CREST Formalization *

Technical Report

Stefan Klikovits Alban Linard Didier Buchs

Software Modeling and Verification (SMV) Group

Faculty of Science, University of Geneva

Geneva, Switzerland

June, 2018

CREST is a novel modelling language for the definition of Continuous-time, REactive SysTems. This domain-specific language (DSL) targets small cyber-physical systems (CPS) such as home automation systems. While CREST is a graphical language and its systems can be visualised as CREST diagrams, the main form of use is as *internal* DSL for the Python general purpose programming language. Nevertheless, CREST systems are based on a formal structure and semantics. This report provides this formalisation and elaborates on the design choices that have been made.

1 Introduction

In this article we present the Continuous REactive SysTems (CREST) language. CREST is a domain-specific language (DSL) created for the modelling of CPS such as automated gardening systems or building automation. CREST particularly focuses on the creation of models that capture the reactive behaviour of a CPS' components and the flow of resources between its physical parts. Data is typed as resources such as light, electricity, heat or switch positions and data transfer as influences between system components ("entities"). CREST follows a strictly hierarchical system view that encourages composition and system-of-systems designs. Entity behaviour is modelled via automata and continuous value updates take real-valued time into account.

One of CREST's main features is the recognition of the complexity, concurrency and parallelism that is inherent to CPS. The language semantics guarantee a synchronous representation and evolution of the model, while still preserving dynamic behaviour with

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

arbitrary time granularity, as opposed to other formalisms, which demand the use a base-clock and fixed time steps. This allows the modelling, precise representation and faithful verification of the system.

CREST's syntactic structure and semantics were designed with six core principles in mind:

1. Assert the synchronism of CREST systems,
2. Preserve data locality for coherent modeling and efficient verification,
3. Support the concurrency and parallelism that is inherent to CPS,
4. Enforce reactive systems principles,
5. Enable continuous behaviour through arbitrary time granularity,
6. Support the inherent non-determinism of physical systems.

This paper provides details about how CREST's formalisation supports these principles and allows the precise verification and simulation of CPS.

The report is structured as follows: Section 2 elaborates on CREST and it's basic syntax and language structure. Since CREST uses under-specification for some parts of its syntax, Section 3 introduces semantic constraints that restrict this functionality. Section 4 offers information about extensions to the syntax, which enable the simplified description of CREST systems. Section 5 provides details about the formal, operational semantics of CREST.

2 Language Structure

CREST is a modelling DSL that combines a system's architectural and behavioural aspects. From an architectural point of view, CREST systems are strictly hierarchical compositions of entities, forming a tree-like structure. This means that every system has one root entity which may contain arbitrarily many child-entities ("subentities"), that possibly have subentities themselves, etc.

We will use the model of a plant growing lamp displayed in Figure 1 as a running example for CREST's syntax, structure and semantics. The example consists of a GrowLamp entity and two separate submodules for heating and lighting.

An entity can define *ports* to model the transfer of data and resources between its internal value storage and other entities. Ports are associated with resources, such as lumen, Celsius or "switch", which have a value domains (e.g. $\mathbb{R}$ for `lumen` or $\{on, off\}$ for `switch`). An entity's interface consists of *input* and *output* ports. *Local* ports are used to store data values internally. The growing lamp for example defines *on-time* to locally store a value.

The behaviour of CREST entities is defined by automata that consist of states and state transitions. In our example the lamp transitions between `Off` and `On` states. Each

transition defines a Boolean *guard* function (*off-guard*, *on-guard*). Transitions between states are enabled iff the guard functions (over an entity's port values) evaluate to `True`.

The example also shows the usage of *updates* (- -►). Updates are special functions that can modify port values. They are associated with automaton states. When the specified state becomes active, the update is continuously evaluated and the specified port is set to the function's result value. Updates can read ports values to perform calculations and have access to the time that has been passed since entering the state or since last evaluation of the update. They can therefore be used to model continuous changes over time within entities. In the growing lamp example updates are used to modify the values of *on time*, $electricity_L$ and $electricity_H$ when the lamp is turned on.

A port can only be modified by one update function at the same time. This means that per automaton state and port only one update function is allowed. This is to avoid non-deterministic setting of port values through write-race conditions. Further, cyclic update dependencies between ports are not allowed. This means that there cannot be an update function that reads a port A and writes a port B and another one that reads B and writes A. In order to resolve cyclic dependencies (such as algebraic loops), all ports define a *pre* value, that stores the port's value before modifying it through the update execution.

The limitations on the number of updates per port remove the need for a dedicated connector concept (e.g. adding all incoming port values or choosing the smallest/highest value) that defines the desired behaviour. Instead, CREST uses *logical* entities, such as the `Adder` in Figure 1. Such entities behave exactly as standard entities, but do not have an explicit physical counterpart.

Direct data transfer between two ports can be modelled using *influence* (—►) relations. Influences are special updates that are active in every automaton state and transfer the value of one specific port to another. Optionally, they can define transformation functions that convert from the source domain to the target domain. The growing lamp defines for example the `fahrenheit_to_celsius` transformation, which converts the growing lamp's `room temperature` value from Fahrenheit to Celsius as it updates the `temp-in` port.

An implementation of CREST is available as a Python-internal DSL. The preliminary interactive version is available online: `https://mybinder.org/v2/gh/stklik/CREST/sam-demo` [1].

[1] Launching might take one or two minutes to build. We thank BinderHub for hosting their service for free.

