Software and Systems Modeling (2021) 20:837–866
https://doi.org/10.1007/s10270-020-00831-4

REGULAR PAPER

Pragmatic reuse for DSML development

Composing a DSL for hybrid CPS modeling

Stefan Klikovits[1,2] · Didier Buchs[1]

Received: 29 October 2019 / Revised: 26 July 2020 / Accepted: 14 September 2020 / Published online: 14 October 2020
© The Author(s) 2020

Abstract
By bridging the semantic gap, domain-specific language (DSLs) serve an important role in the conquest to allow domain experts to model their systems themselves. In this publication we present a case study of the development of the Continuous REactive SysTems language (CREST), a DSL for hybrid systems modeling. The language focuses on the representation of continuous resource flows such as water, electricity, light or heat. Our methodology follows a very pragmatic approach, combining the syntactic and semantic principles of well-known modeling means such as hybrid automata, data-flow languages and architecture description languages into a coherent language. The borrowed aspects have been carefully combined and formalised in a well-defined operational semantics. The DSL provides two concrete syntaxes: CREST diagrams, a graphical language that is easily understandable and serves as a model basis, and `crestdsl`, an internal DSL implementation that supports rapid prototyping—both are geared towards usability and clarity. We present the DSL's semantics, which thoroughly connect the various language concerns into an executable formalism that enables sound simulation and formal verification in `crestdsl`, and discuss the lessons learned throughout the project.

Keywords Cyber-physical systems · Domain-specific language · Modeling · Simulation · Verification

1 Introduction

Modeling and simulation are used by engineers to design, probe and verify their systems before construction, thereby reducing design flaws and increasing the development speed. From large monolithic installations such as oil platforms to wide, distributed networks (e.g. electrical power grids), to intricate robotic designs for medical applications, the use of these modeling techniques is indispensable. To aid the process of model creation, system engineers rely on a broad set of formalisms, languages and tools. The ever-growing diversity allows expert modellers to choose the most appropriate formalism for their particular system, and even use a combination of different languages to model the various aspects of their system. Modern modeling platforms therefore often support the choice between different formalisms and languages (e.g. Matlab Simulink [58]) or even encourage the use of several modeling paradigms in concert (e.g. Ptolemy II [70]). These developments enable the creation of large-scale and high-performance models but require a lot of experience with the individual languages and a good knowledge of the underlying modeling principles. The required knowledge, the steep learning curves and the often significant financial investment are major deterrents for the adoption of modern systems engineering best practices by non-expert application creators and developers of small-scale installations.

In recent years, consciousness of this problem led to an increased interest in domain-specific modeling solutions. Thus, existing products are modified to more closely represent the needed system features, rather than generic concepts. The *semantic gap* describes this "distance" between a model and the original installation. Wide gaps are often the result

Communicated by Gabor Karsai.

✉ Stefan Klikovits
klikovits@nii.ac.jp

Didier Buchs
didier.buchs@unige.ch

1 University of Geneva, Route de Drize 7, CH-1227 Carouge, Switzerland

2 Present Address: ERATO Meta-Mathematics for System Design Hasuo Project, National Institute of Informatics, Tokyo, Japan

of using tools or languages that cannot properly represent the system's features and frequently demand elaborate workarounds. One common approach is the use of DSLs, languages dedicated to creating models at an abstraction level where domain users are able to model their systems themselves without having to learn another language.

In this article we describe the development of the Continuous REactive SysTems language (CREST), a domain-specific modeling language (DSML) [69] for the hybrid modeling of cyber-physical systems (CPSs). The language targets the creation of CPS models of small-scale, custom systems such as smarthome applications, automated gardening setups and office automation installations. Usually, these CPSs are compositions of off-the-shelve components that interact through the transfer of physical resources such as electricity, heat and water. CREST supports creators of such systems by putting a focus on usability and simplicity. Its target audience includes novice modellers, private developers as well as system creators and maintainers who want to add trust into their existing applications. For this purpose, the DSL joins a system's behavioural and structural concerns inside the same concrete syntax, which lowers its entry-barrier and allows systems engineering newcomers to quickly take advantage of the benefits of simulation and verification. Throughout the paper, we elaborate on our language development methodology, starting from the requirements analysis and evaluation of existing solutions, to the definition of the language's abstract syntax and formal semantics, to the implementation of the language itself. In the process, we describe particular design choices such as the reuse and combination of well-known concepts from existing formalisms into a coherent language. This method builds upon a vital formal syntax and semantics description that asserts a sound combination of the various language aspects, without which, it would be much more difficult to perform advanced modeling tasks such as simulation and verification. The DSL's formalisation itself is provided as an appendix to this article. CREST's pragmatism is manifested in the form of `crestdsl`, an *internal* DSL implementation based on the Python general-purpose programming language (GPPL). Employing a widespread programming language increases CREST's usability by allowing modellers to rely on popular development environments and established execution platforms that they are already familiar with. This lowers the entry bar to modeling, as users prefer to continue working with the tools they already know [73]. Even though it will probably not be required for most audiences, `crestdsl`'s GPPL-basis additionally allows power-users to extend the DSL using scripting application programming interfaces (APIs), interact with common data analysis packages and create object libraries to increase development speed and reusability.

The rest of this paper is organised as follows: Sect. 2 provides the scientific background on the use of DSLs, highlights examples and provides detailed comparisons of different approaches. Section 3 dives into the methodology that we followed for the creation of CREST, a formalism for the specification of hybrid CPS models. First, Sect. 3.1 analyses the target domain requirements towards CREST. Section 3.2 outlines the components of the CREST formalism, and highlights the concepts that were borrowed from existing languages to create a coherent formalism. Sections 3.3 and 3.4 introduce CREST diagrams, CREST's graphical syntax, and `crestdsl`, its implementation as pragmatic internal DSL, respectively. Section 4 describes CREST's operational semantics and its use for simulation and verification. Both aspects are introduced from a CREST-based view, followed by their practical description in `crestdsl`. The section also includes a brief outline of `crestdsl`'s satisfiability modulo theories (SMT) approach to discover the points in time where discrete behaviour changes, i.e. state transitions, occur. This is a vital concept for our DSL's implementation. Section 5 reviews our method and discusses valuable insights into the development process, lessons learned and critically evaluates potential pitfalls that should be avoided. Section 6 compares our approach to related research efforts. Section 7 outlines future developments of our project, and Sect. 8 concludes.

2 Background—from modeling to domain-specific languages

Systems modeling is widely accepted as a standard item in the engineering toolbox. The models facilitate system conceptualisation, design and verification by reducing complexity and hiding unimportant implementation details [55], and often allow simulation and analysis even before the actual system is being built. In many cases, different models are created, each focused on a particular viewpoint such as system architecture, safety, concurrency or performance. Depending on the specific task at hand, engineers choose a suitable modeling formalism (or language) for the appropriate representation of the particular system aspect [16]. Nonetheless, abstract mathematical system descriptions are conceptually distant from the actual system. For instance, formalisms such as Discrete Event System Specification (DEVS) [81], Petri nets [68] or Bond graphs [15] are designed to be generic and application agnostic, but oftentimes so abstract that it is difficult to recognise the original system within the actual model. This leads to an often-criticised lack of practicability and applicability [43]. Sometimes it is even necessary to develop workarounds to assert that the actual system behaviour can be expressed using formalism's semantics. A common example is the discretisation of models to be able to use languages and tools that do not support continuous evolution. Such a

divergence between the system and its model, where system behaviour has to be approximated or modelled using inappropriate means is commonly known as *semantic gap* [30]. Wide semantic gaps, often created by operation on the wrong abstraction level, lead to confusion and problems not related to the actual system itself. The results are understandability issues and risks of introducing hidden bugs [74]. To overcome the complexity, domain experts require the help of professional system modellers who first need to learn about the system's behaviour and then correctly translate the system to the formalism's semantic domain. Obviously, misunderstandings and communication problems are common.

2.1 Modeling languages and tools

An alternative approach to employing expert modellers is to allow domain users to model their systems themselves. To do so, they require means that are understandable even without extensive systems engineering training, experience or modeling knowledge. As displayed in Fig. 1, these approaches still operate on a generic level, but usually provide helpful features for the representation of system engineering concerns. Modeling languages such as the Unified Modeling Language (UML) [64], the Specification and Description Language (SDL) [48] or Modelica [34] raise the abstraction level and bring models closer to the system domains. They allow convenient abstractions to improve system design and reasoning, while still offering translation to various low-level formalisms or directly to software source code. However, even though these languages allow systems to be modelled at an abstraction level that is closer to the original, they tend to be still very generic and involve steep learning curves. For instance, users usually require several months of continuous exposure before they become proficient with UML [33]. This means that the main users of these languages tend to be professional system developers, rather than domain end-users, i.e. the people who will actually build and use the system on a daily basis. In fact, to enable domain users to also model their own systems, it is necessary to provide them with the capability to use modeling tools that operate in the system domain level, using the concepts, terminology and tools that the users are already familiar with. This approach removes steep learning curves and avoids the risk of confusion and misconceptions.

2.2 Domain-specific approaches

In the last few decades, this final domain-adaptation step has evolved in several forms, as shown in Fig. 1. A common approach is to adapt an existing modeling language by altering its concrete syntax (and sometimes its semantics) to more closely represent the features found in the actual system domain. Business process management mod-

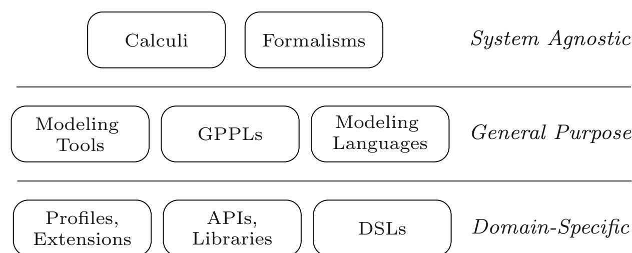

Fig. 1 Levels of domain adaptation in systems modeling

els [78], for example, are inspired by Petri nets and allow the modeling and analysis of business workflows. The matching BPMN [62] language offers a graphical interface for simplified representation. In a similar way, the Architecture Analysis and Design Language (AADL) [75] can be extended using the AADL *annex* mechanism. Such annexes can be used to add discrete behaviour [29], continuous behaviour [2] or error modeling capabilities [23] to the language. UML can be adapted to various domains using UML *profiles* [36], a UML-native means to extend and adapt the language. The UML Profile for Modeling and Analysis of Real Time and Embedded systems (MARTE) [63] for instance, is an extension of standard UML that offers concepts for the modeling of real-time applications. The Systems Modeling Language (SysML) [65] is a well-known adaptation of UML that alters and extends a specific subset of UML to allow dedicated systems engineering. However, both SysML and MARTE can be seen as generic languages themselves, as they are not applicable to a specific target domain. A more concrete experience report of using UML profiles for implementing domain-specific modeling is described in [25], where the authors adapt the language specifically for robotic applications.

Another approach to incorporate domain knowledge into a modeling tool or language is the use of tool libraries. Especially when using versatile modeling platforms such as Simulink or Ptolemy II, it is possible to create domain libraries that aid users by providing reusable APIs or objects. The Modelica language is well-known for its large number of object libraries,[1] that add capabilities to model specific domains such as chemical processes, hydraulic flows or power systems.

2.3 Domain-specific languages

A third method to close the semantic gap between model and domain is to create a dedicated modeling language for a specific target domain or application [69]. DSLs [80] raise the abstraction level of system models to facilitate model creation and reasoning, while at the same time hiding the

[1] https://www.modelica.org/libraries.

underlying complexity of the modeling process. Ideally, this leads to DSLs that allow domain users to easily perform all their tasks using pre-existing knowledge and without the need to newly familiarise themselves with a tool or language. This increases productivity and makes the language easier to comprehend [39]. Language engineering experts work in concert with domain users to learn about their needs and trim the DSL according to these requirements. Hidden behind the DSL, the data are then executed using purpose-built tools or translated to other, well-established modeling languages in order to use already existing execution software. The recent popularity of DSLs is manifested in the numerous conferences and workshops organised on the topic, and seminal books such as [80] and [28] show that the field is reaching maturity, with established best practices, available experience reports and educational resources. Despite all the benefits that they bring, the classical language engineering approach comes with a hefty price tag [79]. DSLs require a significant upfront investment into the initial creation and subsequent maintenance of their infrastructure [24]. Even small DSLs need at least the provision of a parser, a text editor and an execution engine or code generator (i.e. a translator to another language). Larger projects often require further development of a textual or graphical integrated development environment (IDE), auto-completion engines, static and dynamic code analysis tools, language and IDE versioning and version migration, interfaces to GPPLs, library support, etc. These needs and the popularity of DSLs led to the creation of *language workbenches* [27] such as Xtext [11], MetaEdit+ [77] and MPS [17], that promise help throughout the language design phase, but also convince through automatic out-of-the-box generation of many of the aforementioned artefacts. For instance, Xtext can generate a full-fledged IDE including parser, auto-suggestions and code analyser based on a DSL grammar alone. Its tight integration into the Eclipse Modeling Framework (EMF) [76] allows the facilitated development of code generators, model transformation engines or graphical editors for these languages. Nonetheless, even after the use of language workbenches, DSLs often require further customisation and subsequent maintenance effort that can easily become very costly in terms of time and development effort.

Internal DSLs [39] are an alternative to the creation of standalone languages where DSLs are embedded inside another, usually more generic language (e.g. a GPPL). This approach is related to the creation of tool libraries, so that it is hard to clearly distinguish between domain-specific libraries and internal DSLs. Generally, it can be said that—in contrast to tool libraries—internal DSLs tend to embed advanced features such as their own semantics inside their host languages, allowing for sound execution and verification. Further, internal DSLs often “use, and abuse, the host language” [31] by using metaprogramming, reflection and clever language tricks to implement domain-specific behaviour. Next to the reduced initial DSL development time and cost, users can rely on their pre-existing familiarity with compilers, IDE, code analysis frameworks and development best practices. Internal DSLs are usually easily extensible and integrate better into existing language infrastructure and tooling [20]. This is also reflected in their increased interoperability with other tools and third-party libraries. Depending on the particular host language, numerous software packages are available to further extend the DSL’s capabilities.

A well-known example of an internal DSLs is SystemC [12], a hardware description language that is implemented in C++ and distributed as a software library. Models are created using regular C++ source code that instantiates library classes and executes macros. SystemC’s semantics are implemented in a simulator module that is shipped alongside the library. Other wide-spread examples for internal DSLs include various testing frameworks (e.g. jMock [31], XUnit [8], Chai[2]) and extensions for programming languages such as Ruby [32] (e.g. Ruby on Rails [4]), Groovy and Scala [39].

Internal vs. External DSLs The clear advantage of using internal DSLs lies evidently in the reuse of the host language’s syntax and execution environment [38]. One disadvantage of this reuse, however, is the lack of static checking support. Internal DSLs often require models to be setup in a certain manner, use specific properties (e.g. inherit from specific base classes) or prohibit the use of some host language features (e.g. introspection, certain libraries, etc). While the lack of static checking is a DSL problem in general [44], external DSLs can benefit from static type and syntax checking offered by language workbenches. This means, that external languages can be defined to natively enforce these rules inside the parser or interpreter and display the results directly in the IDE. Internal languages on the other hand have to build these features outside of the GPPL’s infrastructure, such that users have to manually trigger syntactic and semantic sanity checks. In the best case, they might use language reflection and introspection APIs to create validation scripts and thereby inform the user of potential problems. However, typically manual execution of these scripts is required, which delays the feedback and might even cause frustration if the suggestions are incomprehensible.

