CREST - A DSL for Reactive Cyber-Physical Systems

Stefan Klikovits, Alban Linard, and Didier Buchs *

Software Modeling and Verification (SMV) Group, *Faculty of Science*
University of Geneva Geneva, Switzerland
firstname.lastname@unige.ch

Abstract. This article presents CREST, a novel domain-specific language for the modelling of cyber-physical systems. CREST is designed for the simple and clear modelling, simulation and verification of small-scale systems such as home and office automation, smart gardening systems and similar. The language is designed to model the flow of resources throughout the system. It features synchronous system evolution and reactive behaviour. CREST's formal semantics allow real-valued time advances and the modelling of timed system evolution. The continuous time concept permits the precise simulation of future system behaviour by automatically calculating next transition times. We present CREST in a practical manner, and elaborate on the Python-based DSL implementation and simulator.

1 Introduction

Cyber-physical systems (CPS) are combinations of software components, that perform computation, and hardware interfaces, such as sensors and actuators, which connect the system to the physical world. Enabled by inexpensive hardware, applications such as home and building automation, or more generally the Internet-of-Things (IoT), are recent and popular manifestations of CPS which offer the possibility to digitally control large parts of our lives. This recent proliferation requires more trust in CPS' correctness.

Classical CPS domains such as aviation and transport, heavy industry as well as large-scale and complex systems using dedicated formalisms, languages and tools to control, simulate and verify their systems. While these solutions are commonly used by financially potent institutions, creators of small and custom systems often lack the knowledge and resources to use such tools. The goal of our project is to give these people the means to easily model and check their CPS.

In this article we present the Continuous REactive SysTems (CREST) language. CREST is a domain-specific language (DSL) created for the modelling

* 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

of small-scale CPS such as home, office and building automation, automated gardening systems and similar. The language particularly emphasises the simple representation of three CPS aspects: 1. the continuous flow of physical resources (e.g. light, heat, electricity) and data signals (e.g. on/off switches, control commands) within a system; 2. the state-based behaviour of CPS components; and 3. the evolution of a system over time.

CREST's strictly hierarchical system view encourages composition and system-of-systems designs. The formal language semantics guarantee a synchronous representation and evolution of the model, while still preserving dynamic behaviour. It features arbitrary time granularity, as opposed to fixed time steps, and hence avoids the need for *ticks* commonly used in other languages. CREST is implemented as an *internal DSL* in Python[1], which means that it uses Python's execution environment and language as a foundation.

The rest of the paper is structured as follows: Section 2 provides related work and the reasoning behind the choice of designing a new DSL, instead of using existing solutions. Section 3 introduces the CREST language, its graphical syntax and semantics. Section 4 outlines the CREST's Python-based implementation, the interactive modelling environment and simulation capabilities. Section 5 concludes and discusses future work.

2 Motivation and Related Works

Over the years, a large number of formalisms, languages and tools have been developed to aid the modelling and verification of systems. Even though each one of them has its own, clear strengths, oftentimes the choice of one is not trivial and requires trade-offs. In order to find the most appropriate candidate for the modelling and simulation of CPS such as the ones described above, we performed a requirements analysis, collecting the properties of the target systems and comparing them to the available solutions.

For this evaluation we assumed three different case studies that should be modelled. The first one, a smart home system includes solar panels, a battery and an standard electricity mains for power supply, a water boiler, shower and various home appliances (e.g. IoT vacuum cleaner, TV, dishwasher). Next, an office system that features automated light and temperature regulation based on presence sensors, environmental sensors and work schedules. The third system is an automated gardening systems that uses a relay to control growing lamps and a water pump to automatically grow plants inside a home. Measurements are performed using light, temperature and soil moisture sensors.

Such systems require a modelling language/tool that is capable of representing the flow of physical influences (e.g. light, water, electricity) between components, additional to expressing the component's state and evolution over time. Next to structural considerations, an analysis of the systems' behaviour was performed. This exploration led to the discovery of six key aspects that should be supported by the chosen language/tool:

[1] `https://www.python.org/`

1. *Locality.* Despite the exchange of data and resources, system components usually have states and data that should remain local. As an example we can think of a lamp. Its state, life-time and power consumption are local attributes, independent of other components. Interaction occurs through a well-defined interface, i.e. the power plug and switch.
2. *Continuous Time.* Most CPS deal in some way with timing aspects. Plants require a certain amount of light per day, water consumption is measured per minute, etc. Ideally, the chosen formalism will allow arbitrary (real-valued) time steps so that all points in time can be analysed (not just the ones that coincide with *ticks*). The time concept has to support continuous influences between components (e.g. a pump filling a water tank).
3. *Synchronism.* While some changes happen over time, most effects are immediate. For example, a room is (virtually) immediately illuminated by a lamp. The actual time delay is negligible for our target applications. Even for energy saving lamps, whose luminosity increases over time, the transition to the on-state and dissipation of light starts immediately. The synchronism concept requires that as soon as a value changes, the entire system is synchronised and checked for possible changed influences between components.
4. *Reactivity.* The goal of CPS is to model components and systems that react to changes in their environment. When the sun sets, a home automation system should adapt and provide another light source.
5. *Parallelism and Concurrency.* While synchronism and reactivity prescribe each individual subsystem's behaviour, CPSs consists of many components acting in parallel. A tripped fuse shut down all electrical appliances at the same time.
6. *Non-determinism.* When it comes to real-world applications the evolution of a system is not always predictable. For example, the communication between wireless components can temporarily fail. It should be possible to model these scenarios.

