The Common Requirement Modeling Language

Daniel Bouskela[1] Lena Buffoni[2] Audrey Jardin[1] Vince Molnár[3] Adrian Pop[2] Ármin Zavada[4]

[1]Electricité de France, France, {daniel.bouskela,audrey.jardin}@edf.fr
[2]Linköping University, Sweden, {lena.buffoni,adrian.pop}@liu.se
[3]Budapest University of Technology and Economics, Hungary molnarv@mit.bme.hu
[3]IncQuery Labs cPlc., Hungary armin.zavada@incquerylabs.com

Abstract

CRML (the Common Requirement Modeling Language) is a new language for the formal expression of requirements. The goal is to release the language as an open standard integrated into the open source modeling and simulation tool OpenModelica and interoperable with the open systems engineering standard SysMLv2. CRML allows for the expression of requirements as multidisciplinary spatiotemporal constraints that can be verified against system design by co-simulating requirements models with behavioral models. The requirements models must be easily legible and sharable between disciplines and stakeholders and must capture realistic constraints on the system, including time-dependent constraints with probabilistic criteria, in recognition of the fact that no constraint can be fulfilled at any time at any cost. The theoretical foundation of the language lies on 4-valued Boolean algebra, set theory and function theory. The coupling of the requirements models to the behavioral models is obtained through the specification of bindings, the automatic generation of Modelica code from the CRML model and use of the FMI and SSP standards. CRML and the proposed methodology is compatible with SysMLv2, forming a comprehensive workflow and tool-chain encompassing requirement analysis, system design and Validation and Verification (V&V). The final objective is to facilitate the demonstration of correctness of system behavior against assumptions and requirements by building a workflow around Model-Driven Engineering and Open Standards for automating the creation of verification simulators.

Keywords: cyber-physical systems, systems engineering, requirement modelling, systems verification, Modelica, FMI, SSP, SysML

1 Motivations and Challenges

Large numbers of stakeholders are involved in the design and operation of complex cyber-physical systems (CPS), especially but not exclusively in the energy sector. When working on a common system, stakeholders tend to express requirements from their own perspective, resulting in a global set of constraints on the system that can be conflicting, even contradictory. Also, to avoid questioning the motivations of poorly-documented past design decisions, new requirements are often added without questioning the soundness of existing ones. This results in over-specifications, delays, and cost overruns. The search for a common agreement between stakeholders that preserves degrees of freedom for optimal design is always difficult and lengthy (Azzouzi et al. 2022).

CRML (Common Requirement Modeling Language) is a new language for the formal expression of requirements collaboratively developed by different industrial and academic stakeholders. **The goal is to release CRML as an open standard to offer stakeholders from different domains and disciplines a common language to express, organize, negotiate, and simulate requirements in order to find the best compromise that suits their needs while complying with their mutual commitments**.

This goal raises the question of expressing CPS requirements that are realistic, understandable, and verifiable by and between stakeholders. More precisely, it means that the underlying formal language and method associated have to tackle the following challenges:

- The language should provide **comprehensive descriptions of all spatiotemporal assumptions and constraints** that bear on the system under study. Constraints can be of all kinds and may vary depending on the system operating mode: physical, performance (reliability, availability, economical…), and regulatory (safety, security, environmental, reserve capacity for grid balancing, grid access, and priority dispatch…).
- The **requirements models must be easily legible and unambiguous**. It is expected that a requirement language common to all stakeholders regardless of their expertise and business domain will improve the productivity of studies. To that end, the syntax *must be close to natural language*.
- CPSs exhibit strong physical aspects. Therefore, **particular attention must be paid to physical aspects**: physical units, real-time, events, synchronism and asynchronism, components and objects, failures, and uncertainties. Time-dependent continuous and discrete variables must be dealt with in a hybrid synchronous and asynchronous framework. This goes

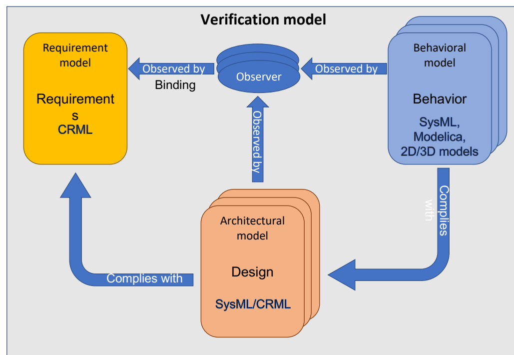

Figure 1. Architecture of the verification model

well beyond finite automata that are the reference in model checking (Baier and Katoen 2008).