Another issue might arise from choosing the wrong host language. The benefit of reusing the syntax also means that the host language’s syntax cannot be adapted or extended. For instance, Matlab requires an ellipsis (“…”) to be written at the end of each line of multi-line expressions. This feature can become problematic when designing so-called fluent

[2] https://www.chaijs.com.

interfaces that chain many method calls as described in [73]. In such situations, external DSLs show more flexibility, as they can support a highly customised syntax. The disadvantage here however often lies in the need to define the entire language and interpretation from scratch. Especially when it comes to adding "basic building blocks" such as mathematical expressions, operators, character string manipulations and type systems, the development process can easily become tedious. Modern language workbenches attempt to provide reusable blocks of syntax and grammar (e.g. Xtext's support for Mixins and Traits [71]), but still require the execution environment to be defined and adapted for the newly created language.

Summarising, one can see that external DSLs provide more powerful, flexible and customisable features, that require time and effort to be created. Internal DSLs on the other hand take advantage of the out-of-the box features, but are limited by the host language's syntax, execution platform and IDEs. In the end, it depends on the language developer's insights to decide whether the benefit of a familiar GPPL syntax and cheap reuse of existing execution platforms outweighs the flexibility of external DSLs.

3 Methodology

Our project focuses on the modeling of CPSs whose behaviour is strongly influenced by the flow of physical resources such as light, electricity, water and heat. In this section, we describe the evaluation of modeling requirements towards a language for our systems, our search for an appropriate candidate and the reasons that led us to develop our own formalism and two modeling languages building upon this formal basis.[3]

3.1 Language requirements

The approach we followed for this project is driven by the practical need of a tool or language, that brings the benefits of modeling and simulation to smaller-scale systems. While the research effort of the last few decades has produced a plethora of modeling formalisms, languages and tools that have been applied to various projects in industries such as transportation, avionics, manufacturing, heavy industry and safety-critical systems, very little effort has been made to transfer these developments to the end-user market. In particular, our target user group are creators of CPSs such as smarthomes, office automation installations and automated plant growing applications. Even though such systems usually are not classically safety-critical, i.e. do not pose harm to health or life, their correct functionality is nonetheless important. A plant watering setup that is assembled by a private end-user might cause to water spillage and break wooden floors or electrical devices, thereby creating significant financial damages. Similarly, a misconfigured office automation system that was installed by a non-expert building manager might lead to high bills if e.g. the lights and heating are run even if no employees are present. In the current situation, such users might shy away from creating their applications as the risk of faulty installations is high and they lack the time, capacity and financial means to use existing modeling and verification solutions.

Before starting the development, we first designed three case study systems that allowed us to evaluate the main requirements of resource-flow CPSs. Attention was paid to choose applications that represent the heterogeneity of our target domain. While a full description of each would exceed the scope of this article, we still provide a brief outline. A complete description of the individual case studies can be found in [50]. The three systems were designed as follows:

1. A smarthome system that connects appliances and Internet of Things (IoT) devices such as automated vacuum cleaners, televisions, a "smart" dishwasher and remote controlled lights. Additionally, the house features a solar power system and battery that produces electricity during periods of sunshine. The system also has control over the electric hot water boiler, which provides water to the shower and dishwasher. The modeling aim is to reduce resource consumption by performing tasks when cheap solar electricity is available. Additionally, the user's schedule is considered to e.g. do the noisy vacuum cleaning when the user is at work.
2. Our second system is an office automation system whose purpose is to monitor the temperature and lights in a small office environment. The goal is to automatically maintain a productive work environment. Depending on the environmental observations (e.g. temperature, brightness), the system controls window blinds, ceiling lamps and air condition units. A constraint of the system is that the emergency exit path is always lit when employees are present.
3. The third system is an automated indoor gardening application. It uses soil moisture and light sensors to measure the environment conditions and control lamps, heaters and water pumps to assert ideal conditions that lead to maximum harvest, while at the same time reducing resource consumption. Figure 2 shows a schematic representation of the system.

A comparison of these systems resulted in the discovery of two important similarities. First, the types of systems

[3] In this work we follow the definitions of [16], where languages implement formalisms. Thus, CREST is implemented by two languages with different concrete syntax.

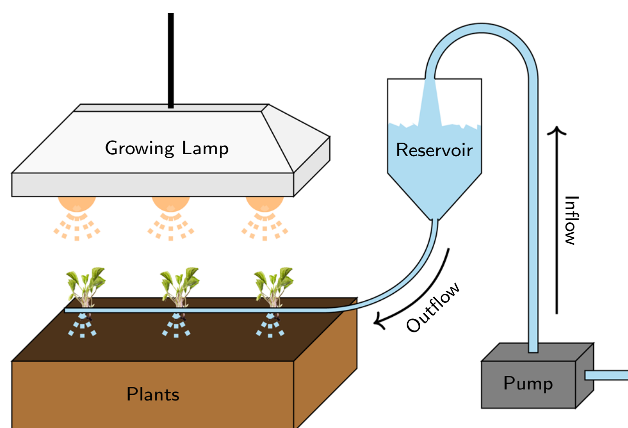

Fig. 2 Schema of the plant system's components

we target in our study are commonly composed of off-the-shelf components. This means, it is highly unusual that system creators use custom components. Instead, they buy devices (e.g. lamps, home appliances, IoT gadgets) according to their needs and connect them to compose their system. Second, despite modern communication interfaces, the influences between the individual components are in the physical domain. For example, a dishwasher consumes hot water and electricity, a vacuum cleaner uses electricity to clean the floor and creates noise in exchange, and plants require sunshine or artificial light on a daily basis.

Next to these two principal paradigms, we identified ten aspects that have to be supported by a modeling language or tool, in order to be able to address all necessary system concerns. These aspects can be split into two groups, such that the first seven describe "functional" properties that can be objectively analysed:

Reactivity allows systems to adapt to changes in their environment. For instance, all three case study systems need to react to daylight changes and activate the lamps when it becomes dark.

Synchronism allows to model the immediate influences between system components and the momentary, discrete CPS behaviour changes. This is necessary, as virtually all physical effects are instantaneous, such as e.g. the flow of water when the shower is turned on or the spreading of light.

Parallelism is a vital concept in systems modeling, as many effects can take place at the same time. In our smarthome for example, the use of a shower draws water from the hot water boiler, which causes it to be replenished with cold water and the heating rod to activate at the same time. In turn, the heating process consumes electricity from the solar panels or the electricity mains. All these actions need to be executed at the same time and update the system state concurrently.

Locality is required since our systems are built from individual components. A clear and coherent composition mechanism facilitates system assembly. For instance, despite the water boiler's external influences (water in and out flows, electricity), the water temperature inside is locally determined and modified, and not set from the outside.

Continuous Time is necessary to express physical effects, such as continuous resource flows which are usually described by differential equations.

Non-Determinism is omnipresent in the real world due to unforeseeable behaviour of various components. Thus, it should also be expressible in our models. As an example, we can look at a light bulb. When the electricity starts flowing, it can either produce light, or alternatively break and cause a fuse to trip due to the power surge. Since both effects can occur in the real system, it is important to also model both.

A *Formal Basis* is required for advanced modeling tasks such as simulation and formal verification. Without a clear operational semantics, the models become imprecise and potentially invalidate verification results.

The remaining three language criteria largely depend on the target users, their pre-existing knowledge and preferences. This means that while they first seven *key aspects*, can be objectively evaluated, the *additional criteria* result from the proper integration of the key aspects and their good correspondence from a syntactic and semantic point of view. Evidently, a sound, formal language definition supports this integration.

Usability describes how easily a modeling tool can be learned and used. It depends on the users' knowledge and capabilities. For instance, novice modellers usually require clear instructions, whereas experts tend to prefer extended configurability and adaptability.

Suitability is a property that states whether a language or tool is apt for the use in a given task. A lack of suitability often results in a wide semantic gap.

Expressiveness describes whether a modeling means is capable of expressing all required domain concepts.

Equipped with these ten criteria, we set out to find a suitable language for the modeling of our case study systems. However, to the best of our knowledge, we did not find any language or tool that supports all our key properties, is usable by non-experts and novice system engineers and suitable for the modeling of smaller-scale resource-flow CPSs. Existing languages and tools that offer physical resources modeling usually target expert engineers from financially potent enterprises in the industrial sector. Their typical application domains include large scale systems such as oil rigs

Table 1 Modeling language evaluation for resource flow CPSs

Formalism/tool	Reactivity	Synchronism	Parallelism	Locality	Continuous	Non-determinism	Formal Basis	Usability	Suitability	Expressiveness
UML / MARTE	✓	✓	✓	✓	✓	✓	∼	×	∼	✓
SDL	✓	×	✓	✓	✓	✓	✓	✓	∼	✓
AADL + Beh.Ann	✓	✓	×	✓	✓	✓	∼	×	×	∼
MontiArcAutomaton	✓	×	✓	✓	×	✓	✓	✓	∼	∼
SystemC	✓	×	∼	✓	×	∼	×	∼	∼	✓
Esterel	✓	✓	✓	∼	×	×	✓	×	×	∼
Lustre	✓	✓	✓	✓	×	×	✓	×	×	∼
Zélus	✓	✓	✓	×	✓	?	×	×	×	?
Simulink/Stateflow	✓	×	✓	✓	✓	✓	×	∼	✓	✓
Modelica	✓	✓	✓	✓	✓	✓	×	×	∼	✓
PowerDEVS	✓	✓	×	✓	∼	×	✓	✓	×	∼
	Key aspects							Add.crit.		

Reprint from [50]
Symbols: ✓ *(Yes)* × *(No)* ∼ *(to some extent)* ? *(unknown)*

and power plants. Other, academic tools lack usability and therefore also require a long training phase. The tools that we selected for closer evaluation represent a range of different modeing languages that are actively being used in the domain of CPS modeling. We aimed to cover a broad spectrum of approaches, covering general purpose languages (e.g. MARTE, SDL), architecture description languages (e.g. AADL [75], MontiArcAutomaton [72]), hardware description languages, synchronous programming languages (e.g. Esterel [10], Lustre [42], Zélus [13]), event-based approaches (e.g. the quantised state [54] extension of PowerDEVS [9]) and modeling platforms such as Simulink/Stateflow and Modelica. The selection is based on an informal upfront evaluation and feature comparison. For languages that are very similar, we chose the more promising candidate. Table 1 shows our evaluation based on the previously obtained requirements. It can be seen that most languages lack at least one of the key requirements. MARTE and Modelica appear to be favourable candidates, although both of them are inapt in terms of usability and suitability. MARTE for instance builds upon UML's fourteen diagram types and requires the knowledge of even more languages such as the Object Constraint Language (OCL).[4] This significantly expands the language's learning phase. Furthermore, the language adds timing annotations to the system which can be (ab-)used to describe continuous variable evolution, but does not provide dedicated concepts. Modelica on the other hand provides this possibility, but requires the creation of a domain library to augment its usability for our target domain, which is a non-trivial task. Furthermore, its textual syntax is difficult to understand for newcomers and requires a lot of experience for proficiency. A more complete and detailed description of our evaluation, including discussion of the other languages and formalisms is provided in [50] and [52].

3.2 CREST—concept

Based on the previous evaluation and the lack of an appropriate modeling language for the CPSs and system creators targeted by our research, we decided to develop CREST as a standalone project, rather than an extension or adaptation of an existing product. CREST's goal is to provide an easily comprehensible and usable solution to CPS modeling. Thus, for instance, from a user's perspective, CREST should be easy to learn and use, but flexible enough to model the broad range of devices that are installed in modern smarthome and office automation projects. On the development side, we decided to reuse well-known abstractions and parts from established modeling formalisms and languages. This allows us to rely on a large body of knowledge, build on existing

[4] https://www.omg.org/spec/OCL/About-OCL/.

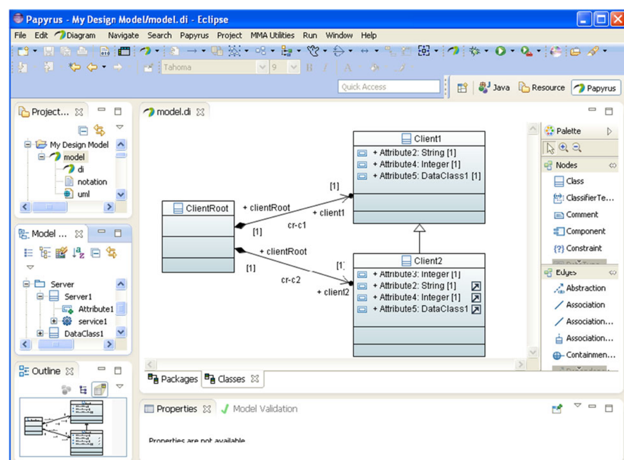

Fig. 3 Papyrus modeling editor . Source: Eclipse (Neon) Help—Papyrus guide

tools by transforming CREST models and cater to the knowledge of users with pre-existing modeling skills.

CREST's main purpose is to support its users through three principal modeling tasks namely, the *modeling*, i.e. the creation of a system model, *simulation*, i.e. the evaluation of dynamic runtime behaviour and *verification*, i.e. the formally sound evaluation whether a model can reach certain beneficial or malicious states at all.

For the modeling itself, developers require a tool or language that helps in the rapid prototyping and subsequent refinement of their system and components. The provided modeling means has to be both easy to handle and also serve as an understandable resource for discussion. During the aforementioned evaluation of other modeling languages, we observed that these two requirements are often opposing. Graphical modeling languages such as UML or SDL encode data using easily understood information such as shape, size, colour and position. Their advantage over textual languages is the facilitated comprehensibility and the quick identification of required modifications [59]. The problematic however lies behind the scene. A clear graphical syntax has to be carefully engineered [67] and the creation of complex graphical models often requires the use of cumbersome graphical editors [18] which demand manual positioning and sizing of components, drawing of connection edges and frequent switching between keyboard and mouse. Especially in the prototyping phase, where models have to be edited and redrawn frequently, the creation of graphical models can quickly become a tedious task. Figure 3 shows a screenshot of the wide-spread Papyrus Modeling Editor, a common tool for UML modeling. Note, how several views need to be used in combination for navigation and model creation.

Textual modeling languages such as Modelica and SystemC overcome these problems by using well-structured text to express system architecture and behaviour. Models encode relationships between components and subcomponents using keywords, similar to programming languages. The advantage of these types of modeling languages is the quick model creation and manipulation, since the focus remains on the actual model, rather than positioning, shape or size of its representation. They also tend to be more powerful in terms of "intelligent editors and copy&paste support" [1]. Further, they often benefit from advanced features such as IDEs, static analysis, code suggestions, as they are more easily created for textual than graphical languages. On the other hand, it is clear that textual models usually are not self-explanatory and thus require secondary notions such as accompanying documents or code comments to carry additional information for developers and users [67].

In the CREST project, we follow a pragmatic approach that resulted in the creation of a textual and a graphical language, such that their respective concrete syntax implements the same formalism. Thus they share the same abstract syntax and operational semantics. *CREST diagrams*, the graphical language, reuses abstractions and concrete syntax from other widespread languages to increase comprehensibility and usability. To build upon pre-existing knowledge, CREST diagrams reuse notational conventions from well-known formalisms such as automata and architecture description languages. Evidently, the language suffers from the same drawbacks as other graphical languages, in that the positioning and resizing of elements is intricate and time consuming. Thus, to increase the development speed and add the benefits from textual modeling, we created `crestdsl`, an internal DSL in Python. It offers the same expressive modeling power as CREST diagrams and adds reusability to the modeling by supporting classic programming concepts such as inheritance and shared code. `crestdsl` further poses as a flexible bridge to the implementation of advanced modeling tasks such as simulation and verification. The DSL facilitates the model creation and invites the development object and domain libraries, providing scripting APIs that also support user-developed language extensions and tooling. Due to their expressive equivalence, CREST diagrams and `crestdsl` models can be transformed from one representation into the other, and, `crestdsl` does in fact provide functionality to create interactive CREST diagrams based on its models. A discussion of the translation functionality is provided in Sect. 5. The next two sections introduce both CREST diagrams and `crestdsl`, and highlight our design choices.