This list served as a reference guide for the search of a suitable language. Additionally to the above properties we took properties such as simplicity, expressiveness and availability of formal semantics into account. Lastly, we are interested in the usability and suitability for our target domain, i.e. how it allows the expression of the data types and concepts required by our systems, as well as the complexity of the created solutions. The rest of this section presents the tools and languages that were evaluated before choosing to develop CREST.

2.1 Evaluation of Existing Tools and Languages

The modelling of software systems has been dominated by languages such as UML 2 and SysML [19]. Despite their versatility in the software world, their support for physical systems is rather limited. They lack important embedded systems concepts such as real-time behaviour and timing constraints. Extensions, such as the MARTE UML profile [18] aim to provide those missing features, at the cost of added complexity. MARTE for example provides a very complex

web of languages, which makes the modelling of simple systems (e.g. home automation) complicated and time consuming. UML also entails an often-criticised architectural focus, which is necessary for efficient CPS modelling.

Architecture Description Languages (ADLs) such as AADL [9] are designed to overcome this problem by modelling systems using architectural component and connector views. However, in most cases they focus on pure architectural concepts and do not support behavioural concepts. CREST in contrast aims to merge the behavioural and architectural side. Extensions to ADLs have been proposed to overcome this shortcoming. AADL's Behavioural Annex [11] and MontiArcAutomaton (MAA) [22] extend the capabilities of AADL and MontiArc [12], respectively, and allow modelling of CPS using automata. While these extension do add the missing behavioural features, AADL's extension lacks a formal basis and MAA only supports the time-synchronous or cycle-based (*tick*) evolution and lacks support of clocks and similar time concepts. Further, MAA uses MontiArc's asynchronous message passing system and hence contradicts our synchronism requirement.

Hardware Description Languages (HDLs) such as VHDL [1] and Verilog [23] have been successfully used to model System-on-Chip designs and embedded systems from a functionality level down to the Transaction-Level Modeling and Register-Transfer Level. The C++-based and IEEE standardised SystemC [3] language is a valuable addition to the HDL domain. All three languages offer design as modules, events and message passing between ports, and allow for the storage of data. Aptly named, HDLs mostly target low-level systems and provide built-in support for embedded concepts (e.g. mutex, semaphores, four-valued logic) and measure time in sub-second granularity (e.g. picosecond). Most tooling and verification support only focuses on the generation and verification of TLM and RTL level designs, which is too low-level for our purposes. Another caveat is, that the language's semantics are not formally defined.

The Specification and Description Language (SDL) [10] is a strong candidate for the modelling of the systems such as ours. It provides hierarchical composition of entities (called *agents*) and behaviour using extended finite state machines. Its design is reactive, agents can perform their processing upon input signal receipt. Timing constraints can be modelled using *timers* that also trigger a signal upon expiration. SDL's rigorous formal basis is a compelling advantage that allows formal verification (e.g. [24]) and tool-independent simulation. SDL's weak point with respect to our requirements, is that all SDL signals are asynchronous. This goes against our view of CPS, where influences and signals are synchronous.

The family of synchronous languages, such as Lustre [13] or Esterel [2], is commonly employed in the field of reactive systems. A synchronous module waits for input signals acts upon them instantaneously and produces output signals. It is assumed that the reaction (i.e. computation) of a module is infinitely fast and hence no time passes during execution. One caveat however is, classical synchronous systems do no have a notion of time. In order to introduce this concept an external clock has to be defined as signal input. Recently as a Lustre-based extension, Zélus [4] overcomes this limitation by adding support

for ordinary differential equations that model continuous behaviour. However, just as Lustre, Zélus' suffers from a steep learning curve and difficult syntax.