- **Verification should be automated as much as possible all along the system lifecycle by co-simulating requirements models with solution models of any kind and growing complexity** ranging from *min* and *max* limits (that represent authorized operation domains), to finite state automata (that represent logical system operation), and to multidomain physical 0D/1D/2D/3D models (that represent detailed system physical behavior).

  The verification model (Fig. 1), which tells whether requirements are satisfied or not, consists of the requirement models, the solution models (behavioral and architectural), the observers of the solution models, and the links between observers and requirements, which are called "bindings". Requirements models with observers act as virtual sensors to detect possible requirement violations of the solution models. Behavioral models capture the dynamic behavior of the system in various forms (state automata, algebraic or differential equations), whereas architectural models carry static information about the system.

- **Probabilistic criteria must be added to requirements** because no realistic requirement can be satisfied with absolute certainty: in general, it is not enough to specify what should happen in nominal mode, one should also define what should happen in the event of specific hazards and with which probability the real system will not enter in one of these cases (i.e. how reliable the system should be). These stochastic requirements have to be verified against stochastic solution models. Monte Carlo techniques could hence be used to simulate the verification models.

2 State of the Art

The current state of the art regarding the above challenges tends to consider them separately. Consequently, there is no integrated tool able to deal with all of them in a consistent way. The main gaps concern the links between logical design, physical design, and dependability analysis as they currently involve completely different methods and tools: logical design uses methods such as UML (OMG 2017) and SysML (OMG 2023) based on first-order logic that originates from the software industry. Physical design uses tools such as Modelica tools, Matlab, Simcenter Amesim, etc. that deal with physical laws in the form of DAEs (Differential-Algebraic Equations) (or block diagrams) and dependability analysis uses probabilistic methods.

2.1 Limitations of Requirement Modeling Tools for CPS

Regarding the modelling of requirements no convincing solution exists to express CPS requirements independently from design solutions in a formal way. Current state-of-the-art tools hence focus on expressing requirements in natural language such as in Rational DOORS, Polarion, etc. This comes from the fact that the existing formal requirements modelling methods such as LTL (Linear Temporal Logic) and CTL (Computation Tree Logic) (Baier and Katoen 2008), timed (Alur 1999) and hybrid (Henzinger 2000) automata or UML/SysML state behavioral diagrams (OMG 2019) tend to bear on abstractions of the system in the form of state machines, which *already express a solution* and hence **is not appropriate to correctly deal with CPS physical aspects**. For instance, with LTL, one can prove that a system will always or eventually pass through a given state. Timed automata can handle real-time, but only when the states are known in advance. This corresponds to an idealistic view of the system that is not for instance subject to wear or external aggression. Hence they do not consider situations where existing states are subject to gradual drift due to wear, or new states appear due to unexpected events. In other words, CPS contain finite-state machines, but they cannot as a whole be considered as finite-state machines. Other limitations could be quoted such as:

- **Lack of object-orientation**: temporal constraints cannot be (easily) associated to the system architecture (i.e. its decomposition into subsystems and components).

- **Difficult mathematical syntax**: although mathematical syntax (and semantics) is necessary to perform formal proofs (model checking) or even model simulation, it is difficult to use on a day-to-day basis for the whole system.

Various attempts have been made to alleviate these limitations, for example by extending OCL (Object Constraint Language) with temporal constraints, but none of them are used convincingly in practice (Kanso and Taha 2013). Therefore, formal requirements languages are usually only used for small (sub)systems with critical safety

DOI
10.3384/ecp204497

concerns, and they cover only the very early phases of system design when (only) the logic of the system is investigated.

The relatively new SysPhS (SysML Extension for Physical Interaction and Signal Flow Simulation) profile for SysML (OMG 2021) provides extensions to model physical interaction and flows independently of the simulation platform. It includes a textual syntax for mathematical expressions as well as reusable simulation elements. It also defines translations to Modelica and Simulink. Along with the previous approaches, SysPhS also focuses on the precise modeling of specific designs and not the specification of a *design envelope*.

