figure 4a depicts an example of an entity with two inputs and two outputs. Of the four candidates for replacement only 4b and 4e are directly acceptable as replacements since they require fewer or provide more services to the system. 4c demands an interface that might not be available and 4d does not provide an output that is potentially required. The two rejected entities can be used if the former's new input is optional (e.g. a port for optional debug information) and the latter's missing output was not connected to any other entity.

Secondly, all inputs and outputs need to accept and provide the same values as the original, respectively.

5. CREST Formalization

In the following section we provide the formal definition of a CREST model. While the graphical notation is primarily used for system development, we provide the syntax below in a top-down approach.

We define a system to be a quadruple $\langle \mathbb{T}, \mathcal{R}, \mathcal{E}, e \rangle$, where $\mathbb{T}$ is the system's time-base values (e.g. $\mathbb{T} = [0, \inf)$), $\mathcal{R}$ is

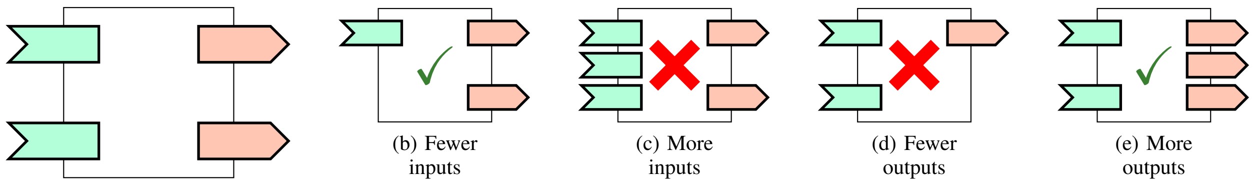
(a) Original (b) Fewer inputs (c) More inputs (d) Fewer outputs (e) More outputs

Figure 4: An entity with two inputs and outputs (a) and several replacement candidates (b) - (e)

a set of resource types, $\mathcal{E}$ a set of entities within the system, as defined inductively below, and $e \in \mathcal{E}$, the system's root element. Below we introduce several classical notations.

Definition 1. *Resources are defined by a set of resource types $\mathcal{R} = \{R_1, \dots R_n\}$. Each type R_i represents a set of possible values for a specific resource unit, for instance electricity in Watts or the state of a switch (on/off). The set of all resource values R is defined as $R = \bigsqcup_i R_i$.*

The distinction of both the resource type (electricity) and its unit (Watts) is important to avoid conversion problems, which are known to create failures in CPS, such as in the Mars Climate Orbiter [18].

Each $e \in E$ within the set of entities E describing the system is defined as:

Definition 2. $e = \langle P_e, resource_e, TS_e, U_e, entities_e, Inf_e \rangle$, *where TS_e is a transition system, U_e the update functions and Inf_e the influences within e.*

An entity e's set of ports (P_e) is the disjoint union of inputs, outputs and locals. Each port is assigned a resource using a *resource* function.

Definition 3. *$P_e = I_e \sqcup O_e \sqcup L_e$, where I_e are inputs, O_e outputs and L_e local variables. The function $(resource : P_E \to \mathcal{R})$ assigns a resource to each port.*

Entity behavior is defined by a transition system TS of states and transitions.

Definition 4. *$TS = \langle S_e, \to_e \rangle$, where S_e are the entity's states, and $\to_e$ is guarded transition relation between states $(\to_e \subseteq S_e \times S_e \times G_e)$.*

Guards G_e are function names whose corresponding functions can be evaluated using a function *eval*. An evaluation with a binding the function returns a Boolean value.

Definition 5. $eval : G_e \times B_e \to \mathbb{B}$.

We do not provide a full syntax for guards here, due to spatial constraints. We impose no limitation on guard expressions, as long as they return boolean values.

Bindings B_e are total functions mapping ports to correctly typed resource values.

Definition 6. *The set of possible bindings B_e is defined as the set of possible mappings:*

$$B_e : P_e \to R \;\; s.t. \;\; \forall p \in P_e, p \mapsto r \in resource(p)$$

Updates assign a function name to a state. The corresponding functions continuously modify local variables and output values as time passes in a state. An update's evaluation is applied to binding, a port (either output or local), and an elapsed time. The evaluation returns the computed value for the port.