The CPS in our case studies consist in several components with state-based behaviour, where the component behaviour can change as time passes. This definition is close to the hybrid automata (HA) formalism [21]. HA contain a finite state automaton and model continuous evolution via variables that evolve according to ordinary differential equations (ODE). Transitions are executed according to state invariants and transition conditions. The popularity of HA and hybrid systems (HS) resulted in the development of many languages and tools, such as Simulink/Stateflow [7], HyVisual [5], Modelica, etc. Simulink is the de-facto industry standard of CPS modelling. It is possible to hierarchically design nonlinear, dynamic systems, using different time concepts (e.g. nonlinear, discrete, continuous). Stateflow adds a reactive finite state machine concept to Simulink. Neither Simulink nor Stateflow have formal semantics defined, although proposals exist (e.g. [14]). HyVisual is based on Ptolemy II [20] and allows the definition of hybrid systems with causal influences. It has, contrary to Simulink, a formal operational semantics that can be leveraged for simulation. However, HyVisual's only features a graphical syntax that can become complex to interact with.

A thorough study of HS tools and languages is given in [6] where Carloni et al. use two well-known case studies for their evaluation. The authors also compare tools for the verification of HS, which is a complex task in general where many properties are undecidable [15]. The drawback of HS is their complexity and required familiarity with the formalism. HS however, serve as a possible transpilation target of CREST models so they are used for verification and validation.

The knowledge gathered from these evaluations led us to the conclusion that the modelling of small CPS cannot conveniently be done by using the previous formalisms. The analysed languages and tools either target other domains (UML, HDL, ADL) or lack vital concepts (e.g. time in MontiArcAutomaton). The most promising candidates, the hybrid systems applications either lack formal semantics for verification purposes (Simulink, Modelica) or lack usability (e.g. HyVisual's graphical modelling environment, as pointed out in [6]). A subset of evaluation results is compared in Table 1.

3 CREST Language

The decision to develop CREST is based on the recognition that none of the evaluated candidates fills the need of a formal language meeting our requirements. CREST is the result of combining the most useful concepts of other systems languages, adapted to increase simplicity and usability. This section introduces CREST's graphical syntax and outlines its semantics. For spatial reasons we cannot provide the formal definition and semantics, but refer the interested reader to the detailed technical report [17].

We will use the concrete example of a growing lamp to introduce the individual CREST concepts. Our growing lamp is a device that is used for growing

Formalism/Tool	Locality	Cont. Time	Synchronism	Reactivity	Parallelism	Non-determ.	Formal basis	Simplicity	Expressiveness	Usab. & Suitab.
UML (MARTE)	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗
AADL + Beh.Ann	✓	✓	✓	✓	✗	✓	✗	∼	✓	✗
MontiArcAutomaton	✓	✗	✗	✓	✓	✓	✓	∼	✓	✓
SystemC	✓	✗	∼	✓	✗	∼	✗	∼	✓	✗
SDL	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Esterel	✓	✗	✓	✓	✓	✗	✓	∼	∼	∼
Zélus	?	∼	✓	✓	?	?	✓	✗	∼	∼
Simulink/Stateflow	✓	✓	✗	✓	✓	✓	✗	∼	✓	✓
HyVisual	✓	✓	✓	✓	✓	✓	✓	∼	✓	∼
Modelica	✓	✓	∼	✓	✓	✓	✗	✗	✓	∼
CREST	✓	✓	✓	✓	✓	✓	✓	✓	∼	✓

Table 1: Evaluation of a selection of candidates for modelling of small-scale CPS. Symbol meaning: ✓(*Yes*), ✗(*No*), ∼(*to a certain extent*), ?(*not fully known*)

plants indoors. When turned on, it consumes electricity and produces light. There is also a function where the lamp converts electricity into heat. This feature is controlled by an additional switch.

3.1 CREST syntax

CREST's graphical syntax, called *CREST diagram*, was developed to facilitate the legibility of architecture and behaviour within the system. Figure 1 displays the complete CREST diagram of the growing lamp.

In CREST, each component clearly defines its scope. Visually this is represented by a black border, showing the scope's limits. The component's communication interface is drawn on the edge of this scope, while the internal structure and behaviour are placed on the inside.

System Structure CREST enforces the view that all CPS are defined as hierarchical compositions. This concept is by expressed by defining components ("**entities**") in a nested tree-structure. A CREST system contains one, sole *root* entity. This entity can define arbitrarily many subentities, which can also contain children, etc. The growing lamp for example, consists of two separate modules, one for light (`LightElement`), one for heating (`HeatElement`). Both are embedded within the `GrowLamp` entity.

The strict hierarchy concept asserts a simplified, localised view on an entity level. Each entity encapsulates its internal structure and allows us to treat it as a *black box*. This black box view facilitates composition, as the entity's *parent* can treat it as coherent instance, disregarding the inside.