3.3 CREST diagrams

CREST diagrams [50,53] are a graphical modeling language geared towards usability. Its purpose is the modeling of physical resource flows between the components of CPSs. Thus, it contains dedicated features to address concerns such as

continuous flows and component hierarchies. The language implements a formal syntax and structure specification that clearly states requirements and constraints of CREST models (see Appendix 1). In this section, we use CREST diagrams to introduce the structure of a typical CREST model. Note, that the CREST diagram syntax reuses modeling notions known from other modeling languages to increase familiarity.

For simplicity, CREST diagrams combine architectural and behavioural system aspects within the same language. The advantage of this approach is that especially novice modellers have all information directly displayed to them, without the need to switch between different viewpoints.

Architecturally, CREST models are primarily defined through a system structure whose components (**entities**) are arranged strictly hierarchically. Thus, each entity only has one parent and a CREST system has only one, single *root* entity that defines the system's scope. CREST's behaviour description is inspired by finite state machines (FSMs) and hybrid automata. This means, that every entity defines a state-automaton that represents its behaviour modes. In each state, the system can expose different behaviour, such that a device might produce other output when it is turned on than when it is turned off, for instance.

Entities enforce CREST's *locality* requirement and hide their internal structure from the outside. Hence, an entity's automaton states cannot be read from another entity and it is not possible that one entity's state automaton directly influences the output of another entity. Another advantage is that within the entity hierarchy, a subentity can be treated as coherent "black box", whose system state is always up to date and does not have pending state updates. This concept is very important for CREST's semantics (see Sect. 4.1), which revolve around the creation of such "stable" system states, by locally searching for fixpoints on the lowest entity levels and recursively ascending in the entity hierarchy and stabilising until the entire system is stable.

An entity's black box view also exposes its communication interface, consisting of **input** (▭) and **output** (▭) ports. Similar to some common architecture description languages (e.g. MontiArc [40]), CREST's ports define which type of value they accept. These types (called **resources**) consist of a unit (e.g. Celsius or Lumen) and a value domain such as natural numbers $\mathbb{N}$, real numbers $\mathbb{R}$ or discrete sets (e.g. {on, off}). Next to its name and resource, a port also defines its current valuation. Inputs and outputs are complemented by a third type of port: **locals** (▭). This port type is not part of the communication interface, but rather serves as internal storage of data.

CREST's behaviour definition is inspired by FSMs. Thus, each entity defines a set of **states** and guarded **transitions**

(e.g. Off —guard→ On). Transitions relate two states and a *guard* function, such that the automaton switches to the new state if the transition is enabled, i.e. if its guard condition holds. Usually, guard functions will access the entity's port values for their evaluation. Note, that the locality principle is enforced here as well. Thus, to preserve consistency, transition guards can only use their own entity's ports and the output ports of subentities for the computation.

Continuous behaviour, i.e. the flow of resources, can be modelled in several forms. The most fundamental way are **update** functions (- ►). Updates extend the automaton behaviour in a similar way to hybrid automata (HA). However, instead of defining each variable's rate in each state (as in HA), updates only modify specific port values. Specifically, an update relates a function to an automaton state and a target port, such that the function is continuously executed while the automaton is in its related state and the result is written to the port. For instance, a device might define that an update modifies the value of a specific output port while it is in state on. To add timing behaviour to the language, an update function also has access to $\delta t \in \mathbb{R}_{\geq 0}$, the amount of time that has passed since it was last executed. This means that in theory, updates are executed in infinitesimal intervals ($\delta t = \varepsilon$) to produce continuous behaviour. Practically though, CREST's implementation is responsible to execute updates as often as needed and often very coarse time steps can be achieved.

CREST further offers **influences** (⟶) to statically link two ports. Each influence relates two ports and a function in a way, such that the function is executed with the source port's value as parameter and the result is written to the target port. **Actions** (···►) are similar to updates, except that they are executed at the exact point when a transition is triggered. This also means that they do not have access to a δt value. Note that influences and actions are purely syntactic extensions of CREST to increase the DSL's usability. When required (e.g. for static analysis purposes) influences can be replaced by sets of updates, actions by introducing additional automaton states and transitions.[5]

Note, that neither the CREST formalism (i.e. the abstract syntax), nor CREST diagrams prescribe a specific definition language for update and influence functions. This makes the language flexible and adjustable to the knowledge of its users. For instance, in the example in Fig. 4 we use a mathematical notation for updates that uses δt to explicitly highlight timing behaviour in updates and use $value - 5 * \delta t$. However, due to CREST's flexibility it would be possible to adopt an ordinary differential equation notation by e.g. writing $value' = -5$ instead. One problem of such genericity is that in principle any kind of functions and notations can be used and defined, causing potential risks to the calculability of CREST systems. To restrict the situation, CREST's formalisation defines lim-

[5] See [50] for details of the syntactic and semantic definitions of these shortcuts.

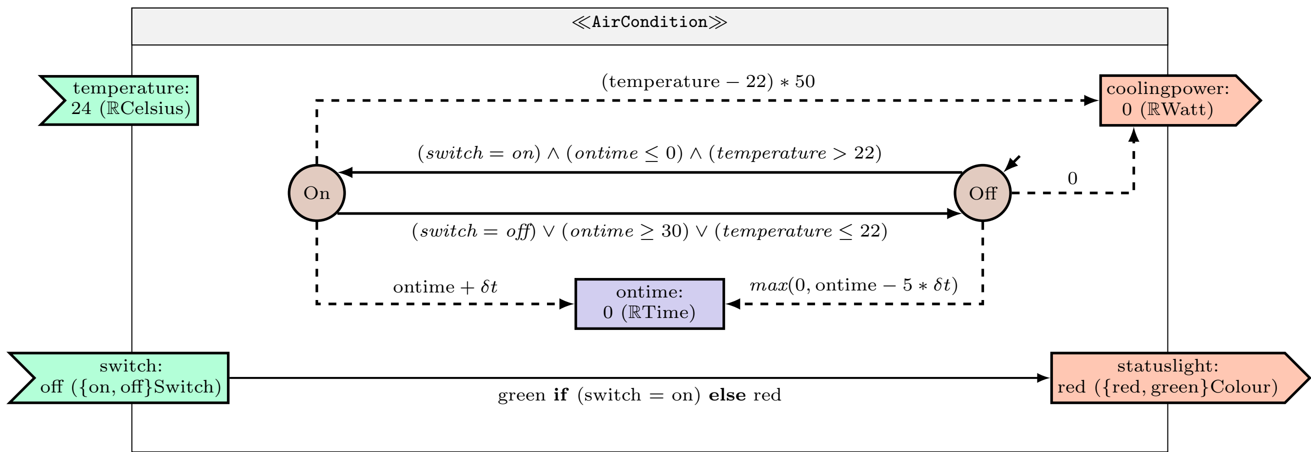

Fig. 4 An air conditioning unit modelled as a CREST diagram

itations by specifying the formal update/influence functions' signatures and the enforced function return value, to assert it matches the target port's value type. Additionally, there are limitations as to which ports' values can be used for the return value calculation, so as to not infringe upon CREST's locality principle.

Figure 4 depicts the CREST diagram of a simple air conditioning unit with an automatic timer. Its functionality depends largely on the values provided by the two input ports for temperature and switch, which influence the two output ports for cooling power and statuslight. Internally, the local port `ontime` measures how long the AC has been in the `On` state. Due to the shown abstraction level, the entire component can be modelled as a single entity, although we could imagine a refined version that uses sub-entities to e.g. model separate heating and cooling elements. Such a refined system would however exceed the purpose of this article.

The working principle of the AC unit is rather straightforward: Provided that the `temperature` input reads more than 22 degrees and the `switch` is on, the AC unit will run for 30 time units, or until the temperature or switch conditions are invalid. This is expressed using the condition $(switch = off) \wedge (ontime \geq 30) \wedge (temperature \leq 22)$. After a maximum of 30 time units, the device switches back to the `Off` state and remains there until the timer reaches zero again. The timer functionality is modelled using two updates that modify the ontime port. When in state `On`, the port's value is continuously increased according to the time that passes. This means, that the update calculates the result of $ontime + \delta t$ and writes that value to `ontime`. In state `Off`, the value is decreased towards zero by five times that rate (i.e. $\max(0, \text{ontime} - 5 * \delta t)$) until it reaches zero.[6] Two more updates are used to modify the `coolingpower` output, depending on the automaton state. When `Off`, the AC produces 0 Watt, when `On`, the power increases by 50 Watt for every degree over 22C. Finally, an influence is used to set the `statuslight`'s colour: green when the `switch` is on, red otherwise.

The CREST diagram in Fig. 4 uses mathematical notation for updates and transition guards, and pseudo-code for the influence's functionality specification. As introduced above, the notational form can be easily adapted to the modeller's knowledge and preferences. For instance, for more practical projects a programming language could be used specify each function's behaviour. This approach is followed in `crestdsl`, CREST's implementation in the Python GPPL, as introduced in the next section.

Note, that the ports of Fig. 4 display the initial values of the system (including its initial inputs). Though these values are theoretically part of the model's environmental embedding, we allow their definition. This (rather operational) view, is founded in the fact that similarly sensors often output a default value (e.g. 24C) or special value (e.g. *None*, *undefined*) before the first, actual reading.

3.4 `crestdsl`

In many cases, system developers prefer the use of textual over graphical languages. Even though graphical representations can be more easily understandable when used correctly [67], in recent years numerous tools were developed that allow textual modeling, followed by a conversion to a graphical representation.[7] Inspired by these tools, we aimed to create a textual DSL that is both simple to learn and write for programmers, but also offers a native conver-

[6] This function uses max to prevent `ontime`'s value from dropping below zero.

[7] See e.g. https://modeling-languages.com/text-uml-tools-complete-list for a comparison of text-to-UML tools.

sion to CREST diagrams for system developers who prefer graphical models.

Before developing the DSL, we evaluated the pros and cons of external DSLs against those of embedded languages and decided against the use of language workbenches. An often-used argument for the development of external DSLs is their integration with many existing products. Languages that are based on Xtext for example, promise to seamlessly integrate with many expert model transformation, model analysis and model execution platforms. In our project, however, the model translation to other formalisms and integration into meta-modeling frameworks are not primary goals and we do not expect large parts of our user base to require them for their purposes. Thus, the adaptation to new simulation engines and translation of models are not considered as primary requirements. The following sections will show however, that our approach is capable of interfacing with libraries and programs, which is beneficial for DSL-development on the one hand, but also allows experienced power-users to address potential advanced use cases on the other. Another reason to avoid the workbench-based DSL development method is our experience with the arising problems when it comes to the iterative development of DSLs and the problematic of changing grammars and their underlying meta-models. We therefore decided for the more lightweight approach and reuse a GPPL as the foundation for our language. As programming is nowadays widespread skill that is even taught at high-school level, we rely on the assumption that even non-professional and private CPS creators will have the familiarity and capacity to write simple scripting tasks. Thus the target users of `crestdsl` include professional and private CPS creators with (at least some) existing scripting and program development knowledge. They might for instance also include building managers that aim to increase the safety of their (small-scale) installations. Our method also allows us to reuse a wide-spread host language's syntax and semantics for the specification of transition guards, updates and influences. This approach is also well-known in modern data analysis and machine learning frameworks, that offer low-entry-barrier solutions for novices, but powerful programming APIs for advanced purposes.

`crestdsl` uses Python as host language for several reasons. First, it is easy to learn and use, and experiences growing popularity. This fact is also manifested in a large number of third-party libraries, numerous tutorials and a helpful online community. From a DSL development view, Python offers many attractive features such as a flexible meta-class system that allows customisation of object creation, function and class annotations (so-called *decorators*), and the availability of a powerful reflection API. Furthermore, many external tools such as SMT solvers, theorem provers and formal verification engines offer Python bindings and thus a native integration. The latter is a vital advantage for implementing the DSL and its execution engine.

During the development of `crestdsl` we paid attention to make its use as simple as possible, while at the same time support common Python development best practices. The software is available through the Python package index[8] and can be installed through Python's `pip` program. Thereafter, `crestdsl` can be used by importing, just as any other Python library.

`crestdsl` entities are modelled as instances of the `Entity` class. Features that belong to that entity (e.g. its ports, states, transitions, updates) are modelled as entity attributes. To simplify the creation of several similar entities, `crestdsl` provides *entity types*. These types are classes that inherit from the `Entity` class. The type's structure is then defined using class attributes and methods. Listing 1 shows an entity type that models the architectural parts of our air conditioning unit. It features two inputs, two outputs and two states. Note that just as CREST diagrams, `crestdsl` also requires every port to have a resource and current value specified. Similarly, every entity has to have one state designated as `current` state.

Listing 1 Definition of entity types via classes

```
1  from crestdsl.model import *
2
3  # Resources require a unit and domain
4  watt = Resource(unit="Watt",domain=REAL)
5  celsius = Resource("Celsius", INTEGER)
6  onoffSwitch = Resource("Switch", ["on","off"])
7  colour = Resource("Colour",["red","green"])
8  time = Resource("Time", REAL)
9
10 class AirCon(Entity):
11   temperature = Input(celsius, 24)
12   switch = Input(onoffSwitch, "off")
13   coolingpower = Output(watt, 0)
14   statuslight = Output(colour, "red")
15
16   on = State()
17   off = current = State()
18
19 my_new_ac = AirCon()
20 my_other_ac = AirCon()
```

8 https://pypi.org/project/crestdsl/.

Listing 2 An entity type with transitions and update functions

```
class DynamicAirCon(AirCon):
  # add the ontime port
  ontime = Local(time, 0)

  # A transition created by a transition object
  off_to_on = Transition(source=off, target=on,
     guard=(lambda self: self.temperature.value > 22
      and self.switch.value == "on" and self.ontime.
     value <= 0))

  # Another transition, this time using the
     decorator
  @transition(source=on, target=off)
  def on_to_off(self):
    return self.temperature.value <= 22 or self.
     switch.value == "off" or self.ontime.value >=
     30

  # update ontime
  ontime_when_on = Update(state=on, target=ontime,
     function=(lambda self, dt: self.ontime.value +
     dt))

  # An update via decorator
  @update(state=off, target=ontime)
  def ontime_when_off(self, dt):
    new_time = self.ontime.value - 5 * dt
    return max(0, new_time)

  # Adding an update as object
  cooling_when_off = Update(state=off, target=
     coolingpower, function=(lambda self, dt: 0))

  # An update via decorator
  @update(state=on, target=coolingpower)
  def cooling_when_on(self, dt):
    return (self.temperature.value - 22) * 50

  @influence(source=switch, target=statuslight)
  def light_influence(value):
    if value == "on":
      return "green"
    else:
      return "red"

dyn_ac = DynamicAirCon()
```

Upon instantiation, the `Entity` class's constructor and metaclass work to provide a correctly instantiated object. `crestdsl` supports class inheritance to create specialised or extended versions of an entity type. This feature is not a general CREST concept, but significantly increases `crestdsl`'s usability. Listing 2 for instance, shows the extension of `AirCon` by adding transitions, updates and an influence to the class. Note, that the listing shows two ways to specify transitions and updates. The first one is similar to the definition of ports and states: A `Transition` object is created and assigned as class attribute. The required transition guard is specified either as a lambda-expression or using a function. Alternatively, it is possible to define the transition guard as a class method. By using the `@transition` decorator, it is then possible to convert the method into a transition. The listing also shows the specification of updates via `Update` object and `@update` decorator, and the creation of an influence using `@influence`.

Class inheritance is a powerful concept, as it can also be used to adjust entity behaviour. Python's multiple inheritance functionality allows class fragments (a.k.a. *mixins*) to be added to the language. This enables modular development and the separation of specific concerns (e.g. reusable component fragments for adding value inspection or signal probing) into individual classes, that can then be reused where required. If used correctly, this feature permits advanced users to create reusable component fragments and thereby even add aspect-oriented system design to the DSL. The inheritance design principle invites code reuse and the creation of object libraries. For this purpose, `crestdsl` also provides parametrisable entity types using class constructors.