Definition 7. *The updates of an entity e are given by $U_e : S_e \to F_e$, where F_e is the set of function names* [2]. *Evaluation is performed by:*

$$eval : F_e \times B_e \times (O_e \cup L_e) \times \mathbb{T} \to R$$
$$b, p, t \mapsto resource(p)$$

where b is a binding, p a local or output port and t the time that has passed since the last update. All applicable updates are performed concurrently, updating their referenced ports at the same time.

An entity's direct subentities are defined by the *entities* function.

Definition 8. *$entities : E \to \mathcal{P}(E)$, where $\mathcal{P}(E)$ is the power-set of E.*

We also demand that the subentity structure forms a rooted tree.

The composition of entities is based on the concept of influences. Influences represent resource/signal transfers between and entity's ports and its subentities' ports and amongst an entity's subentity ports. Formally, an entity e's influences Inf_e is a function that, given two ports, returns a translation function between source's values to the target's values.

Definition 9.

$$Inf_e : \left(I_e \cup \bigcup_{e' \in entities(e)} O_{e'} \right) \times \left(O_e \cup \bigcup_{e' \in entities(e)} I_{e'} \right) \to T_e$$

where T_e is a homomorphism between the source s and target t of the resources:

$$T : R \to R, s.t.$$

Definition 10. *The signature of the translation function must conform to the resources in the ports of the influence, thus*

$$\forall \langle s, t \rangle \mapsto f \in Inf_e, f : resource(s) \to resource(t)$$

2. Graphical CREST can annotate updates with the function itself instead of its name.

We demand ports to only have one incoming and one outgoing influence.

$$\forall p, q \in P_e : \Big(|\langle p, x, y\rangle \in Inf_e| \leq 1\Big) \wedge \Big(|\langle x, q, y\rangle \in Inf_e| \leq 1\Big)$$

This increases uniformity as signal splitters and combinators have to be modeled as (logical) entities instead of separate port behavior.

Semantics. CREST's formal semantics are beyond the scope of this publication. We will however outline it as follows: An entity's behavior is controlled by reactive port observation, continuous update function execution and transition guard evaluation. The inter-entity semantics link ports and modify resource transfers using influence functions, such as the lumen-lux conversion.

6. Advanced Usage of CREST

While the modeling and graphical visualization of CPS greatly facilitate communication and clarify the flow of resources throughout the system, CREST aims to address some even more important purposes.

6.1. Validation

CREST was designed with validation in mind. The concept of specifying a continuous system using mathematical equations (in update functions and guard conditions) instead of defining state transitions based on pre-defined time advance functions (as in DEVS for instance), allows for the calculation of the exact moment when a guard condition is satisfied. An example for this is the calculation of the time when a transition fires. Assuming a water tank entity that changes to state `full` when a local variable reaches a certain value. CREST can continuously track the fill-level over time (*fill-level* $=$ *water_in*$*\delta t$) and trigger the transition as soon as the guard condition applies. Discrete formalisms on the contrary only provide certain time slots to test guard conditions.

For the validation of CREST behavior, i.e. the automaton, we intend the use of region-based verification, as known in timed automata [19]. By splitting the variable value space into equivalence regions, using the guard conditions as discriminator one can reduce the number of testing regions.

6.2. Controller Synthesis

Based on the behavior information encoded within a CREST model we plan on synthesizing software controllers that manage the dynamic behavior of the system. As visible in the plant growing system, certain inputs (e.g. lamp and heating switches) and outputs (electricity consumption, temperature readings, light exposure) would usually be read and set by a software program. These readings and controls could be displayed via a user interface in the form of a *supervisory controls and data acquisition* (SCADA) software, or, be used to autonomously issue system commands for predefined programs.

6.3. Simulation, Planning, Optimization

While the execution of an individual entity's model might be used for behaviour visualization, CREST's goal is to aid the development of multi- and many-entity assemblies. Simulation of such compositions can help system architects discover optimized parameters within a state-space. Applied on the plant growing example a simulation can provide lamp schedules to optimize electricity consumption or lamp placements for ideal light exposure. CREST models can be translated to existing formalisms and simulation platforms such as 20-sim and Simulink.