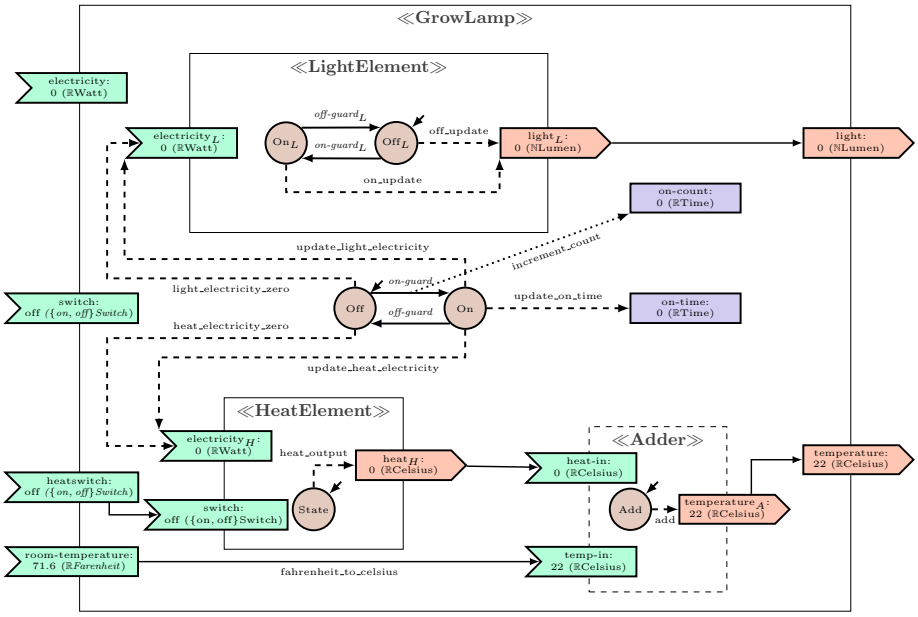

Fig. 1: A growing lamp entity with subentities.

The black box view is completed by the definition of an entity's communication interface, which consists of **input** ports and **output** ports. Ports are required for the modelling of the flow of resources through the system.

CREST specifies a third kind of port: **locals**. This port type is not part of the interface, but rather serves as internal storage of data. In the example we see `on-time` and `on-count` as internal ports. All three types of port are associated with a particular resource.

In CREST, **resources** are value types consisting of a value *domain* and a *unit* (formally $Domain \times Unit$). The growing lamp specifies units such as Watt or Lumen. Domains are sets of values, e.g. the natural numbers $\mathbb{N}$, rationals $\mathbb{R}$ or a set of discrete values such as $\{\text{on}, \text{off}\}$ (for `Switch` in the example). Next to resources, each port specifies a value from its resource as its current value.

Entity Behaviour CREST uses finite state machines (FSMs) to specify behaviour. Each entity defines a set of **states** and guarded **transitions** between them. Transitions relate source states with target states and the names of transition guards (e.g. off —on-guard→ on). The transition guard implementations are functions that take an entity's set of port value bindings *bind* (and previous port bindings *pre*) as parameters and returns a Boolean (`True`, `False`) value indicating whether the transition is enabled or not. Note that CREST does not provide a syntax for the definition of guard functions. Instead, the formal syntax prescribes the function signature and leaves the implementation of a guard language

under-specified for flexibility[2]. Mathematically, the behaviour of `on-guard` could be specified using the following formula:

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

Formally *bind* and *pre* are defined as functions that applied onto a port return the port's value.

Note, that the concept of previous values is required for two reasons: First, it can be used to discover and analyse ports' value changes (i.e. *bind(port)* $\neq$ *pre(port)*). Second, in certain situations it can be used to resolve algebraic loops, which otherwise could not be supported in CREST. The concept of supporting previous values is present as *pre* operator in other languages such as Modelica, Lustre and Esterel. In CREST's implementation *pre* is automatically managed for the user and in certain cases automatically used when necessary.

Resource flow Resource transfers between ports can be modelled using **updates** (`-►`). Updates are defined using a state, a target port and an update function name. If the automaton is in the specified state, the update function (identified by its name) is executed, modelling continuous changes. The function itself returns the target port's new value binding. Self-evidently the returned value has to be in the domain of the target port. Conceptually, updates are continuously executed so that the system's ports always hold the latest values. Practically, CREST's simulator asserts that the evaluations are performed when necessary, as explained below. The growing lamp defines several updates, such as `update_on_time`, `update_light_electricity` (both in GrowLamp) or `heat_output` (in HeatElement). Updates enforce CREST's *synchronism* principle. Provided the automaton is in the update's matching state, the continuous evaluation of update functions guarantees that the target port's value is the result of the update function execution, without delay or explicit message passing.

Similar to transition guards, the update functions' syntax is under-specified but constrained by a required signature. Update functions are executed with the current and previous port bindings *bind* and *pre* and additionally have access to another parameter δt. It is a value of the system's time-base $\mathbb{T}$ and holds the information about the amount of time that has passed since entering the associated FSM state. Hence, update functions can be used to model continuous behaviour and value updates. In the growing lamp's example the time-base is rational (i.e. $\mathbb{T} = \mathbb{R}$). As an example we provide the mathematical definition of `update_on_time`, which continuously accumulates the amount of time the automaton spent in time `on`:

$$\text{update\_on\_time}(\mathit{bind}, \mathit{pre}, \delta t) = \mathit{pre}(\textit{on-time}) + \delta t$$

[2] Section 4 shows how Python is used as a host language for implementing transition guards and other parts of CREST.