Without the provision of a separate example, we want to highlight that system composition follows the same DSL design, where subentities are defined by assigning entity objects as class attributes.

To test whether the system model was correctly assembled, a `SystemCheck` class can be used. This static system analysis performs a number of sanity checks such as testing the entity hierarchy and asserting that each entity defines a current state. Its purpose is to help `crestdsl` users debug their model quickly and find potential causes for misbehaviour. If necessary, users can also extend the class and implement their own static analysis routines that assert certain system setups (e.g. testing for specific architecture setups, port type conformance, etc.).

4 Simulation and verification

Usually, model creation serves as stepping stone for further, advanced modeling tasks such as model-based analysis, simulation and verification. Above, we introduced `SystemCheck` as an example for simple, static analysis. Due to its formally defined semantics, CREST also supports sound simulation and verification.

4.1 CREST semantics

To support the combination of aspects from several formalisms (FSM, architecture descriptions, hybrid systems), it is necessary to pay careful attention to the precise meaning of each element. For this reason, CREST's semantics have been formally defined to assert consistency with the key principles that we obtained in the requirements evaluation. Thus, it is paramount, that the operational semantics uphold the reactivity, synchronism, parallelism, locality, continuity and non-determinism that are essential for the expression of

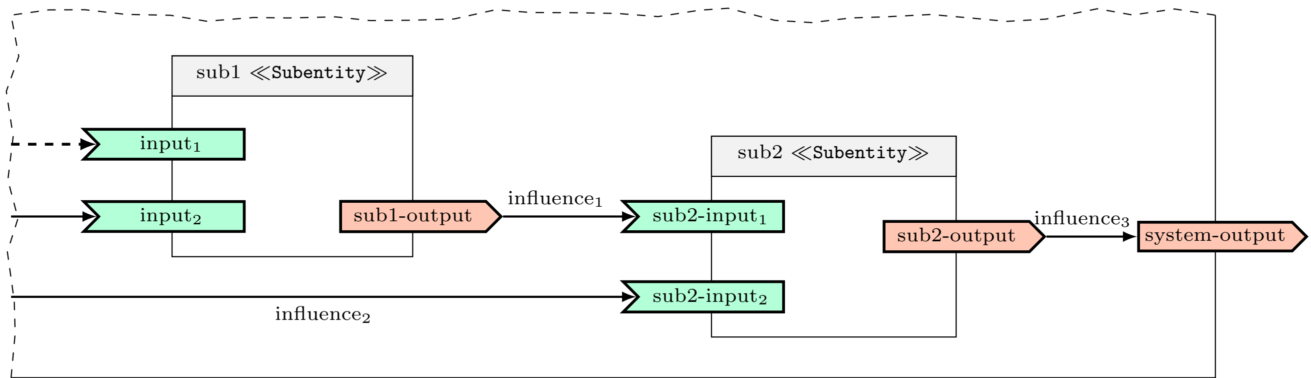

Fig. 5 An excerpt of a CREST diagram highlighting dependencies between subentities. Note, that for clarity the port types and values were omitted

our systems. In this section, we outline CREST's semantics and highlight the relation to other languages and formalisms, such as data-flow languages and hybrid automata. Note, that this section focuses on providing a high-level description of the semantics and its effects in the simulation of CREST. The actual formalisation, including its structured operational semantics (SOS) rules, is provided in Appendix 1. A more elaborate description, that exceeds the scope of this publication can be found in [50].

CREST foresees two ways of model interaction. First, the change of the root entity's input port values, and second, the advance of time. After each of these, the model has to be *stabilised* until a "fixpoint" is reached. Fixpoints are system states where the model is stable and will not change without another interaction.

Input value change The external modification of input port values represents changes in the system's environment to which it should react. Thus, such a modification can only occur at the hierarchical top level. This is due to CREST's locality principle, which requires every entity—hence also the root—to be treated as coherent black box that only exposes its communication interface (input and output ports). Once a port value change is observed, the semantics require the stabilisation to be triggered, which then propagates the updated information throughout the system. The process starts by performing the stabilisation first on the system's root-entity, which then recursively calls stabilisation on its subentities. Evidently, *modifiers* (i.e. influences, updates and subentities) within an entity can depend on each other. For instance, Fig. 5 shows an excerpt of such a CREST system. It is clear, that `sub2` depends on the output value of `sub1`. Thus, to assert a correct system state, it is necessary to first make sure that `sub1` has been stabilised and reached a fixpoint, before executing $\texttt{influence}_1$ to propagate the value of `sub1-output` to $\texttt{sub2-input}_1$.

This brief example illustrates the need to establish a modifier execution order. The creation of this execution order

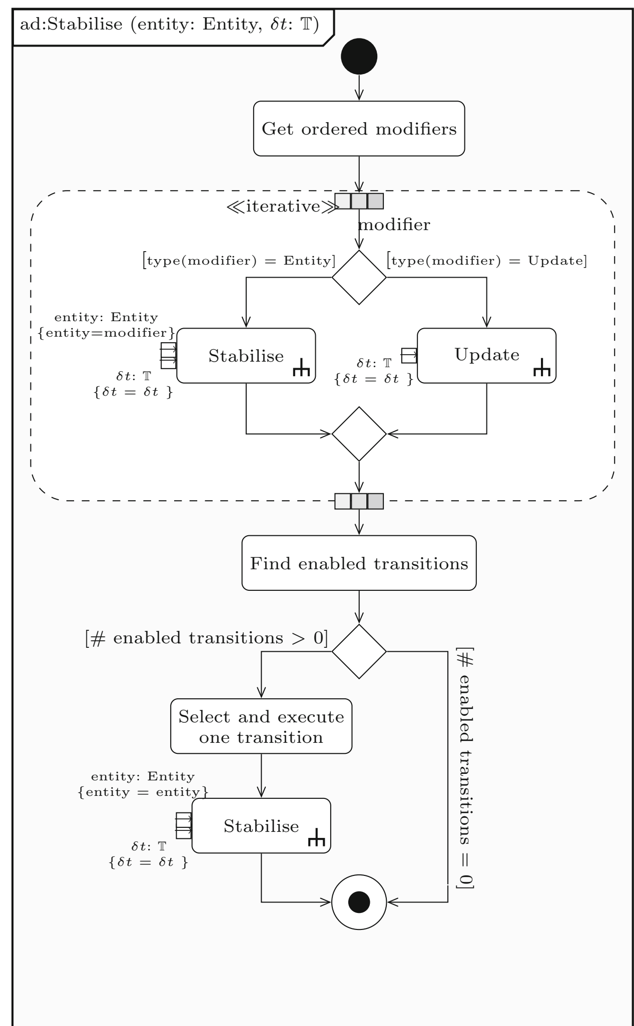

Fig. 6 Schematic representation of the stabilisation semantics as UML 2.0 Activity diagram

is closely related to the use of Kahn process networks [49] and commonly used for the execution of data-flow languages such as e.g. Lustre [42]. The ordered modifiers are then exe-

cuted one by one. This means that in this order, each update function is evaluated and the result written to its target port, and subentities are recursively stabilised (see Fig. 6).

Only after all modifiers have been executed, the transitions from the current state are tested whether they are enabled. If no transition can be triggered, the process ends and the entity is stabilised. Otherwise, one transition is chosen and executed. CREST does not prescribe a resolution strategy in case multiple transitions are enabled at the same time. This means, that any enabled transition can be triggered, which is CREST's way of introducing non-deterministic behaviour—a vital concept and one of CREST's key aspects. After a transition is executed, the stabilisation process has to recursively call itself again, to assert that updates that are related to the new automaton state—and are thus newly active—are executed.

Note, that the semantics in Fig. 6 do not mention influences and actions, as these are syntactic sugar and can be expressed through updates and states alone, as described in Sect. 3. This trait both simplifies the formal semantics and CREST's system analysis.

It is also of interest to mention that generally, the reaching of a fixpoint cannot be guaranteed. Especially in the presence of Zeno behaviour (i.e. an infinite number of transitions in finite time [82]) or cycles of continuously enabled transitions it can happen that infinitely many actions are executed without ever reaching a stable state. Even though CREST's implementation provides a few configurable heuristics to detect these kinds of situations, it is up to the system designer to avoid the modeling of such problematic behaviour.

Advancing time The notion of time is an important aspect of CREST. It allows the modeling of continuous effects, such as the filling of a water tank without the need of introducing artificial discretisation. Since CREST implements *must* semantics (as known from e.g. hybrid automata), this introduces an interesting challenge, as it is necessary to find the precise moment in time when a transition becomes enabled.

For instance, when looking at the time-based behaviour of the air condition, we observe that—provided the input values do not change—the model will alternate between on and off states, so that it is 30 time units in state `on`, followed by 6 time units in state `off`, before switching to `on` again. Figure 7 shows a trace of this behaviour. Evidently, when simulating continuous time systems, it is important to choose the right step-size for the current system behaviour. Advancing in too small time intervals might cause a high calculation overhead and perhaps even render the simulator unusable. Too coarse step sizes can lead to "missing" an important point in time, which either requires the simulator to step back and retry with a smaller step size, or, if undiscovered, might even lead to wrong simulation results. CREST's semantics assume the existence of a *next transition time* function, that calculates

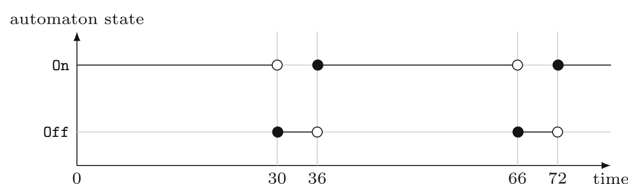

Fig. 7 A trace showing the air condition model's automaton state over time

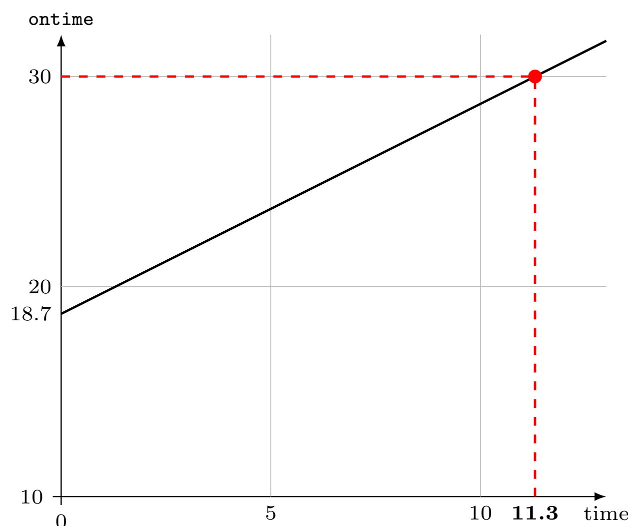

Fig. 8 Locating the next transition time, assuming that `ontime` = 18.7 at the beginning

the precise amount of time until a behaviour change (i.e. a state transition) occurs. This means, that calling the function when the air condition system is in state `on` and `ontime` has a value of 18.7, should return the exact value of 11.3 time units, since at this time the port's value reaches 30 and thus a transition should be triggered (see Fig. 8).

`crestdsl` implements *next transition time* using an SMT approach. Whenever it is necessary to calculate the time until a transition event, the simulation engine translates the update functions and influences that potentially affect the value of a transition guard into a set of SMT constraints (see details in the next section). This constraint set is then handed to Microsoft's Z3 Theorem Prover [22], which either calculates a value for δt that solves these constraints and hence enables the transition, or alternatively signals that the constraint set cannot be solved and therefore the transition cannot be enabled by the advance of time alone.

Once the simulator obtains the knowledge at what point in time the next transition becomes enabled, advancing time is merely a matter of iteratively stepping forward to the next transition point and triggering the stabilisation process after every step. Note, that in this case the stabilisation process has to trigger updates with the correct δt-value, to consider the

timing behaviour when calculating the updates' target port values.

CREST's formal semantics (see Appendix 1) do not specify how the next transition time should be calculated. Thus, CREST can be used for any type of system. The implementation's SMT approach however relies on the use of Z3, which is not capable of solving nonlinear optimisation problems. This means, that even though `crestdsl` produces a nonlinear constraint set, it cannot be solved and the simulation is likely to run into issues in the presence of nonlinear dynamics. At the moment, we are aiming to extend `crestdsl` to add support for nonlinear systems by using theorem provers with nonlinear optimisation capabilities such as *dReal* [37].

Next Transition Time Calculation

As stated above, CREST's semantics require the calculation of the nearest point in time when a transition will occur. `crestdsl` implements this functionality by creating a constraint set for each transition, which expresses the value change of its guard condition over time. These constraints are then handed to the SMT solver to find a minimal solution to these constraints.

The challenge of this approach is the creation of a constraint set that captures the dependencies between the ports. For instance, we see that the input `sub2-input`$_1$ of Fig. 5 is influenced by `sub1-output`, whose value might in turn depend on other ports. `crestdsl` automatically converts these modifier-dependencies into constraint sets, so that any time-based updates of predecessor-ports are also considered, as these might indirectly enable transitions.

Listing 3 The `on_to_off` transition

```
1 @transition(source=on, target=off)
2 def on_to_off(self):
3   return self.temperature.value <= 22 or self.
      switch.value == "off" or self.ontime.value >=
      30
```

Listing 3 shows one of the air condition's transitions. It states that the device should turn off when either the `switch` is off, `ontime` reaches 30 or the `temperature` drops below 22 degrees. Thus, the first constraint c_1 captures the guard condition.

$$c_1 := (temperature < 22) \lor (switch = \text{“}off''') \\ \lor (ontime >= 30)$$

From the CREST diagram in Fig. 4, we see that the three sub-expressions are based on the values of two input ports (`switch` and `temperature`) and one local port (`ontime`). Since `AirCondition` is the root entity, its input port values cannot be changed by any system modifier (neither by updates, nor influences, nor subentities). `ontime` on the other hand is continuously modified by the `ontime_when_on` update function. It is therefore necessary to add its functionality to the constraint set. From the update function's source code `lambda self, dt: self.ontime.value + dt` we can see that when executing, the function reads the current ontimer value and adds `dt`, i.e. the amount of time since the update's last execution, to it. Therefore, the next constraint to add is

$$c_2 := ontime = ontime_0 + \delta t$$

where $ontime_0$ and $ontime$ are the port's values before and after `ontime_when_on`'s execution, respectively.

As the air condition has no other dependencies, the system input ports have to be linked to their current port values, since otherwise the SMT solver might assign any value that solve the constraint. Our analysis aims at time-based changes alone. We thus have to set the values of these ports, so that δt is the only assignable variable in the constraint set. The constraints c_3 to c_5 express this by linking *temperature*, *switch* and $ontime_0$ to the values shown in Fig. 4. c_6 asserts that δt is positive. The final constraint set contains all equations necessary to discover when the `on_to_off` transition becomes enabled:

$$\begin{aligned} c_3 &:= switch = \text{“}on'' \\ c_4 &:= temperature = 24 \\ c_5 &:= ontime_0 = 0 \\ c_6 &:= \delta t \geq 0 \end{aligned}$$

When handing these six constraints to an SMT solver, it will not directly return the nearest transition time, as any $\delta t \geq 30$ is a solution to the problem. Therefore, we use Z3's *optimization* functionality to find a minimal δt, which in the example above is 30.