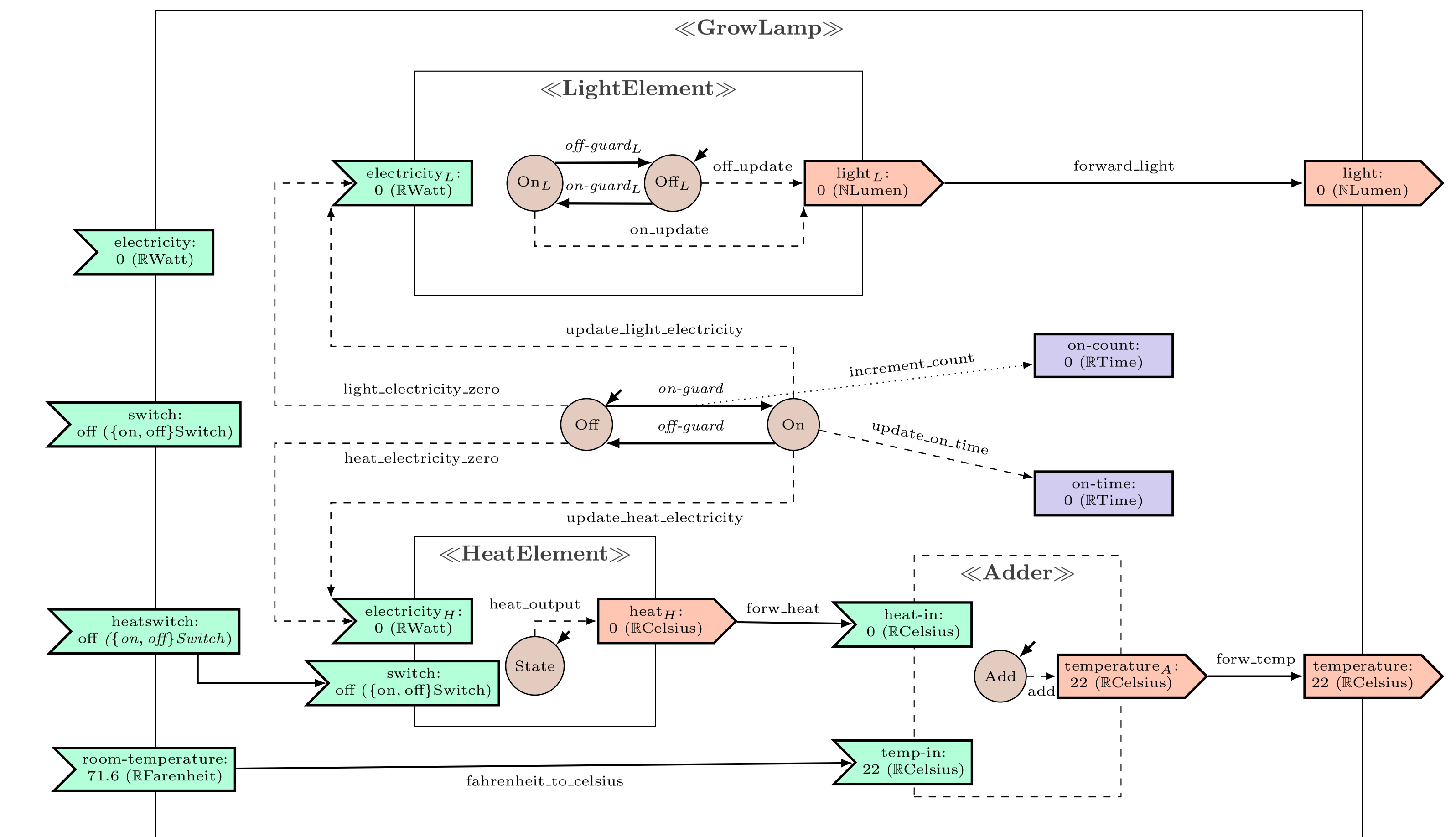


Figure 1: A growing lamp entity with subentities. Note that this diagram annotates updates with the function names, but omits their implementation.

2.1 Formal language structure

CREST's formalisation (structure and semantics) is defined on a system-global level. States, transitions, ports, etc. are then divided into mutually exclusive sets for each entity to preserve locality. The rest of this section refers to non-overlapping partitions of sets, denoted by the $\bigsqcup$ operator.

Notation (Partition of sets)**.** Formally we define a partition as follows: Given a set S, we define the subsets $S_1, \dots S_n$ to be a partition of $S = \bigsqcup_i S_i$ or $S = S_1 \sqcup \dots \sqcup S_n$ iff $\exists S_1, \dots S_n \subseteq S$ such that $\forall i, j, i \neq j \implies S_i \cap S_j = \varnothing$ and $S = \bigcup_{1 \leq i \leq n} S_i$.*

Note that we will not provide a complete description of the concepts of Figure 1 but pick a few representative examples instead. By convention we use the following notations: Sets are *Capitalised*, functions are *lowercased*, and sets of function names are $\mathcal{C}$alligraphed. This section defines the sets and functions that together specify a CREST system.

Definition 1 (Types and Values)**.** Given a set of units *Units* and a set of domains *Domains*, the set of resource types is defined as $Types = Domains \times Units$. The values of a resource type *type* are $\{\langle v, unit\rangle \mid v \in domain, \langle domain, unit\rangle \in Types\}$, where $type = \langle domain, unit\rangle$. The set of all resource values is defined as $Resources = \{\langle v, unit\rangle \mid \exists\langle domain, unit\rangle \in Types \wedge v \in domain\}$. It contains all possible couples of values and units.

For legibility, we use the simplified notations *domain unit* and *v unit* for a resource type and value. We therefore write e.g. $\mathbb{N}$Watt and 3Watt for $\langle \mathbb{N}, \text{Watt}\rangle$ and $\langle 3, \text{Watt}\rangle$.

Finally, we define the $\in$ operator on resource values and resource types, in order to test for compatibility between a value and a type. This feature is used in CREST to verify that a value can be written to a port, which is annotated with a resource type. Formally we define that a resource value is part of a resource type, iff the value is an element of the type's domain, and the value's unit and the type's unit match:

$$\forall res = \langle value, unit_1\rangle \in Resources, \forall type = \langle domain, unit_2\rangle \in Types,$$
$$res \in type \Leftrightarrow value \in domain \wedge unit_1 = unit_2$$

We therefore conclude that e.g. 3Watt $\in \mathbb{N}$Watt and 3Watt $\in \mathbb{R}$Watt, but 3Watt $\notin$ $\mathbb{N}$Lumen. In Figure 1 we see the following resource type definitions:

$$\begin{aligned} Units &= \{\texttt{Watt}, \texttt{Switch}, \texttt{Celsius}, \dots\} \\ Domains &= \{\mathbb{R}, \mathbb{N}, \{\texttt{on}, \texttt{off}\}, \dots\} \\ Types &= \{\mathbb{R}\texttt{Watt}, \{\texttt{on}, \texttt{off}\}\texttt{Switch}, \mathbb{R}\texttt{Celsius}, \dots\} \\ Resources &= \{0\texttt{Watt}, \texttt{onSwitch}, 22\texttt{Celsius}, \dots\} \end{aligned}$$

Definition 2 (Hierarchy of Entities)**.** CREST components are modelled as *entities*. Each entity can contain other entities, which are referred to as *children* or *subentities*.

An entire CREST system's structure forms a rooted tree. It is defined by a set of entity names *Entities*, and a function $parent : Entities \to Entities \cup \{\bot\}$, which returns the parent of an entity or $\bot$ if it has no parent. CREST's strict entity structure asserts that there is exactly one entity without parent, which we refer to as the system's *root.* Formally the constraint is expressed as: $\big|\{e \mid e \in Entities \wedge parent(e) = \bot\}\big| = 1$

$children : Entities \to \mathcal{P}(Entities)$ returns the direct children of any entity, and the constant $root : Entities$ provides the system's root entity.

$$\begin{aligned} children(e) &= \{e' \mid e' \in Entities \wedge parent(e') = e\} \\ root &= e \text{ s.t. } e \in Entities \wedge parent(e) = \bot \end{aligned}$$

The growing lamp example consists of one root entity that encapsulates three subentities for lighting, heating and an adder, which sums the values of its two input ports:

$$\begin{aligned} Entities &= \{\text{GrowLamp}, \text{LightElement}, \dots\} \\ root &= \text{GrowLamp} \\ children(\text{GrowLamp}) &= \{\text{LightElement}, \text{HeatElement}, \text{Adder}\} \\ children(\text{LightElement}) &= \varnothing \\ \dots &= \dots \end{aligned}$$

Definition 3 (Ports)**.** CREST uses *ports* for transfer and storage of resources. A CREST system defines these ports as set of port names *Ports*, and a function $type : Ports \to Types$ that defines the resource type of each port. In the example above, the port names of the growing lamp system are:

$$Ports = \{\text{electricity}, \text{switch}, \text{on-time}, \text{light}, \text{temperature}, \dots\}$$

The types associated with these ports are for instance:

$$type(\text{electricity}) = \mathbb{R}\texttt{Watt} \qquad type(\text{switch}) = \{\text{on}, \text{off}\}\texttt{Switch} \qquad type(\text{light}) = \mathbb{N}\texttt{Lumen}$$

The system's port names are partitioned into *inputs*, *outputs* and *local* ports $Ports = Ports^I \sqcup Ports^L \sqcup Ports^O$. Each port is also assigned to exactly one entity such that

$$Ports = \bigsqcup_{e \in Entities} Ports_e$$

The intersection of these partitions provides us with each entity's inputs, outputs and local ports:

$$\forall e \in Entities \begin{cases} Ports_e^I & = Ports^I \cap Ports_e \\ Ports_e^O & = Ports^O \cap Ports_e \\ Ports_e^L & = Ports^L \cap Ports_e \end{cases}$$

CREST allows only a subset of ports to be used within transition guards and update functions. In fact, an entity can only read certain ports called *sources*. Further, an entity's updates can only write to specific ports called *targets*. These rules enforce the locality principle, as explained above.