In CPS, resources are often continuously transferred from one port to another, independent of entity state or the time that has passed. In the example above, the growing lamp's `heatswitch` port value is transferred to the HeatElement's `switch` input, disregarding whether the lamp is on or off. In order to avoid the specification of the same update function for every state in the system, CREST offers **influences** ($\longrightarrow$) as a syntactic shortcut. Influences relate a source-port to a target-port and an update function name. The behaviour of influences is similar to updates, with the difference that only the source's value is considered for calculation of the target port value. Neither δt nor any other port values are considered for the calculation. In the growing lamp the influence `fahrenheit_to_celsius` is defined as follows:

$$\text{fahrenheit\_to\_celsius}(\mathit{bind}, \mathit{pre}, \delta t) = (\mathit{bind}(\mathit{room\text{-}temperature}) - 32) * 5/9$$

Lastly, a third type of resource flow is offered by CREST: **actions** ($\cdots\blacktriangleright$). Actions define update functions that are executed during the triggering of transitions. Similar to influences, actions are not allowed to access the δt parameter of the related update functions. The growing lamp scenario defines one action (`increment_count`) that is executed when the transition from `Off` to `On` is triggered. It is used to count the number of times the lamp has been switched on.

3.2 CREST semantics

Note that for spatial reasons this section only contains a short description of the semantics. The full, formal semantics that are based on SOS rules are provided in the technical report on CREST's formalisation [17].

CREST's semantics allow two basic ways of interaction with the system: Setting of the root entity's input values and advancing time. After either one of these is performed, the system might be in an "unstable" state. The term unstable refers to a system where, due to the interaction a transition might become enabled or an influence or update target port value outdated. To correct this situation, the system has to be *stabilised*. Stabilisation is therefore the process of bringing a system into a state where all influences and updates have been executed, and no transitions are enabled.

In the following, we describe the stabilisation process after changing port values and advancing time. Figure 2 shows a diagram that is inspired by call-multigraphs [16]. Instead of procedure calls however, the arrows represent the triggering of other semantic procedures.

Setting Values As stated, any external modification of input port values requires a subsequent stabilisation. This means that all value modifications have to be relayed to dependant ports through updates and influences. In the `GrowLamp` example, a modification of the `electricity` value has to be propagated to the corresponding inputs of the light and heat modules' input ports. These modules will in turn modify their respective output port values, which will then trigger further propagation. We see that a simple value change has to be recursively

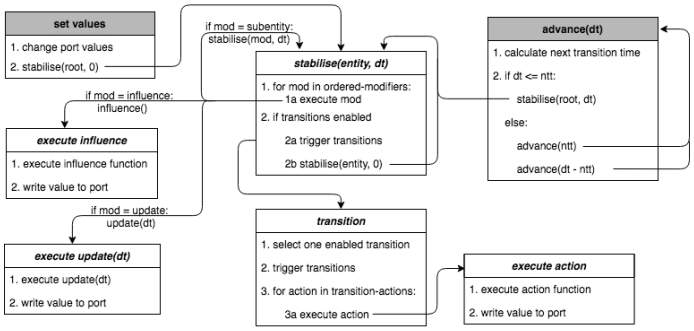

Fig. 2: An informal, schematic diagram of the semantic (sub-)processes for the `set-values` and `advance time` actions. Arcs represent the triggering of sub-procedures. Arc annotations represent conditional sub-procedure calls.

propagated throughout the entire entity hierarchy, starting at the root entity, whose inputs have been modified.

In an entity it is possible that influence and updates are "chained", meaning that one *modifier* (influence, update or subentity) changes a port which is then read by another modifier to modify a different port. Such dependencies have to be taken into account when performing stabilisation to avoid delayed or erroneous value propagation. Therefore, the entity will sort the *modifiers* so that modifiers which read one port, in the sorting appear after a modifier writing to that port. The modifiers are then executed in this order, based on their type. If the modifier is an influence or update, the specified function is executed. If it is a subentity whose input bindings changed, the stabilisation is performed inside that subentity. As a result of the sorting, the subentity's inputs will have necessarily already received their updated values (provided there are any). The testing for changed input values, and recalculation only upon their change, enforces the *reactivity* principle that we specified as a requirement. Note, that circular dependencies are not allowed within CREST systems. If there are any interdependencies between values, they cannot refer to the current time period and instead have to be expressed using a port's *pre* value to break circularity. This solution is used extensively in other languages, see Simulink's *Unit Delay* blocks and Modelica's *pre* operator.