4.2 `crestdsl` simulation

Using CREST's formal semantics and `crestdsl` it is possible to create a simulation engine that helps answering "*What happens if…?*" questions. Users can try out system interactions and observe possible usage scenarios. Listing 4 shows the example code of a simulation scenario. The listing uses the standard `Simulator` that follows CREST's semantics and implements non-determinism. However, `crestdsl` provides two other simulation engines that allow slightly altered usage. The first one, `InteractiveSimulator`, prompts the user to choose an enabled transition anytime a non-deterministic situation is encountered. This allows users to explore specific scenarios despite non-deterministic models. Finally, `PlanSimulator` can be used to define strategies on how to deal with non-determinism. This sce-

Listing 4 `Simulator` example

```
from crestdsl.simulation import Simulator
system = DynamicAirCon() # create system
sim = Simulator(system) # init simulator
sim.stabilise()  # automaton state is off

system.switch.value = "on" # modify input
sim.stabilise() # state: on
sim.advance(10) # ontime: 10
sim.advance(20) # ontime: 30, state: off
```

nario can be used to model controllers that, given a range of similar transition choices, will choose one according to a pre-defined policy. The `PlanSimulator` can also be used to follow a specific execution trace that is e.g. discovered by state space exploration or similar verification techniques, as described further below.

4.3 Verification

CREST's semantics allow the construction of state spaces and the use of formal verification techniques, such as model checking, thereon. Based on existing approaches, logic formulas can be defined and their truth-value examined on a model's state space. Since CREST is a hybrid formalism, it is however necessary to extend the classical temporal logics (e.g. LTL, CTL) to incorporate timing aspects in their language. Given CREST's non-determinism, and thus its branching behaviour, we use the timed computation tree logic (TCTL) [45], a timed extension of CTL, and its operators and formulas for the specification of system behaviour. We will not elaborate on the foundations of classic and temporal model checking, but instead refer the interested reader to seminal works such as [3] and [14]. In general, TCTL formulas are composed of a set of operators, which are inductively defined:

$$\phi = \texttt{True} \mid p \mid \neg\,\phi \mid \phi \wedge \phi \mid \phi EU_I\ \phi \mid \phi AU_I\ \phi$$

where $p \in AP$ is an atomic proposition, and I is an interval in CREST's time base that defines the timing aspect of the EU_I and AU_I operators. Based on these operators, further abbreviation operators can be defined, such as $EF_I\,\phi = \texttt{True}\ EU_I\,\phi$ and $AG_I\,\phi = \neg\,EF_I\,\neg\,\phi$. Note, that in comparison to classical CTL, there is no X ("next") operator, since in continuous formalisms there is no "next point in time".

Subsequently, we can test these formulas on timed Kripke structures [57]. Timed Kripke structures are graph-based encodings of a system's state space, whose nodes represent discrete system states and edges are timed transitions between them. Each node is annotated with a set of APs

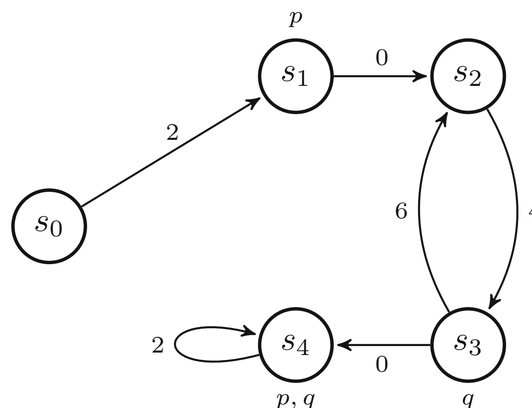

Fig. 9 A timed Kripke structure with five system states $S = \{s_0, \ldots, s_4\}$. Nodes labels are the APs $\{p, q\}$

that represent certain behaviour (e.g. "the AC is `on`" or "the `coolingpower` output is above 100 Watt"). CREST's continuous semantics demand that the transitions be annotated with the amount of time it takes to pass from one state to another. Figure 9 shows an example of a timed Kripke structure.

Following some minor adaptation of the timed Kripke structure, the TCTL formulas can then be evaluated using graph search algorithms that are inspired by classical CTL model checking approaches. Details on these algorithms are provided in [57] and an elaborated description of the creation and adaptation of timed Kripke structures for CREST models is described in [50].

4.4 `crestdsl` verification

`crestdsl` implements a Python-native `verification` subpackage that provides APIs for the specification of system configurations, the creation of a model's state space and the search of that behaviour within that state space. System states are specified as *checks*, which can be used to either define that an entity is in a given automaton state (e.g. `check(entity) == entity.on`), or to compare a port against a constant or another port value (e.g. `check(model.coolingpower) > 200`). `crestdsl` also allows to create composed checks as conjunctions, disjunctions and negations of other checks.

Checks can then either be evaluated directly on a system state using by calling its `check()` method. Alternatively, they can be used for the formal verification of a `crestdsl` model's state space. To do so, `crestdsl` provides two APIs. The first is `crestdsl`'s *convenience* verification API. It provides interfaces for users that are unfamiliar with model checking and formal verification. Modellers are provided with a set of functions that can be executed for common

Table 2 Verification API and TCTL equivalents

crestdsl Function	TCTL Formula
is_possible(chk)	$EF\ chk$
always(chk)	$AG\ chk$
always_possible(chk)	$AG\ AF\ chk$
always_possible(chk, within=time)	$AG\ AF_{[0,time]}\ chk$
never(chk)	$AG\ \neg chk$
forever(chk)	$EG\ chk$

where *chk* is a crestdsl check and time is numeric

verification tasks. For instance, the always function, verifies that a given condition can never be invalid. Similarly, is_possible asserts that it is possible to reach a given state and never asserts that a given system configuration can never be reached. Table 2 shows some convenience API functions and their TCTL equivalent formulas, Listing 5 shows their programmatic usage.

Listing 5 Model checking of crestdsl checks.

```
1  from crestdsl.verification import check,
       is_possible, always, before
2
3  system = DynamicAirCon()
4  chk = check(system.ontime) == 25
5
6  # model checking
7  is_possible(chk).check() # True
8  always(chk).check() # False
```

crestdsl also implements an *expert* API, that allows to manually create complex TCTL formulas and obtain direct control over the model checking process. By directly creating and using the StateSpace object, it is possible to gradually explore a state space. This allows crestdsl's routines to be used in some infinite state spaces. The tctl API groups implementations of all TCTL operators, so that experienced modellers can create complex formulas to be evaluated. Listing 6 provides an example of the expert verification API.

Listing 6 Examples of tctl formula definitions.

```
1  from crestdsl.verification import tctl, check,
       StateSpace, ModelChecker
2
3  # define a system and check
4  system = DynamicAirCon()
5  chk = check(system) == system.on
6  before_30 = tctl.EF(chk, tctl.Interval(end=30))
7
8  # explore state space and model checking
9  statespace = StateSpace(system)
10 statespace.explore()
11 mc = ModelChecker(statespace)
12 mc.check(before_30)
```

5 Discussion

DSL Development Process Our experiences throughout the CREST project were mostly positive. Although our aim to cut the development effort for the creation of a new modeling formalism drove us to reduce some of the typical modeling overhead (e.g. syntax definition), we still had to invest into defining a coherent language semantics. Furthermore, our internal discussions often resulted in debates about the tradeoff between the need for the creation of independent, dedicated modeling concepts and the effort of adopting well-known aspects from other languages. For instance, there are many different types of component composition described in research, and indeed there is evidently a possibility to invent new ones if needed. However, our reusage goal led us to adapt the hierarchical modeling aspect known from architecture description languages (ADLs) and their reported experiences in using ports for composition. Similarly, we managed to reuse abstractions and concepts from hybrid automata, synchronous data-flow languages, and combined visual and textual modeling approaches within the same language. Their joint use in CREST is based on a formalisation that thoroughly combines the syntactic and semantic aspects of the individual language parts. The success of the SystemC hardware description language inspired us to implement CREST as internal DSL, so we can rely on existing IDEs, execution environments and programming aids.

Indeed, by implementing crestdsl in Python, we quickly noticed the advantages of its large developer community. It also allows us to integrate the language with third-party developments, such as the Project Jupyter[9] runtime. This interactive platform allows code to be defined in so-called *cells* that are executed at command, inviting iterative and fine-grained development approaches and thereby increasing usability and user experience. Jupyter's user interface is browser-based, can be easily installed via Docker images or hosted on external servers. Cells are housed in *notebooks* and also provide a means to write text in e.g. using Markdown format right next to executable code, which merges documentation and executable code in one document.

Through our integration of crestdsl into Jupyter, models can be plotted directly using a combination of HTML and JavaScript code. This results in an interactive output (see Fig. 10) that can be explored, zoomed in and out and interacted with. Through popups and mouse-over events, it is possible to access and view live data from the model, as well as to manipulate the display itself (e.g. move and resize entities).

Python's many third-party-libraries can also be used for other aspects of CREST. For instance, it is possible to use the

[9] https://jupyter.org.

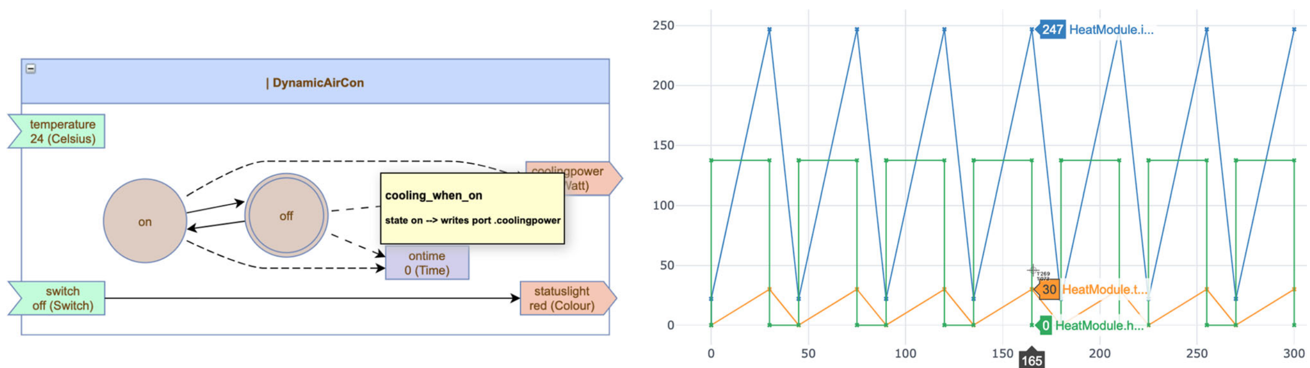

Fig. 10 Interactive CREST diagram plot and simulation trace of a `crestdsl` model

Pandas[10] data analysis package to query and analyse system traces, or the Plotly[11] library to create an interactive representation thereof, as shown in Fig. 10. We invite readers to visit the `crestdsl` webpage at https://crestdsl.readthedocs.io/ for more details on our work and `crestdsl`'s online documentation. You can also find a demo that can be launched online inside your browser at https://github.com/crestdsl/sosym-aircondition.

Translation between Languages We found a major advantage of CREST to be the combination of both, the textual and graphical modeling environment. `crestdsl`'s module for the automatic translation of its models to interactive CREST diagrams proved to be a useful commodity at model creation time. As a result, developers create a system model using an efficient textual programming style, but also benefit from the perks of a visual language for discussion and brainstorming. Additionally, as the languages share the same abstract syntax and operational semantics (see Appendix 1), it is possible to first draw a graphical diagram model and then implement it in `crestdsl` for simulation and verification. At the moment, our tool implementation only supports the textual-to-graphical translation though, as we initially believed that our users' workflow would typically start in `crestdsl` and move to CREST diagrams for enabling visual inspection and discussion. However, we noticed that our users often "think in CREST diagrams" and actually start the modeling process by sketching out system models as CREST diagrams using pen and paper, before switching to implementing individual components in `crestdsl`. This insight is an important insight and big encouragement for the creation of a parser that enables automatic translation from CREST diagrams to `crestdsl` models, to enable creation of an outline in CREST diagrams and switching to `crestdsl` for the detailed model creation. We are currently in the process of evaluating the requirements to such an interactive graphical development environment and evaluating likely development workflows to find out which tasks can be more efficiently executed in which syntax. This analysis should also help us answer the question for which phases the user will switch between languages. A point of concern is that a translation from CREST diagrams to `crestdsl` necessitates the parsing of transition guards, update functions, etc, whose syntax is currently not limited by CREST. Thus, a graphical-to-textual translation mandates a limitation of the supported syntax, which could for instance be Python for integration with `crestdsl` or some dedicated, embedded DSL. Though the benefits of this extension appear evident, we consider a thorough evaluation of the complete, required feature scope (e.g. a graphical editor, parser, etc) and the analysis of development and maintenance costs for the (interactive) bidirectional translation as future work.

User feedback Since our project aims at the use by non-expert modellers, throughout the project we sought the input of people from outside the modeling research domain. Given our academic background, we therefore asked our students (Bachelor's and Master's level) to use CREST and `crestdsl`, in order to gather feedback and improve the usability. The conclusions drawn from these evaluations are largely positive and show that both, the CREST formalism as well as the `crestdsl` implementation can be easily learned. Even though the users neither had a background in modeling nor any training in CPS design, they rapidly acquired the necessary knowledge to model our case study systems and were even able to create object libraries.

These experience reports provide us with valuable feedback and confirm that our method is promising, although we admit the necessity of a more formal evaluation. Thus, we are currently in the process of designing a DSL usability study based on the principles found in works such as [5] and [6].

Lessons learned Throughout the project, we relied on our experience with semantic definitions and language for-

[10] https://pandas.pydata.org.

[11] https://plot.ly/python/.

malisation. Our methodology is based on the thorough formal definition of all required syntactic and semantic concepts before their implementation in CREST diagrams and `crestdsl`. This increased our trust in the DSL's correctness and helped us identify potential problems that we otherwise may have overlooked. We see this part of our method as vital for the success of this project and a creation of a usable language. It is still important to clarify that the formal definition did not follow a "one-shot" approach, as we had to go through several iterations before all required features were implemented. Compared to other approaches to DSL creation by reuse of existing design concepts (see Sect. 6), our approach proved to be dynamic and flexible, as our focus was not placed on a direct combination of existing formalisms and languages. Instead, our goal was to increase CREST's usability as much as possible and, although we aimed to maintain "the spirit" of the original formalisms (e.g. automata, ADLs), we valued a concise product over their direct reuse. Our design method helped us to iteratively select the best concept from various options. For instance, we chose state automata for the definition of entity modes over a Petri net-based approach, which would have simplified the modeling of concurrent behaviour and certain parts of the semantics, but increased complexity and reduced intuitive usage. Similarly, we evaluated various forms of component composition and different formalisations of ports that exist in literature. Nonetheless, it is important mentioning that our approach is not meant for algorithmic or (semi-)automatic DSL composition. It appears that such a composition is in many cases opposing the coherent integration of language features. Resulting language compositions typically require adapters (or some form of "glue") to enable the interaction between the domains of the existing languages [41].

The availability of the three case study systems also facilitated DSL design, increased development speed and helped us identify which language aspects should be implemented. One noteworthy point is that even after identifying the DSL's requirements and necessary system features, the formalisation caused discussions about tradeoffs. For instance, as mentioned above, CREST's influences and actions are treated as syntactic sugar and translated to their corresponding update representations. Before this definition, we initially developed an equivalent semantics that treated the former separately. It was only after another iteration and application study, that we decided to simplify the operational semantics to their final state.

Tool Implementation A more practical problem arises when it comes to the use of `crestdsl`. Even though the language offers much flexibility, one caveat of using an internal DSL is that the available host language syntax is not limited. This can lead to problematic situations, as the user does not receive any feedback when they try to use illegal concepts. For instance, in `crestdsl` updates and influences are not allowed to model nonlinear behaviour, as the simulation engine relies on the use of Microsoft's Z3 Theorem Prover which is restricted in that sense. An extension to a more powerful SMT solver (e.g. dReal [37]) could greatly increase the applicability of `crestdsl`. In a similar way, modulo operations are not supported and the use of string datatypes is limited. If modellers use these constructs in their system models, however, they will not be informed of their wrongdoing until the simulation or an explicit system check is triggered. This delay in feedback can easily lead to frustration. `crestdsl` provides a `SystemCheck` class that aims to discover and inform the user of illegal models using reflection APIs. Nonetheless, users have to explicitly execute this functionality for feedback. We see this behaviour as less powerful when comparing to external DSLs, which tend to have an explicit grammar and abstract syntax tree (AST) checks built into editors and IDEs and provide imminent feedback during model creation and editing.