The function $sources : Entities \rightarrow \mathcal{P}(Ports)$ provides the ports of an entity, that can be used to calculate transition guards or the value of update functions. The set is composed of an entity's inputs, the entity's local ports and, to pass data from subentities, the direct subentities' output ports.

$$\forall e \in Entities, sources(e) = Ports_e^I \;\cup\; Ports_e^L \;\cup \bigcup_{e' \in children(e)} Ports_{e'}^O$$

In the growing lamp we find for example

$$\begin{aligned} sources(\text{GrowLamp}) &= \{\text{electricity}, \text{switch}, \text{room-temperature}, \text{light}_L, \\ &\qquad \text{on-time}, \text{heat}_H, \text{temperature}_A\} \\ sources(\text{Adder}) &= \{\text{heat-in}, \text{temp-in}\} \end{aligned}$$

$targets : Entities \rightarrow \mathcal{P}(Ports)$ is a function that returns the set of possible targets of update functions for an entity. Targets can be an entity's local ports, outputs and all direct subentities' input ports.

$$\forall e \in Entities, targets(e) = Ports_e^O \;\cup\; Ports_e^L \;\cup \bigcup_{e' \in children(e)} Ports_{e'}^I$$

Definition 4 (Bindings)**.** During the execution of a CREST system, each port is associated with a value of its respective *type*. The mappings from ports to values are defined by the set $Bindings = \{b : Ports \rightarrow Resources \mid \forall p \in Ports, b(p) \in type(p)\}$. The initial port bindings for some of the ports in Figure 1 are as follows:

$$\begin{aligned} b(\text{electricity}) &= 0\texttt{Watt} & b(\text{switch}) &= \text{off}\texttt{Switch} \\ b(\text{on-time}) &= 0\texttt{Time} & b(\text{light}) &= 0\texttt{Lumen} \\ b(\dots) &= \dots & b(\dots) &= \dots \end{aligned}$$

Definition 5 (States and Transitions)**.** The behaviour of a CREST entity is defined by an automaton consisting of a set of states *States* and a guarded transition relation. The set of all states is partitioned into distinct subsets for each entity, such that

$$States = \bigsqcup_{e \in Entities} States_e \wedge \forall e \in Entities, States_e \neq \varnothing$$

In Figure 1 we find the following states: $States = \{\text{On}, \text{Off}, \text{On}_L, \text{Off}_L, \text{Add}, \text{State}\}$. These states are split up for each individual entity as follows:

$$\begin{aligned} States_{\text{GrowLamp}} &= \{\text{On}, \text{Off}\} & States_{\text{LightElement}} &= \{\text{On}_L, \text{Off}_L\} \\ States_{\text{HeatElement}} &= \{State\} & States_{\text{Adder}} &= \{\text{Add}\} \end{aligned}$$

The *Transitions* relation associates a source state to a target state using a guard function name. CREST requires a transition's source and target states to be part of the same entity. $\mathcal{T}$ is the set of all guard function names. Based thereon, the set of transitions is defined by:

$$Transitions \subseteq \bigcup_{e \in Entities} States_e \times States_e \times \mathcal{T}$$

In Figure 1 we find the following definitions

$$\begin{aligned} Transitions &= \{\langle \text{On}, \text{Off}, \text{off-guard}\rangle, \langle \text{Off}, \text{On}, \text{on-guard}\rangle, \dots\} \\ \mathcal{T} &= \{\text{on-guard}, \text{off-guard}, \dots\} \end{aligned}$$

The function $\tau : \mathcal{T} \to (Bindings \times Bindings \to \mathbb{B})$ maps the guard function names to guard function implementations. Guard implementations return a Boolean value (`True`/`False`) when called with the current port bindings *bind* and the previous port bindings *pre* ($bind, pre \in Bindings$). The return value states whether a transition is enabled. The guard function must only use the values of ports in the entities' source ports to compute its result. (See the semantic constraints section.)

In the growing lamp example τ(on-guard) points to the following guard function:

$$\tau(\text{on-guard})(binding, pre) \begin{cases} \texttt{False} & \text{if } binding(\text{electricity}) < 100\text{Watt} \\ \texttt{True} & \text{if } binding(\text{electricity}) \geqslant 100\text{Watt} \end{cases}$$

Definition 6 (Updates)**.** Updates are functionality that allows to write values to ports. Each update specifies an automaton state and evaluates continuously while that state is active. Updates specify functions that evaluate which values are written to ports. These functions are called with the current and previous port bindings as parameters. This makes it possible to modify a port value based on another port's value.

In the example above we see that the `Adder` defines an update `add`. `add` reads the values of `Adder`'s two input ports, sums them up and writes them to the output port.

Updates can also be used to model changes over time. This is possible since update functions have access to the amount of time that passed in the state δt. Hence they can be used to model timing aspects.

Formally, the set of updates *Updates* associates states, ports and update function names $\mathcal{U}$:

$$Updates \subseteq \bigcup_{e \in Entities} \big(States_e \times targets(e) \times \mathcal{U}\big)$$

Per target port and state only one update definition is allowed, to avoid conflicts when two updates try to write to the same port:

$$\forall p \in Ports, s \in States, \mid \{\langle s, p, u\rangle \in Updates\} \mid \leq 1$$

The function $\upsilon : \mathcal{U} \to (Bindings \times Bindings \times \mathbb{T} \to Resources)$ maps the update names to their implementations. Applied to port bindings *bind* and the previous port bindings

pre (*bind*, *pre* $\in$ *Bindings*) and a passed time span $\delta t \in \mathbb{T}$ they provide a new value for the specified port.

The execution of the update function (identified by υ) can only change ports which are *targets* of the entity that contains the associated state. Further, only *source* ports are allowed for the calculation of the returned value. See the semantics section for more details. In our example, updates are for instance:

$$\begin{aligned} \mathit{Updates} = \{&\langle \text{On}, \text{electricity}_L, \text{update_light_electricity} \rangle, \\ &\langle \text{On}, \text{on-time}, \text{update_on_time} \rangle, \dots \} \\ \mathcal{U} = \{&\text{update_on_time}, \text{update_light_electricity}, \dots \} \end{aligned}$$

The CREST diagram also displays a special kind of update functions: *influences*. Influences are static updates that connect two ports statically, independent of an entity's automaton state and the time that passed. Since such influences are a purely syntactic addition (they can be expressed through a set of "normal" updates), they are not directly part of the structure and semantics of a CREST system. See Section 4.1 for the formal definition of influences.

Definition 7 (*dependencies*)**.** We define a function *dependencies* : $\mathcal{U} \to \mathit{Ports}$ that returns a set of ports for each update function name. We add a constraint that dependencies can only be source-ports:

$$\forall \langle s, p, u \rangle \in \mathit{Updates}, s \in \mathit{States}_e, p \in \mathit{targets}(e), \mathit{dependencies}(u) \subseteq \mathit{sources}(e)$$

The dependencies-function is used to determine the execution order of updates within the operational semantics.