After triggering all influences and updates, one of the enabled FSM transitions is executed, provided there is one. CREST does not prescribe a selection procedure in case of multiple enabled transitions, meaning that *non-determinism* may occur. If a transition was enabled and executed (including the corresponding *actions*), another stabilisation is started to execute all updates that are related to the new FSM state. This stabilisation phase will, again, look for enabled transitions and trigger one if applicable.

The stabilisation process operates recursively. That means that if an entity triggers a subentity stabilisation, the subentity's modifiers are executed in order

and any transitions within that subentity are triggered (followed by stabilisations) until no transitions are enabled. Only then, the control is returned back to the parent entity to continue. If there are several subentities that are independent (i.e. don't have dependencies between their inputs and outputs), they can be safely executed in *parallel*, as a result of the *locality* principle.

Note that no time passes between the update of port values and the end of the stabilisation process, whereas some other languages (e.g. Simulink) introduce a small time delay at every modification. CREST's *synchronism* can be found in languages such as Esterel. CREST differs from Esterel however, in that the entire system is stabilised instead of just the affected subset.

Advancing time The prior part of this section states that updates allow the modification of a system over time using a δt parameter. In fact, the semantics of advancing of time triggers the same stabilisation process as `set-values`, except that while `set-values` uses a $\delta t = 0$, `advance` specifies a $\delta t \geqslant 0$ as parameter. Further, all subentities perform the updates, independent of whether their input values change. This asserts that the update functions are executed correctly (i.e. according to the time parameter).

There is one particularity of time advances that has to be considered though: CREST implements eager transition triggering. This means that a transition has to be fired as soon as it is enabled. When advancing time however, it can occur that the `advance` routine is called with a δt that is bigger than the minimum time required to enable a transition. CREST implements a *continuous time* concept, that does not foresee "ticks" as system synchronisation points at which transition guards are evaluated. In order not to "miss" the precise moment when a transition becomes enabled, CREST makes use of a function that attempts to calculate the precise amount of time *ntt* that has to pass until any transition will be enabled. *ntt* is in the range $[0, \ldots, \infty)$, where 0 states that a transition is currently enabled (and that the system is not stabilised) and ∞ means that no transition can become enabled by just advancing time. Note, that the next-transition-time function depends on the implementation of updates, influences and guards and involves complex tasks such as the creation of inverse functions or the expression of the functions as sets of constraints. We will further discuss this function in the next section, with the Python implementation of CREST.

The information of the next transition time *ntt* creates two possible scenarios:

1. $ntt \geqslant \delta t$ (i.e. the time we plan to advance). CREST advances δt and the stabilisation task will execute updates and transitions until reaching a fixpoint.
2. $ntt < \delta t$. CREST divides the advance into two steps: First, *advance ntt*, advances until a transition is enabled. Updates and transitions are triggered, followed by stabilisation. Next, CREST recurses on the remainder ($advance(\delta t - ntt)$).

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

The time-based enabling of transitions extends the language and adds a *continuous behaviour* to the otherwise purely reactive system. Other synchronous languages such as Lustre need external clocks to provide timing signals.

4 CREST Implementation and Simulation

While the graphical view is convenient for analysis and discussion of a system, the creation of larger systems is more efficient when using textual representations. We therefore implemented CREST as an internal DSL in the Python language. The concept of using a general purpose programming language as host for another DSL is famously used by SystemC, which is implemented in C++.

We chose Python as a target language for three reasons:

1. Distribution and package installation allow easy installation and extension. It also comes pre-installed on various operating systems.
2. It is easy to learn, flexible, has many useful libraries, and a large community.
3. Python's internals let us alter class instantiations and hide CREST specifics from users, while still enabling the use of the default execution engine.

4.1 PyCREST implementation

PyCREST is developed as a set of Python libraries. This means the functionality can be imported and used in any standard Python program. PyCREST is developed to make use of Project Jupyter[3] notebooks as an interactive development and execution environment. Since PyCREST also features integrated plotting utilities, it is possible to create PyCREST systems and visualise them as CREST diagrams. In the following we provide a small excerpt that showcases the use of PyCREST and the definition of an entity, as displayed in Listing 3. A more complete example is provided online as an introduction to CREST[4].

Entities are defined as a regular Python class that inherits from PyCREST's `Entity`. PyCREST further provides a class for each model concept (Input, State, Update, etc.) as well as additional decorators (e.g. @influence, @transition). Entity ports, transitions and updates are defined as class attributes or decorated methods as shown in the example. PyCREST also supports many other classic Python concepts such as constructors, sub-classing, etc.