When developing internal DSLs, it is also important to invest into properly informing users of the activity that is performed in the background. Many times certain modeling tasks need a significant amount of time to perform, and thus, users require feedback when calculations are performed in the background. For instance, during model checking the creation of a state space and the search of the resulting timed Kripke structure take a long time. Here, it is especially important to provide the user with feedback about the exploration state.

Target systems We found that, as intended, our DSL provides a convenient means for the modeling of smaller systems. However, we noticed that CREST can also be used to elegantly describe larger, more complex systems, at higher abstraction levels, although its `crestdsl` implementation quickly reaches acceptable performance limits for simulation and verification times in its current implementation. For instance, some of our benchmark implementations modelled the above mentioned case study systems at various levels of detail. Though these studies are preliminary and warrant for further exploration, they show that `crestdsl` can be effectively used for the modeling and simulation of systems with 50 to 70 non-trivial components. As another benchmark, we included a linear approximation of the heating process inside a water boiler into one of these systems. The component's linearisation splits the water volume into several interconnected "heating zones"[12] which expose mutual temperature influences among each other. Though `crestdsl` has not been implemented for modeling of such complex nonlinear processes or their linear approximations, the system shows

[12] See https://github.com/crestdsl/case-studies/tree/master/SmartHome for details.

that for small numbers of heating zones (up to 7), the system can provide valid simulation results in acceptable performance. Nonetheless, for larger numbers of zones (and thus finer model granularity), `crestdsl`'s performance exceeds acceptable limits. Similarly, due to the large state space an efficient verification of such systems is not efficiently possible at the moment.

CREST joins the architectural and behaviour system aspects into one, coherent language. The creation of models as a team of multiple modellers is a necessity for large projects, but has not been foreseen in CREST and certainly poses inconvenience. Similarly, the use of Python as a host language leads to significant performance bottlenecks that cannot be resolved easily. Especially when it comes to verification of complex TCTL formulas on large model spaces, for example, Python and `crestdsl` reach their limits.

6 Related works

The approach presented in this paper is related to several current research directions. Next to the generic background work described in Sect. 2, we aim to highlight several works that tackle similar gaps as our research, namely the modeling of CPSs using DSLs on the one hand, and creation of new DSLs through combination and integration of aspects and design of existing ones on the other.

In the recent past, the employment of DSLs in modeling has become a widely popular means to manage complexity, especially in elaborated domains. Nordmann et al. [61] survey the vast landscape of DSL use in robotics, a CPS domain closely related to ours. To not exceed the scope of this article, we focus in this section on research that is closely related to our work, namely the creation of internal DSLs. Notable advances in this subdomain includes the work of De Laet et al. [21], which provides insights into the differences of a DSL that was both implemented internally (in Prolog) and externally using the Xtext framework. In their discussion, the authors note amongst other points the enhanced readability of the internal DSL. Functional programming languages pose a popular choice for host environments due to their readability, as shown in [26,46,66]. The authors of the former use F# to implement a DSL for the synthesis of programs by a learning system. Readability is of vital interest, as humans are required to verify the generated programs. Recently, Sadati et al. [73] created an internal DSL within Matlab that is based on *fluent interfaces*.[13] This research is of particular interest, as the authors' motivation lies in the experience that users are hesitant of employing new tools and prefer using familiar development environments—an observation that similarly drove the design of `crestdsl`. Their publication describes the efficient use of Matlab's specific language features and also highlight some unavoidable caveats introduced by the Matlab language syntax. Another internal DSL that uses similar concepts to `crestdsl` is presented in Fryer and McKee [35]. Their language is implemented in C++ and —similar to `crestdsl`—uses `@annotations` to specify additional model semantics and to extend the model class definitions.

Generally, it appears that though other languages and (internal) DSMLs integrate existing modeling formalisms and languages to extend their capabilities (see also e.g. MontiArcAutomaton [72], AADL's Behavioral Annex [29]), it appears that these languages' target audience remains to be modeling experts, such that their usage is often very difficult for newcomers or non-experts (see Sect. 3.1). On the other hand, various DSLs exist e.g. in the domain of "smart" end-user electronics that allow the simplified assembly of systems and creation of workflows using pre-defined building blocks and object libraries.[14] These languages, often created by the gadgets' manufacturers are an easy means to system configuration, but often closed-source and limited in capabilities and without clear syntax and semantics. As a result, users cannot easily—often not at all—extend the language, define custom behaviour or add own components. In comparison, CREST aims to fill the gap between these two sides by providing an open, versatile approach that allows the precise definition of system behaviour, model simulation and formal verification, while still aiming for ease of use. Thus CREST features the reuse of formally sound concepts (e.g. automata, ports) that are well-known by expert system creators, and combines them to form a coherent language that is also easy to learn and comprehend by novice users and private smarthome creators.

The domain of reusing and combining various aspects of modeling languages has similarly been approached using different methods. Lacoste et al. [56] describe the creation of a hybrid language by combining the Event Scheduling formalism with ODEs. They implement their language in the ATOM^3 platform, which is the basis of creating a graphic editor. Similarly [7] describes the syntactic and semantic composition of DSLs for testing purposes.

Mustafiz et al. [60] follow a similar research, combining Timed Finite State Automata and Causal Block Diagrams to a hybrid language. Their approach is novel in that it relies on the extraction of language specification fragments (LSF), which represent modular, formal extracts of language features that allow the flexible combination on syntactic and semantic level. They then combine these LSFs by embedding one into another. In comparison, CREST operates on a more integrated level. Instead of extracting individual lan-

[13] https://martinfowler.com/bliki/FluentInterface.html.

[14] Amazon Alexa "Skills" or Google Assistant "Actions" are two examples of workflow languages that can automatically activate smart devices according to predefined rules.

guage features as LSFs, it combines syntactic and semantic concepts of different languages coherently. Mustafiz et al.'s approach on the other hand it is possible to distinguish the features that belong to each LSF, and the newly added concepts that "glue" the LSFs together.

Modularity and reuse of language design are also treated from a Software Product Line (SPL) methodology (e.g. [47]), where DSLs can be dynamically created based on product line models such as feature diagrams. A similar approach is followed by the Neverlang project [19], which aims to capture language features in so-called *slices* (parts of syntax and semantics), that can easily be combined. Compared to the SPL method, Neverlang aims to provide flexibility throughout the evolution of a DSLs and its components. These approaches are very powerful when it comes to the creation and adaptation of DSLs, as individual language features and slices can be easily added or removed to achieve a high level of customisability. Nonetheless, for CREST's purposes the initial cost of extracting such features and slices for later combination is too high, especially when taking into consideration that we do not have any concrete plans for the creation of a family of similar DSLs.

7 Future directions

The development of CREST not only provided valuable insights into the creation of DSLs from existing formalisms and hands-on experience in the development of internal DSLs, but also resulted in a powerful basis for continued research in this area. So far, we identified several directions in which we want to extend our method.

CREST Improvements and Extensions CREST's current state can be adapted and extended in several ways. First, we noticed certain difficulties when it comes to the verification of models with large state spaces. Due to the use of Python as a host language, we attribute a lot of performance loss to the interpreted execution environment. To overcome the performance bottleneck, we are investigating several possibilities. On the one hand, we might translate CREST models into other formalisms (e.g. hybrid automata, Petri nets, DEVS) for which performant tools already exist. Alternatively, we are looking into adapting verification heuristics (e.g. symbolic model checking) to CREST's specific semantics, in order to reduce the state space size.

Another project we are working on aims to use CREST in concert with modern machine learning and artificial intelligence approaches. For instance, we are working actively on a means to allow CREST systems to synthesise the behaviour of individual update functions and transition guards. Initial results of this research have already been presented at dedicated workshops [51].

We are also looking into the use of reinforcement learning approaches for the automated generation of system controllers for CREST models. The aim is to create software models of controller entities that will provide valid system inputs and assert that beneficial behaviour is enforced and unfavourable behaviour avoided. Usually a manual controller creation process is time-intensive and error-prone, and highly complex for hybrid systems specifically. Thus, automated generation techniques will allow to strongly improve this system engineering task.

Embedding a Behaviour DSLs It appears, that for some user groups (e.g. non-programmers) the implicit restrictions on the Python code that can be used inside transition guards and update functions outweigh the advantages of using an internal DSL. While a completely external DSL would certainly solve this issue, one of our future developments investigates the embedding of a dedicated DSL into `crestdsl`. For instance, it would be possible to use the `crestdsl` to define the basic system structure, but enforce another (restricted) DSL for the specification of transition guards and behaviour functions. This way, it is more easily possible to guide the users' behaviour and avoid frustration by incomprehensible statements. In the background, the embedded behaviour DSL could then be transpiled to Python code to rely on the existing language implementation.

Similar DSLs We intentionally designed CREST and `crestdsl` for a specific type of CPSs. In these systems, each component has one behaviour and parallel activities are separated in individual subentities. However, there are numerous industrial installations where the component behaviour is highly dynamic and dependent on many factors. Petri nets have been successfully used to model parallel and concurrent behaviour. Currently, we are investigating an adapted version of CREST, where the entity automata are replaced with Petri nets. Evidently, this modification also requires an alteration of the operational semantics, such as an adaptation of the synchronism concern. Similarly, other system requirements demand different extensions. Digital communication can be introduced by reusing concepts from e.g. communication diagrams.

8 Summary

CREST is a hybrid domain-specific language (DSL) specifically created for the modeling of cyber-physical system (CPS) whose components primarily interact through the exchange of physical resource flows such as water, heat or electricity. The language reuses well-known concepts from existing formalisms to create a coherent language with aim on usability and simplicity. Through its formal basis, CREST

concisely combines the syntactic and semantic features from several languages to support vital modeling tasks such as sound model creation, well-defined simulation and formal verification. In this article, we elaborate on the development of CREST and provide details on the language requirements of the DSL. CREST's graphical syntax is inspired by common modeling formalisms such as automata notations and architecture description languages. Its carefully formalised operational semantics assert the thorough combination of the languages' features into an executable formalism that focuses on the implementation of key aspects such as reactivity, locality, synchronism, parallelism and non-determinism. These principles have been identified as necessities for the correct modeling of our target systems, prior to the definition of the DSL. CREST is implemented in the form of `crestdsl`, an *internal* DSL that is based on the foundations of the Python programming language. Its aim is to provide a convenient means to CPS modeling, with the advantage of reusing a widespread programming language syntax. The use of Python as host language also has the advantage that existing language infrastructure such as familiar IDEs, established best practices and well-tested execution environments can be reused. Both CREST and `crestdsl` were developed with pragmatism in mind, so as to avoid recreating and redeveloping modeling concepts that are already broadly available.

Acknowledgements The authors would like to express their gratitude towards Alban Linard and Dimitri Racordon for the lively discussions and the feedback on our research. We also appreciate the effort of Aurélien Coet, Stefan Bodoarcă and Guillaume Marthe for helping us improve the usability of CREST and `crestdsl`.

Funding Open access funding provided by University of Geneva

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

A formalisation of CREST

This section provides an overview of the formal aspects of CREST. The following definitions are a summarised version of the original formalisation (see [50]). Due to spatial considerations, we only provide a brief introduction and refer the reader to the extended original for a more elaborate description including detailed explanations, examples and discussions.

A.1 formal language structure

CREST's formalisation (both structure and semantics) is defined on a system-global level. This means that states, transitions, ports, etc., are first defined as a system-wide sets and then divided into mutually exclusive sets for each entity.

To help this separation, the formalisation uses the notion of non-overlapping set partitions (denoted by the $\bigsqcup$ operator).

Notation (Partition of sets) *Given a set S, the subsets $S_1, \dots, S_n$ are defined to be a partition of $S = \bigsqcup_i S_i$ iff $\forall i, j, i \neq j \Rightarrow S_i \cap S_j = \varnothing$ and $S = \bigcup_{1 \leq i \leq n} S_i$.*

Definition 1 *(Time Base)* Being a timed language, CREST requires the definition of a time base $\mathbb{T}$ that the systems operate in. Next to the set of time values $\mathbb{T}$ is also required to contain an infinitesimal element ϵ and infinity element ∞. Usually we assume $\mathbb{T}$ to be positive rational or real, e.g. $\mathbb{R}_{\geq 0} \cup \{\epsilon, \infty\}$.

Definition 2 *(Types and Values)* Given *Units*, a set of resource units, and *Domains*, a set of value domains, the set of resource types is defined as $Types \subseteq Domains \times Units$. 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, the simplified notations *domain unit* and *v unit* can be used for resource types and values. Thus we write e.g. $\mathbb{N}$Watt and 3Watt for $\langle \mathbb{N}, \text{Watt}\rangle$ and $\langle 3, \text{Watt}\rangle$. For instance, Fig. 4 uses these definitions:

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

Definition 3 *(Hierarchy of Entities)* A CREST system's structure forms a rooted tree, where each entity can contain *children* entities. The system's entity tree is defined by a set of entity names *Entities*, and a function $parent : Entities \rightarrow Entities \cup \{\bot\}$, which returns the parent of an entity or $\bot$ if it has no parent. The function $children : Entities \rightarrow \mathcal{P}(Entities)$ returns the direct children of an entity, and $root : Entities$ provides the system's only entity without parent.

Definition 4 *(Ports)* CREST systems use *ports* for the transfer of resources, and storage of data. These ports are defined

by a set of port names *Ports*, and a function $type : Ports \rightarrow Types$ that assigns the resource type of each port. The AC example's port names and types are

$$\begin{aligned} Ports &= \{\text{temperature, switch, ontime}, \dots\} \\ type(\text{temperature}) &= \mathbb{R}\texttt{Celsius} \\ type(\text{switch}) &= \{\text{on, off}\}\texttt{Switch} \\ type(\text{ontime}) &= \mathbb{R}\texttt{Time} \end{aligned}$$

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

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

Intersection of these partitions defines each entity's inputs, outputs and locals:

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

To enforce the locality principle, CREST allows only certain ports to be used in transition guards (*sources*) and updates only to write to other specific ports (*targets*).

The function $sources : Entities \rightarrow \mathcal{P}(Ports)$ defines those ports of an entity, that can be used to calculate transition guards or the value of update functions. They consist of an entity's inputs and locals, and its subentities' outputs.

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

Similarly, $targets : Entities \rightarrow \mathcal{P}(Ports)$ defines the set of possible targets of update functions, i.e. its 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 5 *(Bindings)* $Bindings = \{b : Ports \rightarrow Resources \mid \forall p \in Ports, b(p) \in type(p)\}$, is the set of mappings that associates each port with a value of its respective resource *type*. For instance $b(\text{temperature}) = 24\texttt{Celsius}$ and $b(\text{switch}) = \text{off}\texttt{Switch}$.

Definition 6 *(States and Transitions)* Entity behaviour is defined using state automata, such that each entity specifies its own automaton. The system's set of states *States*, is globally defined and partitioned into subsets for each entity, such that each entity has at least one state:

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

Transitions are defined by the *Transitions* relation, which associates two states (of the same entity) and a guard function name $t \in \mathcal{T}$

$$Transitions \subseteq \bigcup_{e \in Entities} \left(States_e \times States_e \times \mathcal{T}\right)$$

The function $\tau : \mathcal{T} \rightarrow (Bindings \times Bindings \rightarrow \mathbb{B})$ maps the guard function names to guard function implementations. CREST does not specify a syntax or semantics for these implementations but requires them to adhere to a signature. Specifically when called with a current port bindings *binding* and a previous port bindings *pre* ($binding, pre \in Bindings$) guard implementations need to yield whether the transition is enabled.