2.2 State of the System

Definition 8 (State of the system)**.** The state of an entire CREST system $w \in W$ is a combination of the current states of all entity automata, the port bindings, the ports' previous bindings, and a global time.

$$W = \mathit{Currents} \times \mathit{Bindings} \times \mathit{Bindings} \times \mathbb{T}$$

Each CREST system defines its initial state $w_0 \in W$.

The set of current states is given by $\mathit{Currents} = \{f : \mathit{Entities} \to \mathit{States} \mid \forall e \in \mathit{Entities}, f(e) \in \mathit{States}_e\}$

In the example *current* $\in$ *Currents* is initially defined as

$$\begin{aligned} \mathit{current}(\text{GrowLamp}) &= \text{Off} & \mathit{current}(\text{LightElement}) &= \text{Off}_L \\ \mathit{current}(\text{HeatElement}) &= \varnothing & \mathit{current}(\text{Adder}) &= \text{Add}_L \end{aligned}$$

2.3 CREST Syntactic Structure

Based on the definitions above, a CREST system is specified as a structure containing information about the data types, entity hierarchy, ports, states and transitions, updates, influences, and an initial state:

$$\begin{aligned}\langle \mathit{Units}, \mathit{Domains}, \quad & \mathit{Entities}, \mathit{parent}, \quad \mathit{Ports}, \mathit{type}, \quad \mathit{States}, \\ \mathit{Transitions}, \mathcal{T}, \tau, \quad & \mathit{Updates}, \mathcal{U}, \upsilon, \quad \mathit{dependencies}, \quad w_0\rangle\end{aligned}$$

2.4 Changes to the system state

Definition 9 (Change of automaton states)**.** The state transition of an entity e to a state s is represented by $w[e \mapsto s]$. This creates a new system state w' such that *current* of all entities remains the same, except for e (the entity to be updated), which now maps to s.

$$\begin{aligned}&\forall w \in W, w = \langle \mathit{current}, \mathit{bind}, \mathit{pre}, t\rangle, \\ &\forall e \in \mathit{Entities}, \forall s \in \mathit{States}_e, w[e \mapsto s] = \langle \mathit{current}', \mathit{bind}, \mathit{pre}, t\rangle \\ &\text{where } \forall e' \in \mathit{Entities}, \mathit{current}'(e') = \begin{cases} s & \text{if } e' = e \\ \mathit{current}(e') & \text{otherwise}\end{cases}\end{aligned}$$

Definition 10 (Change of port values)**.** Changes to port bindings are denoted by $w[ps]$, where ps is a set of port-value mappings $(p \mapsto r)$. We define the value assignment to be the creation of the global state where the bindings for all ports p appearing within ps are the new values and all ports not specified within ps remain unchanged.

$$\begin{aligned}&\forall w \in W, w = \langle \mathit{current}, \mathit{bind}, \mathit{pre}, t\rangle, \\ &\forall ps \in \{f : P' \to \mathit{Resources} \mid P' \subseteq \mathit{Ports} \wedge f(p) \in \mathit{resource}(p)\}, \\ &w[ps] = \langle \mathit{current}, \mathit{bind}', \mathit{pre}', t\rangle, \text{ where} \\ &\forall p \in \mathit{Ports}, \begin{cases} \mathit{bind}'(p) = r \wedge \mathit{pre}'(p) = \mathit{bind}(p) & \text{if } p \mapsto r \in ps \\ \mathit{bind}'(p) = \mathit{bind}(p) \wedge \mathit{pre}'(p) = \mathit{pre}(p) & \text{otherwise}\end{cases}\end{aligned}$$

Note, that the the previous port values *pre* of the ports in ps are updated.

To set the GrowLamp's inputs we could call e.g.

$$w[\{\text{electricity} \mapsto 500\mathtt{Watt}, \text{switch} \mapsto \text{on}\mathtt{Switch}\}]$$

These definitions allow us to model the modification of individual automata and port values. The effects of such changes on an entire CREST system and the upkeep of a well-formed system-state require more complex behavioural routines that are defined as CREST's semantics.

3 Semantic Constraints

CREST systems have certain constraints on update functions and transition guards. These constraints limit the implementations of update functions and transition guards. As CREST does neither prescribe a syntax nor a semantics of these implementations it is of absolute importance that these constraints are upheld. Below, we present the constraints that these implementations need to fulfil.

Transition guard locality This constraint states that the guard conditions can only use ports that are part of an entity e's source ports ($sources(e)$) for evaluation. The formal requirement below expresses this using the requirement that the application of a guard function onto two bindings b_1 and b_2 produces the same result, when b_1 and b_2 are equal for all *sources* ports, but differ in (at least) one non-*sources* port.

$$\forall e \in Entities, \forall \langle s,t,g \rangle \in Transitions, s,t \in States_e, \forall b_1, b_2, pre_1, pre_2 \in Bindings,$$
$$\begin{pmatrix} \forall p_1 \in sources(e), b_1(p_1) = b_2(p_1), pre_1(p_1) = pre_2(p_1) \\ \wedge \\ \exists p_2 \notin sources(e), b_1(p_2) \neq b_2(p_2) \vee pre_1(p_2) \neq pre_2(p_2) \end{pmatrix} \implies \tau(g)(b_1, pre_1) = \tau(g)(b_2, pre_2)$$

Update resource type The next constraint states that an update always has to produce it's target's resource.

$$\forall \langle s,p,u \rangle \in Updates, \forall \delta t, \forall b \in Bindings, \upsilon(u)(b, \delta t) \in resource(p)$$

Update function locality Similarly to transition guards, update functions also can only use ports within an entity e's source ports ($sources(e)$)

The constraint below expresses the following: Given any entity e, for all updates writing towards a *target*-port p of that entity, the update's function implementation $\upsilon(u)$ has to produce the same result when applied onto a binding that is equal in it's *source* states.

$$\forall e \in Entities, \forall \langle s,p,u \rangle \in Updates, p \in targets(e), \forall b_1, b_2, pre_1, pre_2 \in Bindings,$$
$$\begin{pmatrix} \forall p_1 \in sources(e), b_1(p_1) = b_2(p_1), pre_1(p_1) = pre_2(p_1) \\ \wedge \\ \exists p_2 \notin sources(e), b_1(p_2) \neq b_2(p_2) \vee pre_1(p_2) \neq pre_2(p_2) \end{pmatrix} \implies \upsilon(u)(b_1, pre_1, \delta t) = \upsilon(u)(b_2, pre_2, \delta t)$$

4 Syntactic Extensions

The basic structure of a CREST system enables to express a large number of constructs. However, in some situations some modelling patterns re-occur many times. One example

is the static linking of two ports with an update function. This update function is continuously triggered in every automaton state and represents constant, state- and time-independent behaviour, such as the conversion of values. Another pattern that one repeatedly comes across is the need to execute an update when a transition is triggered. Both these patterns are easy to express in CREST, however they lead to unnecessary repetition.

In this section two additional syntactic concepts are introduced: *influences* and *transitions with actions*. Both of these concepts translate naturally into the CREST basic system structure. Hence, these additions are purely syntactic and do not add further new modelling power and expressivity. For each of these concepts, the formal definition (translation) is provided and all constraints are provided.