The scheduling of actions can also be expressed as a planning domain. In the context of the International Planning Competition (IPC) the Planning Domain Definition Language (PDDL) [20] was defined to standardize their definition. CREST models can be translated to PDDL (after discretization) in which scheduling and optimization problems can be solved.

7. Conclusions & Future Work

This paper introduces CREST, a domain-specific language for the modeling of continuous, reactive systems. CREST focuses on modeling the physical transmission of resources and signals between system entities. It features automata for the representation of entity states and input/output ports for the transfer of information. Using a reactive mapping of port values and continuous variable updates, a discretization of behavior, as in other formalisms, is avoided. In fact, a discretization is only needed for the execution and simulation of the model.

CREST's hierarchical, composition-based approach allows developers to create models that closely reflect a real-world system. This feature facilitates extension and re-use of models, and the exchange of subsystems.

The language is an ideal basis for further research. In particular, we aim to advance our efforts into area of automatic model validation. Next to that, our efforts go towards automatic synthesis of control software that allow for system optimizations. This means that an eventual controller should operate autonomously and perform choices that will optimize a choice of parameters in the system. Such optimizations include the creation of best operation conditions for all entities or the minimization of resource usage. Extensions of this approach could involve the addition of learning models that adapt the entity's behavior according to lessons learned. An example is a growing system where a plant is monitored and the plant model adapted according to the observations.

Acknowledgements
This project is supported by: FNRS STRATOS : Strategy based Term Rewriting for Analysis and Testing Of Software, Hasler Foundation, 1604 CPS-Move and COST IC1404: MPM4CPS

References

[1] O. Biberstein, D. Buchs, and N. Guelfi, “Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism,” in *Advances in Petri Nets on Object-Orientation*, ser. LNCS. Springer, 2001, pp. 70–127.

[2] D. Buchs, S. Chachkov, and D. Hurzeler, “Modelling a Secure, Mobile and Transactional System with CO-OPN,” in *Proc. Int. Conf. on Application of Concurrency to System Design, Guimarães, Portugal*. IEEE CS Press, 2003, pp. 82–91.

[3] Object Management Group, *Unified Modeling Language (UML) Specification. Version 2.5*, Mar. 2015, OMG Document formal/2015-03-01.

[4] ——, *OMG Systems Modeling Language. Version 1.5*, May 2017, OMG Document formal/2017-05-01.

[5] ——, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Version 1.1*, Jun. 2011, OMG Document formal/2011-06-02.

[6] P. Feiler, D. Gluch, and J. Hudak, “The Architecture Analysis & Design Language (AADL): An Introduction,” Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2006-TN-011, 2006.

[7] F. Oquendo, “Pi-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures,” *SIGSOFT Softw. Eng. Notes*, pp. 1–14, 2004.

[8] P. J. Ashenden, *The Designer’s Guide to VHDL*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[9] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, ser. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1996.

[10] H. Paynter, *Analysis and Design of Engineering Systems: Class Notes for M.I.T. Course 2,751*. M.I.T. Press, 1961.

[11] K. Hangos, R. Lakner, and M. Gerzson, *Intelligent Control Systems: An Introduction with Examples*, ser. Applied Optimization. Springer, 2001.

[12] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[13] C. Products, “20-sim,” http://www.20sim.com/, 2017, accessed: 2017-07-12.

[14] MathWorks, “Simulink: Getting Started Guide, Accessed: 2017-07-12,” http://www.mathworks.com/help/pdf_doc/simulink/sl_gs.pdf, 2017.

[15] B. P. Zeigler, *Multifaceted Modelling and Discrete Event Simulation*. London: Academic Press, 1984.

[16] M. Weidlich, R. Dijkman, and M. Weske, *Deciding Behaviour Compatibility of Complex Correspondences between Process Models*. Springer Berlin Heidelberg, 2010.

[17] B. H. Liskov and J. M. Wing, “A Behavioral Notion of Subtyping,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[18] Mars Climate Orbiter Mishap Investigation Board, “Mars Climate Orbiter Mishap Investigation Board Phase I Report,” NASA, Technical report, 1999.

[19] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, Apr. 1994.

[20] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “PDDL - The Planning Domain Definition Language,” Yale Center for Computational Vision and Control, Tech. Rep., 1998.