Definition 7 *(Updates)* Updates are used to modify port values. Each update relates an automaton state to a target port and an update function name $u \in \mathcal{U}$, so that u continuously updates the target port's value when the automaton is in the related state.

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

Only one update definition is allowed for each combination of target port and state, to avoid write-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 $\upsilon : \mathcal{U} \rightarrow (Bindings \times Bindings \times \mathbb{T} \rightarrow Resources)$ maps the update names to their implementation functions. Applied to port bindings *bind*, 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 target port. Note the use of time $\mathbb{T}$ that allows time-based value evolution.

Notes 1 The *pre* binding stores all ports' previous value bindings. It can be used for various functionality in CREST systems where knowledge of the previous value is required, such as change of a port p's value over time using update u's implementation $\upsilon(u)(b, pre, \delta t) = pre(p) + 2 * \delta t$.

Definition 8 *(dependencies)* The function $dependencies : \mathcal{U} \rightarrow Ports$ returns a set of ports for each update function

name. We add a constraint that an update's dependencies can only be source-ports of the update's entity.

$$\forall \langle s, p, u\rangle \in Updates, \forall e \in Entities, s \in States_e, p \in targets(e),\\ dependencies(u) \subseteq sources(e), p \notin dependencies(u)$$

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

Note, that CREST entities are not allowed to specify circular dependencies between ports. This means that if a dependency e.g. reads a port A and writes B, then there cannot be an update reading B and write A.

Definition 9 *(io-dependencies) io-dependencies* : $Ports \rightarrow \mathcal{P}(Ports)$ is a function that specifies the dependencies inside an entity. As entities are usually treated as black boxes, by default the assumption is that all output ports depend on all input ports. This assumption can occasionally lead to cyclic dependencies between subentities. *io-dependencies* can help resolve these dependencies by "shining a light" into the black box and revealing the actual dependencies inside a subentity.

$$\forall e \in Entities, \forall p \in Ports^O_e, io\text{-}dependencies(p) \subseteq Ports^I_e$$

Due to spatial limitations, we refer the reader to [50] for a detailed explanation of this aspect.

A.2 Global state of a CREST system

Definition 10 *(State of the system)* The global state $w \in W$ of an entire CREST system is a combination of the current states of all entity automata, the bindings of all ports, the previous bindings of the ports, and a global time.

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

Each CREST system further needs to define its initial state $w_0 \in W$.

The set of current automata states (not to be confused with the global system state) is given by $Currents = \{f : Entities \rightarrow States \mid \forall e \in Entities, f(e) \in States_e\}$. In the AC example $current \in Currents$ is initially defined as $current(\text{AirCondition}) = \text{Off}$.

Definition 11 *(CREST Syntactic Structure)* Based on the previous definitions, a CREST system is specified as a structure S containing information about the resources (data types), entity hierarchy, ports, states and transitions, updates, dependencies, io-dependencies, and the initial global state w_0:

$$S = \langle Units, Domains, \quad Entities, parent, \quad Ports, type,\\ States, Transitions, \mathcal{T}, \tau, \quad Updates, \mathcal{U}, \upsilon,\\ dependencies, \quad io\text{-}dependencies, \quad w_0\rangle$$

A.3 Operational semantics

The following definitions specify the modification of individual automaton states and port values. The propagation of such changes' effects on a complete CREST system and the upkeep of a well-formed system state however require more complex routines that are defined further below using structured operational semantics (SOS) rules.

Definition 12 *(Change of automata states)* The state transition of an entity e to a state s is represented by $w[e \mapsto s]$. This change within one entity creates a new (global) system state w' where the *current* automaton state of all entities remains the same, except for e (the entity to be updated), which now maps to s.

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

Definition 13 *(Change of port values)* Changes to port bindings are denoted by $w[ps]$, where ps is a set of port-value mappings $(p \mapsto r)$ such that there is at most one mapping for each p.[15] 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.

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

Note that the previous port values *pre* of the ports in *ps* have to be updated, so that efficient dataflow modeling is possible and value changes can be observed.

A.3.1 Modifiers and precedence

The propagation of state updates within a CREST system requires a correctly ordered execution of updates and state transitions throughout the entire entity hierarchy. Thus, this execution order has to be established and used. The formal semantics use a precedence operator $\prec$ that defines the according dataflows semantics. The operator expresses a partial order between ports, updates and child entities that arises from the dependencies. In this subsection we look at the formal definitions of these helper functions and operators.

[15] i.e. there cannot be a set of mappings $ps = \{p \mapsto r, p \mapsto s\}$.

Definition 14 *(Port Precedence)* The $\prec$ operator for ports defines a partial order based on the *dependencies* function (see Definition 8), and the input-output dependency function *io-dependencies* (Definition 9). We say that for any two ports $p_1, p_2 \in Ports$ $p_1 \prec p_2$ iff one of the following cases applies:

1. there is an update that reads p_1 to calculate the value written to p_2 (i.e. p_1 is a dependency of an update that writes p_2);
2. there is an entity, and p_1 is an input, p_2 is an output and there exists an io-dependency between the two;
3. there exists a port p' so that $p_1 \prec p'$ and $p' \prec p_2$ (i.e. $p_1 \prec p_2$ by transitivity).

Formally $\prec$ is expressed as: $\forall p_1, p_2 \in Ports,\ p_1 \prec p_2$ iff

$$\begin{aligned}&\exists \langle s, p_2, u\rangle \in Updates,\ p_1 \in dependencies(u) && (Case1)\\ &\vee\ p_1 \in io\text{-}dependencies(p_2) && (Case2)\\ &\vee\ \exists p' \in Ports,\ p_1 \prec p' \wedge p' \prec p_2 && (Case3)\end{aligned}$$

Note that $\prec$ satisfies anti-symmetry to avoid circular dependencies between ports.

Definition 15 *(Active-modifiers)* An entity's *modifiers* are all elements that have the capability of altering an entity's target ports' values, i.e. its updates and subentities. To facilitate the subsequent definitions, we define the set of all modifiers to be the union of all entities and updates.

$$Modifiers = Entities \cup Updates$$

Active modifiers are those modifiers that are "active" in a certain system state $w \in W$. This means all updates that are related to a currently active automaton state, and all subentities. *active-modifiers* yields all such updates and child entities for an entity.

$$\begin{aligned}&active\text{-}modifiers : W \times Entities \rightarrow \mathcal{P}(Modifiers)\\ &active\text{-}modifiers(\langle current, bind, pre, time\rangle, e) =\\ &\quad \{\langle s, p, u\rangle \in Update \mid s = current(e)\} \cup children(e)\end{aligned}$$

modified-ports returns the list of ports that are modified by an entity. This means, it consists of all ports that are the targets of the update functions of the current automaton state or outputs of a subentity.

$$\begin{aligned}&modified\text{-}ports : W \times Entities \rightarrow Ports\\ &modified\text{-}ports(\langle current, bind, pre, time\rangle, e) =\\ &\qquad \{p \mid \exists \langle s, p, u\rangle \in Updates, s = current(e)\} \cup\\ &\qquad \{p \mid \exists e' \in children(e), p \in Ports^O_{e'}\}\end{aligned}$$

Definition 16 *(ordered-ports)* The function *ordered-ports* creates a total order of ports that are modified according to their precedence.

$$\begin{aligned}&ordered\text{-}ports : W \times Entities \rightarrow PortLists\\ &ordered\text{-}ports(w, e) : [p_0, p_1, \ldots p_n] \text{ s. t. } \forall p_i, p_j, i\\ &\qquad < j, p_i \prec p_j\end{aligned}$$

The list of ports is defined by the *PortLists* type:

$$PortLists := \varnothing \mid \langle Ports, PortLists\rangle$$

We use the common list notation $[p_0, p_1, p_2]$ instead of $\langle p_0, \langle p_1, \langle p_2, \varnothing\rangle\rangle\rangle$, where $i \in \mathbb{N}$ is a port's list index. The operator ":" splits a list's *head* (its first element) from its *tail* (the rest of the list) as follows: $[p_0 : tail]$, where p_0 is the first element and *tail* the rest $[p_1, p_2, \ldots, p_n]$.

Definition 17 *(ordered-modifiers)* Based on the list of modified ports, we create a list of modifiers (updates and subentities) to specify their correct execution order, so that elements are executed only after the ports whose values they depend on are updated. This avoids calculations using wrong or outdated values. The type *ModifierLists* is used to describe such lists of modifiers. Its signature is given as

$$ModifierLists := \varnothing \mid \langle Modifiers, ModifierLists\rangle$$

The notation $[m_0, m_1, m_2, m_3, m_4]$ and the "head-tail" operator ":" are defined for lists of modifiers, similar to the *PortLists* type above.

ordered-modifiers creates a modifier list so that for each port in ordered-ports there is a modifier that updates it.

$$\begin{aligned}&ordered\text{-}modifiers : W \times Entities \rightarrow ModifierLists\\ &ordered\text{-}modifiers(w, e) : [m_0, m_1, \ldots m_n, m_{n+1}, \ldots, m_{n+k}]\end{aligned}$$

Specifically, for each port p_i, there is a modifier m_i that alters this p_i's value, i.e. $\forall p_i \in ordered\text{-}ports(w, e)$

$$\begin{aligned}&\exists m_i \in active\text{-}modifiers(w, e)\\ &\quad \wedge \begin{cases} m_i = \langle s, p_i, u\rangle \in Updates\\ m_i = e' \in children(e), p_i \in Ports^O_{e'}\end{cases}\end{aligned}$$

All additional modifiers $m_n, \ldots, m_{n+j}$ are appended at the end of the list:

$$\begin{aligned}&\forall m_{n+j} \in active\text{-}modifiers(w, e), 0 < j \leq k,\\ &\quad \nexists p \in ordered\text{-}ports(w, e),\\ &\quad m_{n+j} = \langle s, p, u\rangle \in Updates \wedge m_{n+j}\\ &\quad = e' \in children(e), p \in Ports^O_{e'}\end{aligned}$$

Definition 18 *(enabled-transitions)* The function *enabled-transitions* finds all transitions from an entity's currently active automaton state, whose guard functions evaluate to `True`.

$$
\begin{aligned}
&\textit{enabled-transitions} : W \times \textit{Entities} \rightarrow \mathcal{P}(\textit{Transitions}) \\
&\textit{enabled-transitions}(\langle \textit{curr}, \textit{bind}, \textit{pre}, \textit{time}\rangle, e) = \\
&\qquad \{\langle s, t, g\rangle \in \textit{Transitions} \mid s, t \in \textit{States}_e \\
&\qquad\quad \wedge\ s = \textit{curr}(e) \wedge \\
&\qquad \tau(g)(\textit{bind}, \textit{pre}) \mapsto \textbf{True}\}
\end{aligned}
$$

A.4 Formal operational semantics

CREST's semantics describe modifications of the global system state ($w \in W$) and the propagation of changes from the *root* to the leaves of the system's entity tree. Thus, each entity maintains its own state and triggers the update of its direct subentities.

The semantics revolve around the concept of reaching a fixed point ("fixpoint") after each system modification. In a fixpoint the system is stable and no changes happen unless time passes or external factors modify the system's inputs.

set-values This fixpoint concept is triggered when modifying the system's input port values, as shown in Rule 1. The altering of the port value bindings using according to the set *vs* requires a subsequent application of the `stabilise` rule on the system's *root*. Stabilisation triggers an entity's updates and automaton transitions until a fixpoint (stable state) is reached. In the process stabilisation also recursively propagates port value modifications to the subentities.

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

stabilise Rule 2 is called on an entity e that should be stabilised, using a timestep size δt. Here, δt is the time that should be advanced in the system. For stabilisation of the system without time advance, (i.e. after the setting of port values), this rule is called with $\delta t = 0$, to propagate values but not take time into account in update functions.

Specifically, the rule first obtains the ordered list of modifiers and triggers their (ordered) execution in the `apply-all` rule. Subsequently, `transitions` executes the automaton transitions.

$$
\frac{\begin{gathered}\textit{mods} = \textit{ordered-modifiers}(w, e), \\ \big\langle \textit{set-pre}(w, e), e, \textit{mods}, \delta t\big\rangle \xrightarrow{\text{apply-all}} w_1, \langle w_1, e\rangle \xrightarrow{\text{transitions}} w_2\end{gathered}}{\langle w, e, \delta t\rangle \xrightarrow{\text{stabilise}} w_2} \tag{2}
$$

The above rule uses a *set-pre*, that, given a state $w = \langle \textit{curr}, \textit{bind}, \textit{pre}, \textit{time}\rangle$ and an entity e, creates a new state, where the ports' previous value binding *pre* is updated. This function is used before triggering modifiers execution, to assert that the modifiers have access to the port's previous values (i.e. the values before the updates), which is necessary e.g. for incremental increases of port values and resolution of algebraic cycles.

$$
\textit{set-pre}\big(\langle \textit{curr}, \textit{bind}, \textit{pre}, \textit{time}\rangle, e\big) = \langle \textit{curr}, \textit{bind}, \textit{pre}', \textit{time}\rangle
$$

$$
\text{where } \textit{pre}'(p) = \begin{cases} \textit{bind}(p) & \text{if } p \in \textit{targets}(e) \\ \textit{pre}(p) & \text{otherwise} \end{cases}
$$

apply-all This rule takes an ordered list of modifiers and executes the first one (m_0). Subsequently, the rule recurses to execute the rest of the list. When the list is empty (i.e. $\varnothing$), Rule 4 serves as a break-condition to the recursion. In this case, no action is taken and w remains unchanged.

$$
\frac{\langle w, m_0, \delta t\rangle \xrightarrow{\text{apply-one}} w_1, \quad \langle w_1, \textit{mods}, \delta t\rangle \xrightarrow{\text{apply-all}} w_2}{\langle w, [m_0 : \textit{mods}], \delta t\rangle \xrightarrow{\text{apply-all}} w_2} \tag{3}
$$

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

apply-one Two `apply-one` functions execute a modifier (update or subentity) based on its type. If the modifier is an update, `update` is called to calculate a new port value. Otherwise (if the modifier is a child entity), Rule 6 triggers `stabilise` to propagate the changed system state to the subentity.

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

$$
\frac{\textit{mod} \in \textit{Entities}, \quad \langle w, \textit{mod}, \delta t\rangle \xrightarrow{\text{stabilise}} w_1}{\langle w, \textit{mod}, \delta t\rangle \xrightarrow{\text{apply-one}} w_1} \tag{6}
$$

update calculates the update's target port value based on its function implementation (identified by $\upsilon(u)$). The new system state is calculated by taking the old state w and setting update's target port p to the calculated value.

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

transitions The `transitions` rules are responsible for triggering transitions and deciding whether further stabilisation is required. If enabled transitions exist (Rule 8), one of them is executed ($w[e \mapsto t]$) and the rule recurses on the stabilisation rule. Without enabled transitions, no further action is taken (Rule 9).

$$\frac{\langle s, t, g\rangle \in enabled\text{-}transitions(w, e), \quad w_1 = w[e \mapsto t], \quad \langle w_1, e, 0\rangle \xrightarrow{\text{stabilise}} w_2}{\langle w, e\rangle \xrightarrow{\text{transitions}} w_2} \tag{8}$$

$$\frac{enabled\text{-}transitions(w, e) = \varnothing}{\langle w, e\rangle \xrightarrow{\text{transitions}} w} \tag{9}$$

Note 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".

Time Advance Being a timed formalism, it is necessary to discover at what point the system has to be brought into a coherent state. Specifically, this means that through continuous port value modifications (using updates), a transition might become enabled, requiring subsequent stabilisation. The semantics use four SOS rules to address several possibilities. The first two cover the basic requirements. As the time base $\mathbb{T}$ is defined for positive sets only, a CREST system cannot "step back in time".