4.1 Influences

As previously mentioned, CREST uses the notion of influences are used to statically link the values of two ports. In the GrowLamp example, `fahrenheit_to_celsius` is an example for such an influence. This influence continuously reads the room-temperature port, executes a transformation and sets its target ports value to the calculated result, independent of the current automaton state.

Hence, an influence is the equivalent of a set of updates (which use the same update function) that are defined for each of the entity's states.

The benefit of influences are that CREST diagrams are more legible and an overload of update functions is avoided.

Syntax Formally, an entity's influences are defined as follows:

$$\forall e \in \mathit{Entities},\ \ \mathit{Influences}_e \subseteq \mathit{sources}(e) \times \mathit{targets}(e) \times \mathcal{U}$$

All of a CREST system's influences are defined as the distinct union of the entities' influences.

$$\mathit{Influences} = \bigsqcup_{e \in \mathit{Entities}} \mathit{Influences}_e$$

The above definition of influences implies that for each influence there is an update related to each state of the influence's entity. Formally, the constraint is expressed by the following implication:

$$\forall e \in \mathit{Entities}, \langle p_1, p_2, u \rangle \in \mathit{Influences}_e \implies \forall s \in \mathit{States}_e, \exists \langle s, p_2, u \rangle \in \mathit{Updates}$$

For the calculation of the modifier precedence, we have to provide the function's dependencies. An influence's dependency is only the source-port.

$$\forall \langle p_1, p_2, u \rangle \in \mathit{Influences} \implies \mathit{dependencies}(u) = \{p_1\}$$

Semantic constraint On the semantic side we require that an influence function's output is only influenced by the source-port's value. This is achieved by the following requirement. We assume that for any two bindings and two *pre*-bindings whose values are, respectively, equal for the influence's source port. The returned value of the influence's function needs to be the same, even if there exists a port where binding or *pre*-binding values differ.

$$\forall e \in \mathit{Entities}, \forall \langle p_1, p_2, u \rangle \in \mathit{Influences}, \forall b_1, b_2, \mathit{pre}_1, \mathit{pre}_2 \in \mathit{Bindings},$$

$$\begin{pmatrix} b_1(p_1) = b_2(p_1) \wedge \mathit{pre}_1(p_1) = \mathit{pre}_2(p_1) \\ \wedge \\ \exists p \in \mathit{Ports}, p \neq p_1, b_1(p) \neq b_2(p) \vee \mathit{pre}_1(p) \neq \mathit{pre}_2(p) \end{pmatrix} \implies \upsilon(u)(b_1, \mathit{pre}_1, \delta t) = \upsilon(u)(b_2, \mathit{pre}_2, \delta t)$$

4.2 Transition actions

In some situations it is convenient to execute an update only once when a transition is triggered. The concepts introduced before suffice to express such behaviour. An example for such a situation is a counter that increments a value every time a transition is fired as displayed in Figure 2a. Instead of passing directly from `Off` to `On`, an intermediate state `Count` is added. After transitioning to this intermediate state, the `plus_one` update performs the value increment. The `Count`-state is however left immediately upon completion of the updates, since the next transition is guarded by guard that always evaluates to `True` and hence is triggered immediately. The semantics in the next section will elaborate on CREST's eager transition firing concept and the exact reasons why the update function is triggered.

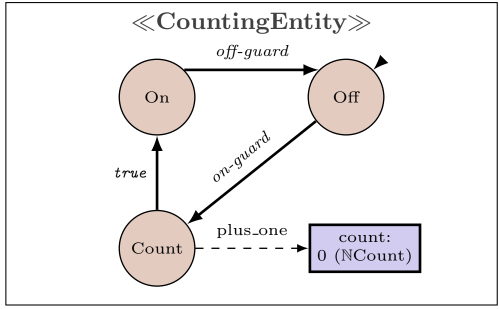
Here we address the syntactical problem of having to define an intermediate state for each such action-update. In order to alleviate this burden, the CREST syntax is extended to include transitions with actions. The graphical syntax of such an action-transition is shown in Figure 2b. Actions are defined as updates connected to a transition, rather than a state.

Structurally, each action-transition defines a guarded transition extended by a set of $\langle$*target-port*, *update function name*$\rangle$ tuples:

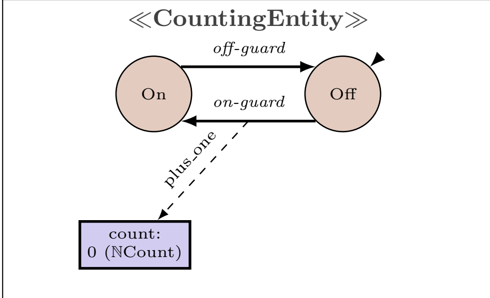
$$\mathit{Transitions}_{\mathit{Act}} \subseteq \bigcup_{e \in \mathit{Entities}} \big(\mathit{States}_e \times \mathit{States}_e \times \mathcal{T} \times \mathcal{P}(\mathit{targets}(e) \times \mathcal{U})\big)$$

As explained above, the introduction of actions is *syntactic sugar*, added to the language to increase usability. This means that each transition with action can be also expressed through an extra state, two transitions and a set of updates (one update per action).

This translation is formalised by the following implication: For each action-transition quadruple that was defined, there exists a an extra state t', a transition from the source-state s to t' that is guarded by the transition guard and another transition from t' to the target-state t that is guarded by *true*, a guard function that always returns `True`.



(a) Transition counter without actions



(b) Transition counter with actions

Figure 2: Two equivalent models of a counter entity. The models count the number of times their `Off`-to-`On` transition is triggered. On the left, a classical implementation without transition actions. The model on the right uses transition actions. `plus_one` reads the value in `count` and increments it by 1.

For each target-port – action couple, a new update is introduced that is linked to t' so that it is executed when the state is entered.

$$\forall e \in \mathit{Entities}, \forall s, t \in \mathit{States}_e$$
$$\langle s, t, g, \mathit{pas}\rangle \in \mathit{Transitions}_{Act} \implies \begin{pmatrix} \exists t' \in \mathit{States}_e, \langle s, t', g\rangle, \langle t', t, \mathit{true}\rangle \in \mathit{Transitions} \\ \wedge \\ \forall \langle p, a\rangle \in \mathit{pas}, \langle t', p, a\rangle \in \mathit{Updates} \end{pmatrix}$$

Semantic constraint Note that transition actions are always called with $\delta t = 0$. This is because transitions are instantaneous and hence time can also not pass during the execution of the update. Hence, actions should hot be influenced by the δt value they are called with.

Formally we require that an action always returns the same value, independent from the time step it is called with. We assume that for any two time steps $\delta t_1, \delta t_2, \in \mathbb{T}$ the action's return value does not change.

$$\forall e \in \mathit{Entities}, \forall \langle s, t, g, \mathit{pas}\rangle \in \mathit{Transitions}_{Act}, \forall \langle p, a\rangle \in \mathit{pas},$$
$$\forall \mathit{bind}, \mathit{pre} \in \mathit{Bindings}, \forall \delta t_1, \delta t_2, \in \mathbb{T}, \upsilon(a)(\mathit{bind}, \mathit{pre}, \delta t_1) = \upsilon(a)(\mathit{bind}, \mathit{pre}, \delta t_2)$$

5 Semantics

CREST's purpose is to facilitate the modelling of cyber-physical systems. CPS are often dominated by parallel events such as the concurrent modification of inputs and state changes based on passed time. An example would be the automated turning off of a

growing lamp after a certain time, while also reacting to changes in temperature. It is evident that the propagation of resource influences is instant (or with negligible delay). The flipping of an electrical mains-switch cuts the power supply that leads to a lamp being turned off, with (almost) no noticeable delay. In order to explicitly model time-spanning behaviour, such as the propagation of heat in a large room, modellers can use delaying components. CREST's instantaneousness warrants the use of a *synchronous* modelling language. Despite the concurrency of actions and influences on one another, the system's components are individual units that each have their own behaviour. Interactions between the components only come into effect after the time of assembly. Therefore, the model should reflect this attribute and upkeep the behavioural locality. These properties are also reflected in CREST's core principles: 1. synchronism, 2. locality, 3. concurrency and parallelism, 4. reactivity, 5. precision and 6. non-determinism.