4.2 Simulation

The previous section briefly outlines the use of next-transition-time calculation. The calculation of the exact time of system changes is vital for the correct simulation of CREST systems, as CREST does not rely on artificial base clocks to identify the points for recalculation of data. Instead PyCREST's *simulator* uses Microsoft Research' Z3 [8] theorem prover to create a set of constraints that

[3] `https://jupyter.org/`
[4] `https://mybinder.org/v2/gh/stklik/CREST/sam-demo/`

```
class LightElement(Entity):
    # port definitions with resouces and an initial value
    electricity = Input(resource=Resources.electricity, value=0)
    light = Output(resource=Resources.light, value=0)

    # automaton states - specify one as the current (initial) state
    on = State()
    off = current = State()

    # transitions and guards (as lambdas)
    off_to_on = Transition(source=off, target=on,
        guard=(lambda self: self.electricity.value >= 100))
    on_to_off = Transition(source=on, target=off, \
        guard=(lambda self: self.electricity.value < 100))

    # updates are annotations
    @update(state=on, target=light)
    def set_light_on(self, dt=0):
        return 800

    @update(state=off, target=light)
    def set_light_off(self, dt=0):
        return 0
```

Fig. 3: The PyCREST definition of the LightElement entity

represents the transition's guard and searches for the minimal δt that will solve the constraints. CREST also searches all influences and updates ("modifiers") that either directly or indirectly modify the transition guard, and translates them to Z3 constraints. The creation of constraints is based on transpilation of the modifiers' source code. After the translation, Z3 is instructed to find the minimum value of δt that will enable a transition. The process is repeated for all outgoing transitions of the individual entities' current states. Finding the minimum of these results yields the next transition time.

Z3 turned out to be powerful and efficient enough for most of the CPS that we defined. However, this strong dependency also imposes limitations. Z3 can quickly find solutions to most linear constraint sets. However, some systems define non-linear constraints. An example is the ThreeMasses problem [6], where three masses are placed on a surface. One of the masses has an initial velocity and bumps into the second one, which in turn bumps into the third one, shortly after. The third mass falls off the edge of a surface and accelerates towards the ground, off which it keeps bouncing, thus repeatedly switching from upwards to downwards motion. The difficulty lies in the consideration of acceleration, velocity and position of the masses in two dimensions, as well as the repeated reduction thereof using a restitution factor.

In the presence of non-linear constraints, Z3 can only provides *a* solution to the constraint set, but cannot guarantee that it is optimized (i.e. minimal or maximal) δt value. We found, however, that the simulation is precise enough for our purposes. The ThreeMasses system is implemented in PyCREST as a

benchmark[5], displaying the capabilities of the simulator. In general, the modelling of CREST systems with non-linear constraints is discouraged until an alternative constraint solver, that adds non-linear optimisation capabilities is introduced. Further, at the time of writing, PyCREST has no special treatment of zero-crossings. In fact, all changes in behaviour, including zero crossings, are executed as usual. Zeno behaviour is discouraged and usually leads to exceptions thrown by the Python interpreter. PyCREST catches this exception and informs the user, but does not put in place any recovery procedures.

4.3 Function Approximation

It is evident that not all functions can be translated to Z3 constraints. In fact, only a subset of Python, consisting of variable assignments, unary and binary operators, and conditional statements and expressions is currently supported. Loops, recursions and function calls are not allowed in CREST.

Instead, such functions can be defined through execution traces and then interpolated. The domains of interpolation, splines and function fitting have been extensively studied in mathematical fields, and there exist many tools and libraries for the creation of interpolations and splines. CREST uses Python libraries such as SciPy and NumPy[6] for these purposes.

CREST distinguishes between influence and update approximation. Influences only depend on one particular port and are assumed to be linear in the form $A * \mathit{source\text{-}val} + B$, where A and B are the parameters to be found. The function can be piecewise defined, e.g. as step function or as shown in Figure 4.

The approximation of update functions is more complex, as updates can calculate a port's new value based on all of its entity's ports' current and previous bindings additionally to the δt time parameter. Despite the increased number of parameters, CREST tries to extract a δt-linear spline from the data provided. This is achieved by first creating an approximation of the multidimensional data and then selecting the slice of data that represents the current port values, as visualised by the dark slice of the multidimensional surface in Figure 5.

$$heat\_output = \begin{cases} 0 & \text{if } electricity_H \leq 0 \\ 60 & \text{if } 1500 < electricity_H \\ \frac{electricity_H * 0.9}{25} & \text{otherwise} \end{cases}$$

Fig. 4: Piecewise Interpolation

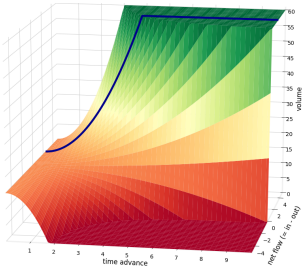

Fig. 5: Multi-variable interpolation

[5] https://mybinder.org/v2/gh/stklik/CREST/sam-demo
[6] https://scipy.org/ http://www.numpy.org/

4.4 Verification

CREST's simulation revolves around the change of a root entity's input ports (i.e. external input), the advance of time (i.e. internal state changes) and output produced by these system changes. Verification on the other hand requires the creation of the state space of the CREST system's evolution and analysis of execution traces. Due to the unbounded number of values of real-valued clocks, state spaces of timed systems are unbounded or even infinitely large. A full discussion of CREST's verification exceeds the scope of this paper. CREST's approach however is closely related to hybrid systems verification [6].