An advance of $\delta t = 0$ triggers a stabilisation.

$$\frac{\delta t = 0, \langle w, root, 0\rangle \xrightarrow{\text{stabilise}} w'}{\langle w, \delta t\rangle \xrightarrow{\text{advance}} w'} \tag{10}$$

For $\delta t > 0$, we distinguish two cases. Assuming that, in a given system the next transition (and hence requirement for system stabilisation) becomes enabled at time t_{ntt} (due to the continuous time advances in update functions), we refer to

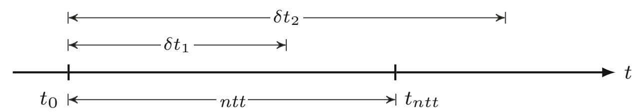

Fig. 11 Depending on δt and the point in time of the next transition (i.e. $t_0 + ntt = t_{ntt}$), Rule 11 is applied if $\delta t < ntt$ (e.g. δt_1) and Rule 12 otherwise (e.g. δt_2)

the duration until that next transition time as *ntt*. Any time advance $\delta t < ntt$ only requires stabilisation at the end of the advance, to update the update ports' target values. This is implemented in Rule 11.

$$\frac{\delta t \leq next_transition_time(w), \quad \langle set\text{-}pre(w, e), root, \delta t\rangle \xrightarrow{\text{stabilise}} \langle curr, bind, pre, time\rangle}{\langle w, \delta t\rangle \xrightarrow{\text{advance}} \langle curr, bind, pre, time + \delta t\rangle} \tag{11}$$

If δt is bigger than the next transition time *ntt*, Rule 12 splits the advance into two steps: First, it will *advance* with $\delta t = ntt$ (using Rule 11) and trigger the a stabilisation, including transition firing and activating of the set of updates related to the new current state. Next, CREST recurses on the *advance* rule using the remaining time (i.e. $\delta t - ntt$). This will trigger either Rule 11 or Rule 12.

$$\frac{\delta t > ntt, \quad ntt = next_transition_time(w), \quad \langle w, ntt\rangle \xrightarrow{\text{advance}} w_1, \quad \langle w_1, \delta t - ntt\rangle \xrightarrow{\text{advance}} w_2}{\langle w, \delta t\rangle \xrightarrow{\text{advance}} w_2} \tag{12}$$

Figure 11 depicts these two scenarios graphically.

CREST's semantics rely on the availability of $next_transition_time : W \rightarrow \mathbb{T}$. Given a system's current state w, the function calculates the precise amount of time δt that has to pass until updates enable any transition's guard condition. It returns ∞ in case time-based advances do not enable any transition.

Note that these semantics do not prescribe an actual implementation of this function. Depending on the available resources, different ways of computing *next_transition_time* are possible, such as a search approach or complex analysis techniques that include symbolic reasoning. CREST's implementation translates the system functions into constraints that are handed to an SMT prover that aims to find a minimum next transition time.

References

1. Abdelzad, V., Amyot, D., Lethbridge, T.C.: Adding a textual syntax to an existing graphical modeling language: experience report with

grl. In: International SDL Forum, pp 159–174. Springer, Berlin (2015)
2. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid annex: An aadl extension for continuous behavior and cyber-physical interaction modeling. In: ACM SIGAda Ada Letters, vol. 34, pp. 29–38. ACM (2014)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
4. Bächle, M., Kirchberg, P.: Ruby on rails. IEEE Softw. **24**(6), 105–108 (2007)
5. Bariic, A., Amaral, V., Goulao, M.: Usability evaluation of domain-specific languages. In: 2012 Eighth International Conference on the Quality of Information and Communications Technology, pp. 342–347. IEEE (2012)
6. Barisic, A., Amaral, V., Goulão, M.: Usability driven DSL development with USE-ME. Comput. Lang. Syst. Struct. **51**, 118–157 (2018)
7. Barroca, B., Lucio, L., Buchs, D., Amaral, V., Pedro, L.: Composition for model-based test generation DSL Composition for model-based test generation. Electron. Commun. EASST, 21, 2009
8. Bechtold, S., Brannen, S., Link, J., Merdes, M., Phillip, M., Stein, C.: Junit 5 user guide. Version 5.0 (2016)
9. Bergero, F., Kofman, E.: PowerDEVS: A tool for hybrid system modeling and real-time simulation. Simulation **87**(1–2), 113–132 (2011)
10. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
11. Bettini, L.: Implementing Domain-specific Languages with Xtext and Xtend. Packt Publishing Ltd, Birmingham (2016)
12. Black, D.C., Donovan, J., Bunton, B., Keist, A.: SystemC: From the Ground Up. Springer, Berlin (2010)
13. Bourke, T., Pouzet, M.: Zélus: A Synchronous Language with ODEs. In: 16th International Conference on Hybrid Systems: Computation and Control, pp. 113–118, Philadelphia, USA, March (2013)
14. Bouyer, P., Laroussinie, F.: Model Checking Timed Automata. In: Merz, S., Navet, N., (eds), Modeling and Verification of Real-Time Systems, pp. 111–140. Wiley, New York (2008)
15. Broenink, J.F.: Introduction to physical systems modelling with bond graphs. In: SiE Whitebook on Simulation Methodologies (1999)
16. Broman, D., Lee, E.A., Tripakis, S., Törngren, M.: Viewpoints, formalisms, languages, and tools for cyber-physical systems. In: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM '12, pp. 49–54, ACM, New York, USA (2012)
17. Campagne, F.: The MPS language workbench: volume I, vol. 1. Fabien Campagne (2014)
18. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and Tools for Hybrid Systems Design. Found. Trends Electron. Des. Autom. **1**(1/2), 1–193 (2006)
19. Cazzola, W., Poletti, D.: Dsl evolution through composition. In: Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE 10, Association for Computing Machinery, New York, USA (2010)
20. Cuadrado, J.S., Izquierdo, J.L., Molina, J.G.: Comparison between internal and external dsls via rubytl and gra2mol. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, pp. 109–131. IGI Global (2013)
21. De Laet, T., Schaekers, W., de Greef, J., Bruyninckx, H.: Domain Specific Language for Geometric Relations between Rigid Bodies Targeted to Robotic Applications. CoRR (2013)
22. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 4963 of Lecture Notes in Computer Science, pp. 337–340. Springer (2008)
23. Delange, J., Feiler, P.: Architecture fault modeling with the aadl error-model annex. In: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 361–368. IEEE (2014)
24. Van Deursen, A., Klint, P.: Little Languages: Little Maintenance? J. Softw. Maint. Res. Pract. **10**(2), 75–92 (2008)
25. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: Robotml, a domain-specific language to design, simulate and deploy robotic applications. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, pp. 149–160. Springer (2012)
26. Feniello, A., Dang, H., Birchfield, S.: Program synthesis by examples for object repositioning tasks. In: IEEE International Conference on Intelligent Robots and Systems, pp. 4428–4435 (2014)
27. Fowler, M.: Language workbenches: The killer-app for domain specific languages. https://www.martinfowler.com/articles/languageWorkbench.html (2005)
28. Fowler, M.: Domain-specific Languages. Pearson, London (2010)
29. França, R.B., Rolland, J.-F., Amine, M.F., Bodeveix, J.-P., Chemouil, D.: Assessment of the AADL Behavioral Annex. Journées FAC, p. 13 (2007)
30. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, pp. 37–54. IEEE (2007)
31. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in java. In: OOPSLA Companion, pp. 855–865 (2006)
32. Freeze, J.: Creating dsls with ruby. Ruby Code and Style **16**. https://www.artima.com/rubycs/articles/ruby_as_dsl.html (2006)
33. Friedenthal, S.: SysML: lessons from early applications and future directions. Insight **12**(4), 10–12 (2009)
34. Fritzson, P., Engelson, V.: Modelica — a unified object-oriented language for system modeling and simulation. In: European Conference on Object-Oriented Programming, vol. 1445 of Lecture Notes in Computer Science, pp. 67–90. Springer (1998)
35. Fryer, J.A., McKee, G.T.: Resource modelling and combination in modular robotics systems. Proc. - IEEE Int. Conf. Robot. Autom. **4**(May), 3167–3172 (1998)
36. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to uml profiles. UML Model Eng. **2**, 6–13 (2004)
37. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning, pp. 286–300. Springer, Berlin (2012)
38. Ghosh, D.: DSLs in Action. Manning Publications Co., New York (2010)
39. Ghosh, D.: Dsl for the uninitiated. Commun. ACM **54**(7), 44–50 (2011)
40. Arne Haber. MontiArc-Architectural Modeling and Simulation of Interactive Distributed Systems, vol. 24. Shaker Verlag GmbH, (2016)
41. Haber, A., Look, M., Seyed Nazari, P.M., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Composition of heterogeneous modeling languages. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) Model-Driven Engineering and Software Development, pp. 45–66. Springer International Publishing, Cham (2015)
42. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE, pp. 1305–1320 (1991)
43. Heitmeyer, C.L.: On the need for practical formal methods. In: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, vol. 1486 of Lecture

Notes in Computer Science, pp. 18–26, Springer, London, UK, (1998)
44. Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL (2011)
45. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. **111**(2), 193–244 (1994)
46. Horswill, I.D.: Functional programming of behavior-based systems. Auton. Robots **9**(1), 83–93 (2000)
47. Huang, C., Osaka, A., Kamei, Y., Ubayashi, N.: Automated dsl construction based on software product lines. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2015, pp. 247–254, Setubal, PRT, SCITEPRESS - Science and Technology Publications, Lda (2015)
48. International Telecommunication Union: Z.100: Specification and Description Language - Overview of SDL-2010, (2016). ITU Recommendation Z.100 (04/16); Article Number: E 40655
49. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
50. Klikovits, S.: A Domain-Specific Language Approach To Hybrid CPS Modelling. PhD thesis, University of Geneva, Switzerland (2019)
51. Klikovits, S., Coet, A., Buchs, D.: ML4CREST: Machine learning for CPS models. In: 2nd International Workshop on Model-Driven Engineering for the Internet-of-Things (MDE4IoT) at MODELS'18, vol. 2245 of CEUR Workshop Proceedings, pp. 515–520 (2018)
52. Klikovits, S., Linard, A., Buchs, D.: CREST - A DSL for Reactive Cyber-Physical Systems. In: Ferhat Khendek and Reinhard Gotzhein, editors, 10th System Analysis and Modeling Conference (SAM 2018), vol. 11150 of Lecture Notes in Computer Science, pp. 29–45. Springer (2018)
53. Klikovits, S., Linard, A., Buchs, D.: CREST formalization. Technical report, Software Modeling and Verification Group, University of Geneva (2018)
54. Kofman, E., Junco, S.: Quantized-state systems: A DEVS approach for continuous system simulation. Trans. Soc. Comput. Simul. Int. **18**(3), 123–132 (2001)
55. Kopetz, H.: The complexity challenge in embedded system design. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 3–12. IEEE (2008)
56. Lacoste-Julien, S., Vangheluwe, H., De Lara, J., Mosterman, P.J.: Meta-modelling hybrid formalisms. In: Proceedings of the IEEE International Symposium on Computer-Aided Control System Design, pp. 65–70 (2004)
57. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Science of Computer Programming, pp. 128–192 (March 2015)
58. MathWorks: Simulink: Getting Started Guide (2017)
59. Mora, B., García, F., Ruiz, F., Piattini, M.: Graphical versus textual software measurement modelling: an empirical study. Softw. Quality J. **19**(1), 201–233 (2011)
60. Mustafiz, S., Gomes, C., Barroca, B., Vangheluwe, H.: Modular Design of Hybrid Languages by Explicit Modeling of Semantic Adaptation. In: TMS/DEVS Symposium on Theory of Modeling and Simulation (TMS/DEVS 2016). Society for Modeling and Simulation International (SCS) (2016)
61. Nordmann, A., Hochgeschwender, N., Wigand, D.L., Wrede, S.: A Survey on domain-specific modeling and languages in robotics. J. Softw. Eng. Robot. (JOSER) **7**(1), 75–99 (2016)
62. Object Management Group: Business Process Model And Notation (BPMN) Version 2.0 (2011) OMG Document Number: formal-2011-01-03
63. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (OMG MARTE) Version 1.1 (2011) OMG Document Number: formal/11-06-02
64. Object Management Group: Object Management Group: Unified Modeling Language (UML) Version 2.5.1 (2017) OMG Document Number: formal/17-12-05
65. Object Management Group: OMG Systems Modeling Language (OMG SysML) Version 1.5, (2017) OMG Document Number: formal-2017-05-01
66. Peterson, J., Hager, G.D., Hudak, P.: A language for declarative robotic programming. Proc. -IEEE Int. Conf. Robot. Autom. **2**(May), 1144–1151 (1999)
67. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. Commun. ACM **38**(6), 33–44 (1995)
68. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn (1962)
69. Pohjonen, R., Kelly, S.: Domain-specific modeling. Dr. Dobb's J. **27**, 8 (2002). https://www.metacase.com/papers/DrDobbs_Domain-Specific_Modeling.html
70. Ptolemaeus, C., (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
71. Rieger, C., Westerkamp, M., Kuchen, H.: Challenges and opportunities of modularizing textual domain-specific languages. In: MODELSWARD (2018)
72. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. CoRR, abs/1509.04505 (2015)
73. Sadati, S.M.H., Zschaler, S., Bergeles, C.: A matlab-internal DSL for modelling hybrid rigid-continuum robots with TMTDyn. In: Proceedings — 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2019, pp. 559–567 (2019)
74. Selic, B.: Model-driven development: Its essence and opportunities. In: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), pp. 7–pp. IEEE (2006)
75. Society of Automotive Engineers: Society of Automotive Engineers: Architecture Analysis and Design Language (SAE AADL) Version 2.2 (2017) SAE Standard: AS5506C
76. Steinberg, D., F. Budinsky, Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
77. Tolvanen, J.-P., Rossi, M.: Metaedit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 92–93. ACM (2003)
78. van der Aalst, W.M.P.: Business process management as the killer app for petri nets. Softw. Syst. Model. **14**(2), 685–691 (2015)
79. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices **35**(6), 26–36 (2000)
80. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C. L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
81. Zeigler, B.P.: Theory of Modelling and Simulation. Wiley, New York (1976)
82. Zhang, F., Yeddanapudi, M., Mosterman, P.J.: Zero-crossing location and detection algorithms for hybrid system simulation. IFAC Proc. **41**(2), 7967–7972 (2008)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Stefan Klikovits is a postdoctoral researcher at the National Institute of Informatics in Tokyo, Japan, researching on formal verification of complex cyber-physical systems such as automated vehicles. His research interests include CPS modeling and simulation, testing and formal verification of artificial intelligence and black-box components, and the applied use of formal techniques such as monitoring and falsification. Stefan Klikovits received his PhD in 2019 from the University of Geneva, Switzerland, where he worked on the modeling of cyber-physical systems and the design of domain-specific modeling languages with focus on usability. During his PhD, he also worked for CERN, the European Center for Nuclear Research, where he researched on automated test case generation and testing of industrial controls software for particle accelerators and high energy physics experiments.

Didier Buchs (PhD 1989) is Full professor at the Computer Science department of the Faculty of Science of the University of Geneva. He has a long experience in the design of modeling languages for complex concurrent systems. His research addresses the development of modeling languages at various levels, from syntactic composition to semantics composition and verification using model checking and testing. Didier Buchs designed various high level Petri nets dialects for solving issues about structuring complex models (CO-OPN language). Tooling was also an important side results of these researches, leading to incorporate Petri nets with other models for simulation purposes. More recently, in order to develop generic model checkers, StrataGEM was designed as a rewrite system for sets of terms. This has been the core concept for verifying various modeling languages such as Petri nets and CTL logic based on the underlying, very efficient new decision diagram data structure called Σ-DD. In good cases it is able to exponentially reduce the costs (in time and memory) of model checking for high level models.