We see the support of these as a main objective of the CREST language and therefore require the semantics to support them.

In general, CREST supports two basic forms of system interaction:

- modifying the root entity's input ports, and
- advancing the global system time.

After each interaction the system has to be stabilised. Stabilisation refers to the process of bringing a system into a state where all influences and updates have been executed, and no transitions are enabled. In the process of stabilising a system, the CREST simulator ensures that the system input modifications are propagated throughout the system, i.e. that affected entities react to these changes. If this process enables any transitions, they will be triggered, due to CREST's *eager* transition firing concept.

5.1 Modifiers and precedence

CREST's operational semantics make use of the precedence operator $\prec$, that expresses an order between ports, updates and child-entities that arises from the updates' dependencies. Further a function *active-modifiers* identifies the set of updates and child-entities that have to be executed to stabilise a CREST system and to recursively propagate time advances throughout an entity. Due to potential interdependencies between update functions, i.e. one update might read a port which is written by another update, it is necessary to execute the stabilisation in a specific order.

This subsection defines functions and operators that are used the semantic rules for this stabilisation process.

Port precedence The $\prec$ operator is used to create a partial order between ports, based on *dependencies* and the input and output port information. We say that for any two ports $p_1, p_2 \in \mathit{Ports}$ $p_1 \prec p_2$ iff one of the following cases applies:

1. there exists an update whose target is p_2 and p_1 is a dependency;
2. there exists an entity, p_1 is an input and p_2 is an output of that entity;

3. there exists a port p' so that $p_1 \prec p' \prec p_2$.

Formally:

$$\forall p_1, p_2 \in Ports, p_1 \prec p_2 \text{ iff } \begin{cases} \exists \langle s, p_2, u \rangle \in Updates, p_1 \in dependencies(u) \\ \exists e \in Entities, p_1 \in Ports_e^I, p_2 \in Ports_e^O \\ \exists p' \in Ports, p_1 \prec p' \wedge p' \prec p_2 \end{cases}$$

active-modifiers When executing an update within a CREST entity, it is important to execute all updates that are linked to the current state and to propagate the update "down-stream" towards the entity's children. The function *active-modifiers* returns these updates and child-entities as a heterogeneous set.

$$\textit{active-modifiers} : W \times Entity \rightarrow \mathcal{P}(Entity \cup Update)$$

$$\begin{aligned} &\textit{active-modifiers}(\langle current, bind, pre, time \rangle, e) = \\ &\qquad \{\langle s, p, u \rangle \in Update \mid s = current(e) \wedge \} \cup children(e) \end{aligned}$$

Note, that instead of considering CREST influences as specific case, *active-modifiers* return the equivalent update from the current state.

In the growing lamp example above, the active modifiers of the `GrowLamp` entity in Figure 1 are the following:

$$\begin{aligned} &\textit{active-modifiers}(w_0, GrowLamp) = \{LightElement, && HeatElement, \\ &\quad \langle Off, electricity_L, light_electricity_zero \rangle, && \langle Off, electricity_H, heat_electricity_zero \rangle, \\ &\quad \langle Off, light, forward_light \rangle, && \langle Off, heat\text{-}in, forw_heat \rangle, \\ &\quad \langle Off, temperature, forw_temp \rangle && \langle Off, temp\text{-}in, fahrenheit_to_celsius \rangle \\ &\quad \} \end{aligned}$$

Modifier precedence The previous function identified the modifiers that need to be executed when advancing the time and propagating updates through the system state. However, it is very likely that a CREST system defines "chained ports", whose values are linked by update functions and subentity behaviour. This means that the modification of one port will trigger the change of another port.

When looking at the GrowLamp example, we can see that the GrowLamp's `light` output depends on the light-element's $\texttt{light}_L$ output. Abstracting over the internal behaviour of LightElement we can say that the LightElement's $\texttt{light}_L$ output depends on the $\texttt{electricity}_L$ input[2], which is set by the `update_light_electricity` update

[2] Due to CREST's locality principle, child-entities are treated as "black boxes". Hence it is assumed that all of a child-entity's outputs depend on all its inputs

that reads the `electricity` input. In order to calculate the correct port values, we need to take these dependencies into account and execute them in the correct order.

For this reason the $\prec$ operator is adapted to also compare modifiers (i.e. updates and child-entities). Given two modifiers $\prec$ establishes if there is an influence between these modifiers, i.e. if one modifier's output/target ports influence linked the other modifiers input. Formally, $\prec$ over modifiers is defined using the following four definitions:

1. one entity precedes another iff (at least) one output of the former precedes one input of the latter:
   $e_1 \prec e_2 \implies \exists p_1 \in Ports^O_{e_1}, \exists p_2 \in Ports^I_{e_2}, p_1 \prec p_2$
2. a entity precedes an update iff one of the entity's outputs precedes one f the update's dependencies:
   $e \prec up \implies \exists p_1 \in Ports^O_e, up = \langle s, p, u \rangle, p_2 \in dependencies(u), p_1 \prec p_2$
3. an update precedes a entity iff the update's target precedes one of the entity's inputs:
   $up \prec e \implies up = \langle s, p_1, u \rangle, \exists p_2 \in Ports^I_e, p_1 \prec p_2$
4. an update precedes another update iff the former's target precedes one of the latter's dependencies:
   $up_1 \prec up_2 \implies up_1 = \langle s, p_1, u_1 \rangle, up_2 = \langle s, p', u_2 \rangle, \exists p_2 \in dependencies(u_2), p_1 \prec p_2$

enabled-transitions A CREST entity can define guarded transitions between its states. In order to evaluate which transitions are enabled, a function *enabled-transitions* is used that, given a current state and an entity returns the set of transitions whose guard functions return `True`.

$$enabled\text{-}transitions : W \times Entity \to \mathcal{P}(Transitions)$$

$$enabled\text{-}transitions(\langle curr, bind, pre, time \rangle, e) = \{\langle s, t, g \rangle \in Transitions \mid s, t \in States_e \wedge s = curr(e) \wedge \tau(g)(bind, pre) \mapsto \mathbf{True}\}$$

5.2 Operational Semantics

CREST's semantics describe modifications to the global system state w. System modifications are executed and the modifications are propagated "downstream" from the *root* entity towards the leafs of the CREST entity hierarchy. The locality principle demands that an entity is responsible for the upkeep of its own state and can only access the interfaces of its subentities. Hence an entity does not know about its hierarchical "ancestors".