5 Conclusion and Future Works

This article introduces CREST, a domain-specific language for the definition of continuous reactive systems. CREST's target domain is the modelling of cyber-physical systems' architecture and continuous timely behaviour. The design, syntax and semantics serve the six core concepts locality, continuous time and behaviour, synchronism, reactivity, parallelism and concurrency and non-determinism. CREST achieves this by evading the base-clock concept, while still preserving synchronism and choosing synchronisation points based on system behaviour. This trait permits continuous value changes, arbitrarily fine time advances and convenient modelling on largely different time scales within the same model. It also allows for the efficient simulation of behaviour and time. CRESTensures a hierarchical structure that facilitates composition. The language supports concurrency and parallelism, as they are a omnipresent in both software and physical worlds. The automaton-based behaviour of CREST entities enables to easily capture the non-determinism and complexity of CPS.

While CREST shows promising results, we see several areas of improvement:

- Currently the calculation of the next transition time has rudimentary support for one type of interpolation and approximation. We aim to extend our research into different algorithms to provide better results.
- We are studying the automatic generation of controllers from CREST models.
- We are developing a property language that allows non-expert users to define queries in a language that part of their systems. This facilitates removes the need to know language such as LTL or CTL temporal logics.

References

1. Ashenden, P.J.: The Designer's Guide to VHDL. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edn. (2008)
2. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming (1992)
3. Black, D.C., Donovan, J., Bunton, B., Keist, A.: SystemC: From the Ground Up. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2010)

4. Bourke, T., Pouzet, M.: Zélus: A Synchronous Language with ODEs. In: 16th International Conference on Hybrid Systems: Computation and Control (2013)
5. Brooks, C., Cataldo, A., Lee, E.A., Liu, J., Liu, X., Neuendorffer, S., Zheng, H.: Hyvisual: A hybrid system visual modeler. Tech. Rep. UCB/ERL M05/24, EECS Department, University of California, Berkeley (2005)
6. Carloni, L.P., Passerone, R., Pinto, A., Angiovanni-Vincentelli, A.L.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation (2006)
7. Colgren, R.: Basic Matlab, Simulink And Stateflow. AIAA (American Institute of Aeronautics & Ast (2006)
8. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proc. 14th Int. Conf. Tools and Algorithms f.t. Construction and Analysis of Systems (2008)
9. Feiler, P., Gluch, D., Hudak, J.: The Architecture Analysis & Design Language (AADL): An Introduction. Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University (2006)
10. Fischer, J., Holz, E., Löwis, M., Prinz, A.: SDL-2000: A Language with a Formal Semantics. Rigorous Object-Oriented Methods 2000 (2000)
11. Franca, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex – experiments and roadmap. In: 12th IEEE International Conference on Engineering Complex Computer Systems (2007)
12. Haber, A., Ringert, J.O., Rumpe, B.: Montiarc - architectural modeling of interactive distributed and cyber-physical systems. CoRR (2014)
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proc. of the IEEE (1991)
14. Hamon, G., Rushby, J.: An Operational Semantics for Stateflow. Fundamental Approaches to Software Engineering (2004)
15. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences **57**(1) (1998)
16. Khedker, U.P., Sanyal, A., Sathe, B.: Data Flow Analysis - Theory and Practice. CRC Press (2009), `http://www.crcpress.com/product/isbn/9780849328800`
17. Klikovits, S., Linard, A., Buchs, D.: CREST formalization. Tech. rep., Software Modeling and Verification Group, University of Geneva (2018). https://doi.org/10.5281/zenodo.1284561
18. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (OMG MARTE) Version 1.1 (2011), `https://www.omg.org/spec/MARTE/1.1/PDF`, OMG Document Number: formal-2011-06-02
19. Object Management Group: OMG Systems Modeling Language (OMG SysML) Version 1.5 (2017), `https://www.omg.org/spec/SysML/1.5/PDF`, OMG Document Number: formal-2017-05-01
20. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
21. Raskin, J.: An introduction to hybrid automata. In: Handbook of Networked and Embedded Control Systems (2005)
22. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. CoRR (2015)
23. Thomas, D., Moorby, P.: The Verilog Hardware Description Language. The Verilog Hardware Description Language, Kluwer Academic Publishers (1996)
24. Vlaovič, B., Vreže, A., Brezočnik, Z., Kapus, T.: Verification of an SDL Specification — a Case Study. Elektrotehniški vestnik (Electrotechnical Review) (2005)