CREST's semantics are based on the concept of reaching a fixed point ("fixpoint") after each system modification. Fixpoints are states in which the system is stable, i.e. the system does not change unless time passes or external factors modify the system.

set-values The fixpoint concept is applied for example when modifying the value of ports, as shown in Rule 1, below. We see that setting values (specified using a port-value mapping) requires the execution of an `update-and-stabilise` procedure after the modification of the port values. This routine triggers an entity's updates and automata transitions until a fixpoint is reached. In the process it also recursively propagate the modifications to its child-entities.

$$\frac{w_1 = w[vs], \quad \langle w_1, root, 0\rangle \xrightarrow{\text{update-and-stabilise}} w_2}{\langle w, vs\rangle \xrightarrow{\text{set-values}} w_2} \tag{1}$$

update-and-stabilise The propagation of system changes (i.e. port values or time advance) throughout a CREST system is based on the concept that after a change, all updates of an entity should be executed and then trigger updates and stabilisation in the child-entities. Rule 2 is called on a specific entity e whose updates are to be triggered and a size of time step δt that should be executed. For stabilisation of the system without time advance, such as after the external setting of port values (see Rule 1 above), this rule can be called with $\delta t = 0$, which will propagate values but not trigger an advance of time with in the system.

After the update propagation phase has finished, CREST looks triggers $\texttt{transition}_{\texttt{fp}}$ to advance the automaton state until no further transitions are enabled.

Specifically, the rule first gathers all enabled modifiers (updates from current state and child-entities) and triggers their execution in the `update-all` rule, followed by the `stabilise` rule.

$$\frac{mods = \textit{active-modifiers}(w, e), \quad \Big\langle \textit{set-pre}(w, e), e, mods, \delta t\Big\rangle \xrightarrow{\text{apply-all}} w_1 \quad \langle w_1, e\rangle \xrightarrow{\text{stabilise}} w_2}{\langle w, e, \delta t\rangle \xrightarrow{\text{update-and-stabilise}} w_2} \tag{2}$$

The above rule uses a function *set-pre* that, given a state $w = \langle curr, bind, pre, time\rangle$ and an entity e creates a new state, where *pre* is modified. The return value's *pre* is set to *bind* for all ports in $targets(e)$. This function is used before executing any modifiers of this entity to assert that the modifiers have access to the port's previous values (i.e. the values before the updates). and before t values or advancing time, so that the update functions that are defined in the system can access the value before the update (and hence operate incrementally). Note, that *set-pre* only modifies the *pre*-bindings of an entity's target ports, in order to maintain consistency with the rest of the formalisation. This means that an entity is responsible for setting the *pre* values of its subentities' inputs.

$$\textit{set-pre}\big(\langle curr, bind, pre, time\rangle, e\big) = \langle curr, bind, pre', time\rangle$$
$$\text{where } pre'(p) = \begin{cases} bind(p) & \text{if } p \in targets(e) \\ pre(p) & \text{otherwise} \end{cases}$$

apply-all This rule is responsible for identifying one modifier within a set of modifiers *mods* that does not have any preceding modifiers and executing it. Formally, it employs the $\prec$ operator to evaluate this condition. After such a modifier is found, it is executed (see `apply-one` below) before recursively calling `apply-all` on the rest of the set. Using this approach, CREST propagates value changes iteratively, taking dependencies between ports into account.

Rule 4 is the break-condition of the recursion. It is called when the list of modifiers is the empty-set $\varnothing$, i.e. all applicable modifiers have been executed and removed. In this case, no action is taken and the system state remains unchanged.

$$\frac{\nexists m' \in (mods \setminus m_1), m' \prec m_1, \quad \langle w, m_1, \delta t\rangle \xrightarrow{\text{apply-one}} w_1, \quad \langle w_1, mods \setminus m_1, \delta t\rangle \xrightarrow{\text{apply-all}} w_2}{\langle w, mods, \delta t\rangle \xrightarrow{\text{apply-all}} w_2} \quad (3)$$

$$\frac{}{\langle w, \varnothing, \delta t\rangle \xrightarrow{\text{apply-all}} w} \quad (4)$$

apply-one The two `apply-one` functions (Rule 5, Rule 6) execute the modifier (update or child-entity) based on their type. If the modifier is an update, the execution of the update is relayed to the specific `update` function. Otherwise (if it is a child-entity), it calls `update-and-stabilise` on the child-entity to propagate the changed system state and triggering updates within the children.

$$\frac{mod \in \mathit{Update}, \quad \langle w, mod, \delta t\rangle \xrightarrow{\text{update}} w_1}{\langle w, mod, \delta t\rangle \xrightarrow{\text{apply-one}} w_1} \quad (5)$$

$$\frac{mod \in \mathit{Entities}, \quad \langle w, mod, \delta t\rangle \xrightarrow{\text{update-and-stabilise}} w_1}{\langle w, mod, \delta t\rangle \xrightarrow{\text{apply-one}} w_1} \quad (6)$$

It is important to understand that the similar treatment of updates and child-entities is a significant feature of CREST. CREST sees child-entities as a complex form of update which reads input values and writes output values. This *black-box* concept encapsulates all child behaviour and allows every CREST entity to always rely on the fact that its children are in a stable state[3].

update The execution of an update is relatively simple. The premises of Rule 7 extract the update function's name u, current port bindings *bind* and previous port bindings *pre*. The new system state is the old state where the update's target port p is set to the value returned by the update function implementation $\upsilon(u)$ executed with the parameters *bind*, *pre* and δt.

$$\frac{\langle s, p, u\rangle = mod, \quad w = \langle curr, bind, pre, time\rangle,}{\langle w, mod, \delta t\rangle \xrightarrow{\text{update}} w[p \mapsto \upsilon(u)(bind, pre, \delta t)]} \quad (7)$$

Update functions have access to δt, the time that has passed since the current state was activated. This allows the modification of port values over time and thereby potentially enabling transitions. Figure Figure 3 shows such an enabling over time. It displays a water tank system that contains a pump. When the pump is turned on, the volume of water is calculated by an update function which continuously evaluates as follows:

[3]Child-entities can become un-stabilisable due to a e.g. cycle of transitions that are continuously enabled

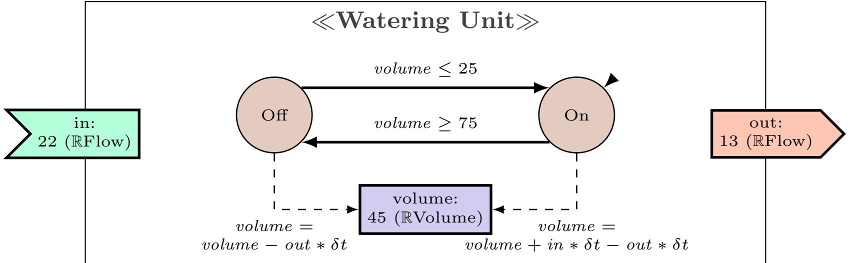


Figure 3: A water tank

$volume = volume + in * \delta t - out * \delta t$, where *in* is the amount of water being pumped into the tank per time unit and *out* the water leaving the tank. The pump transitions to `Off` when the water volume exceeds 75 (guard *volume_above_75*). In `Off` the update subtracts $out * \delta t$ from the *volume*. The pump starts as soon as the volume drops below 25. We see that – assuming reasonable *in* and *out* flows – over time the pump will change regularly between the `On` and `Off` state. Note, that for simplicity reasons in this example we chose to annotate the updates and transitions with their implementations, rather than their names. While this "shortcut" allows for more expressive diagrams, it is important to understand that these annotations are mathematical syntax, and not part of CREST.

stabilise The `stabilise` rules are responsible for triggering transitions. In case a transition was executed (i.e. the global system state w changed, Rule 8), it executes `update-and-stabilise` to trigger the update functions which are connected to the new automaton state.

If no transition was executed, no further action is taken (Rule 9).

$$\frac{\langle w, e\rangle \xrightarrow{\text{transitions}} w_1, \quad w \neq w_1, \quad \langle w_1, e, 0\rangle \xrightarrow{\text{update-and-stabilise}} w_2}{\langle w, e, \delta t\rangle \xrightarrow{\text{stabilise}} w_2} \quad (8) \qquad \frac{\langle w, e\rangle \xrightarrow{\text{transitions}} w}{\langle w, e, \delta t\rangle \xrightarrow{\text{stabilise}} w} \quad (9)$$

CREST implements eager transition evaluation. This means that a transition must be fired if (at least) one is enabled. It is essential that update functions are triggered immediately after the transition phase, as they otherwise risk to be executed at the wrong moments or not at all. A simple example visualising this behaviour in Figure 2a visualises this by showing an on/off automaton with a transition counter. By adding an intermediate state `Count` between `Off` and `On`, it is possible to count the number of times the automaton switched from off to on. The function `plus_one` is evaluated immediately after entering `Count`. On the next iteration of `stabilise` (which is called by `update-and-stabilise`) the automaton switches to state *On* because of the transition guard that always evaluates to `True`.

If CREST were to first execute all enabled transitions and only then trigger the updates when the automaton cannot not advance anymore, `plus_one` would never be executed and it would be impossible to implement a counter such as this one.

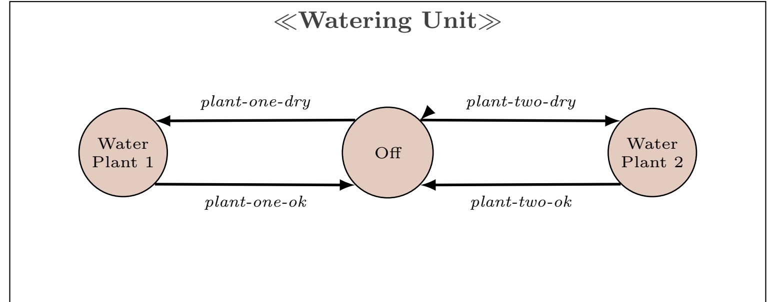


Figure 4: A non-deterministic state automaton

This particularity is the basis for CREST's syntactic extension for transitions with actions.

transitions The execution of transitions itself is based on two rules. First, Rule 10 describes the case where no transitions are enabled. In this case there are no effects to the system state. Second, in case there are enabled transitions, Rule 11 selects one of them and changes the automaton's state.

$$\frac{\textit{enabled-transitions}(w, e) = \varnothing}{\langle w, e \rangle \xrightarrow{\text{transition}} w} \quad (10) \qquad \frac{\langle s, t, g \rangle \in \textit{enabled-transitions}(w, e)}{\langle w, e \rangle \xrightarrow{\text{transitions}} w[e \mapsto t]} \quad (11)$$

It is noteworthy that CREST does not prescribe a strategy in the event where more than one transition is enabled. Thus this is the place where non-determinism is possible, e.g. if guard conditions "overlap".

Figure Figure 4 displays the state automaton of a non-deterministic entity. We assume our system to be a watering unit which is connected to two plants. If a plant's soil is dry, the watering unit automatically waters that plant (`Water Plant X`). We can easily imagine a scenario where both plants are dry at the same time (e.g. just after starting the system). Since CREST does not dictate any strategy, the decision which plant to water first is non-deterministic. This non-determinism is an important property to model many software systems.

Time advance We see from the above rules that updates allow the modification of a system over time. The semantics of these time advances are defined below. The decision of which rule to apply depends on the amount of time to advance δt. CREST only supports positive δt-values, meaning that it is not possible to "step back in time". In the semantics, such actions have no effect. Further, an advance of $\delta t = 0$ does not affect the system state either, as shown in Rule 12.

$$\frac{\delta t \leqslant 0}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} w} \quad (12)$$

The advance of time relies on the availability of $\textit{next_transition_time}(w)$. Given a CREST system's current state w this function tries to calculate the precise amount of

time δt that has to pass until updates enable any transition's guard condition. The calculated value is in the range $[0, \ldots, \infty)$, where 0 states that a transition is already enabled (and implying that the system is not in a stable state) or ∞, meaning that no transition can be enabled by just advancing time. CREST's implementation uses for example an SMT solver that transpiles the update functions' source codes into constraints that can be solved by a minimum next transition time, although other strategies are possible. A detailed discussion about the strategies of $next_transition_time(w)$ exceeds however the scope of this report.

CREST distinguishes between two cases of time advances:

1. If the time we plan to advance δt is less than or equal to the next transition time, Rule 13 applies. Since no transition can become enabled before a time step of δt, CREST can safely advance. Before the advance, the rule sets the *pre* values of the current binding, so they are available for the updates.

   Subsequently `update-and-stabilise` is called on the *root* entity with parameter δt, in order to trigger all system updates and stabilisation of entities (including transition triggering). Lastly (in the rule's conclusion) the system's global time is updated.

$$\frac{\delta t \leqslant next_transition_time(w), \quad \langle w, root, \delta t \rangle \xrightarrow{\text{update-and-stabilise}} \langle curr, bind, pre, time \rangle}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} \langle curr, bind, pretime + \delta t \rangle} \tag{13}$$

2. If the next transition time ntt is less than δt, Rule 14 will split the advance into two steps: First, it will trigger *advance* with the value ntt, which activates Rule 13 and a stabilisation (including transition firing). Next, CREST recursively advances the remaining time $(\delta t - ntt)$.

$$\frac{ntt = next_transition_time(w), \quad \delta t > ntt, \quad \langle w, ntt \rangle \xrightarrow{\text{advance}} w_1, \quad \langle w_1, \delta t - ntt \rangle \xrightarrow{\text{advance}} w_2}{\langle w, \delta t \rangle \xrightarrow{\text{advance}} w_2} \tag{14}$$

CREST's time semantics allow for the simulation and verification based on real-valued clocks with arbitrarily small time advances. This feature is essential for the precise simulation of cyber-physical systems without the need for an artificial base-clock.

6 Conclusion and Future Work

This report presents the formal aspects of CREST, a novel, domain-specific language for the modelling of cyber-physical systems (CPS). CREST is designed for small, custom CPS such as home and office automation applications or automated gardening. The language's syntax and semantics were developed to support the six key aspects that are required by such applications: synchronism, locality, concurrency and parallelism, reactivity, continuous behaviour, and non-determinism. In order to support these properties, the syntax is based on continuous data transfer between the ports of a hierarchical

entity system. Each entity additionally specifies its behaviour using a finite-state automaton and can hence modify and adapt its behaviour based on port value changes. CREST features update functions which allow a modification of port values over time and hence model continuous evolutions. The operational semantics are formally defined and revolve around concept of stabilisation after each system modification. This stabilisation is achieved by fixpoint searches.

The formalisation is extensively used by the CREST implementation[4] for the simulation and verification of CPS. In future, we plan to extend CREST to additionally support other forms of behaviour specification, such as Petri nets. Petri nets have been shown to be highly usable for concurrent systems and allow the modelling of non-deterministic systems. A next version of CREST will support this specification by linking the execution of value updates to the marking within a Petri net place, rather than an automaton state. We additionally plan to translate CREST systems to other formalisms that have extensive verification and validation tool support. Formalisms such as hybrid systems and DEVS seem to be suitable transpilation targets.

[4] `https://github.com/stklik/CREST`