The new SysMLv2 language (OMG 2023) will feature a native expression language and better modeling of requirements. It provides a mechanism to bind requirements to design artifacts to formalize them in terms of the chosen subject. Furthermore, analysis and verification cases provide a way to model evaluation and verification steps, and the language supports modeling spatiotemporal aspects as well. While this is much closer to our requirements, native constructs in SysMLv2 are not designed to fully cover aspects like the modeling of time-dependent hybrid systems, probabilistic criteria, or automated verification. The extensibility of the language, however, provides a way for us to make the two languages interoperable.

2.2 Why Modelica is Natively Suited for CPS Modeling but Not for Requirements?

Modelica (Mattsson, Elmqvist, and Otter 1998) is a language that comes with a convenient graphical interface fit for the description of the physical real-time behavior of CPS. However, Modelica does not allow one to express constraints on a system when its architecture is partially unknown (for example express a constraint on a valve, when it is unknown how many valves will be in the final design), and expresses the behavior of the system in the form of DAEs (Differential-Algebraic equations) rather than the constraints on the behavior of the system. As a consequence Modelica is insufficient to express all that is needed at the early design phases, especially when one wants to specify only the acceptable envelopes without going into realization details, so that the solution space is refined progressively, rather than committing to a single design decision that fits the criteria.

Graph-based design languages with their capability to explicitly modify product topology and parametrics are on the one hand partially able to fill this gap, but need to be extended on the other hand by more powerful formal methods for requirements processing, tracing and consistency checking (as illustrated in Section 2.1 with SysML).

Therefore, connecting formal requirement modeling languages such as the ones mentioned above directly to Modelica does not solve the general problem of having a model-based methodology that covers the whole engineering lifecycle for CPS, as such kind of solution is only valid if the system is considered as a state-machine and for the engineering phases past the detailed design phase (which is somewhat contradictory).

2.3 CRML Origin and History

As previously mentioned, a Modelica model expresses the behavior of the system but does not say for what purpose the model is made. For instance, the model of a cooling system features heat exchangers, but does not say anything about the properties of the system that we want to verify, e.g. whether the flow velocity inside the heat exchanger stays below a given threshold. To alleviate the problem, one of the ITEA EUROSYSLIB project (2007 – 2010) objectives was to investigate the possibility of associating constraints that represent requirements to a Modelica model. For the above example, that meant associating the constraint that the flow velocity must not exceed a given threshold to the model of a heat exchanger. The first idea was to express this kind of constraint directly in the Modelica model, for instance in a dedicated 'constraint' Modelica section (similarly to the existing 'equation' and 'algorithm' sections). However, this solution had the drawback to modify model components in order to handle specific constraints, which is not consistent with the generic nature of model components. Besides, with this solution there was no way to express something like *'No pump in the system must cavitate'* or *'At least one pump in the system must be started'*, for two reasons: (1) The notion of quantifier does not exist in Modelica, so the constraint must be written taking into account the current topology of the circuit (i.e. the number of pumps in the system), and modified when the model topology is changed (i.e. when the number of pumps changes), even if the meaning stays the same. (2) The notion of a pump being started or not is usually not present in the Modelica model, because it expresses the *physical state* of the pump, not its *operational state*. It becomes clear then that the requirement model must be separated from the behavioral model. The rest of the EUROSYSLIB project was then mainly devoted to look at different languages for expressing the properties of systems.

As no interesting requirement modelling language emerged, the idea to create a new language came within the ITEA MODRIO project (2012 – 2016) with the following aspects in mind: (1) the syntax must be close to natural language, (2) the language must handle time periods and probabilistic aspects, (3) the language must handle quantifiers (i.e., sets) and be object-oriented, and (4) it should be possible to automatically generate test sequences from the constraints that represent assumptions on the system. It resulted in the specification of a new language called FORM-L (for Formal Requirement Modeling Language), written by EDF (Nguyen 2014). In parallel other works emerged around the same period with rather close similarities: TOCL (a temporal extension of OCL) and Stimulus (Dassault Systèmes). Regarding TOCL, there is no known implementation and it seems that there is no ongoing effort. For Stimulus, it appears that the tool

is more focused on debugging requirements for a particular design and express them as temporal stochastic state machines, which is less general than FORM-L which aims at expressing requirements that are generic and that could be refined throughout the whole engineering cycle to evaluate multiple design solutions.

In 2013, FORM-L was proposed to the Modelica community (most of the Modelica community was participating in the MODRIO project, except OpenModelica), which offered to extend Modelica with the notion of 'blocks as functions' and develop a new Modelica library for the modelling of requirements (called Modelica Requirements) (Otter et al. 2015). This library proved to be unsatisfactory because the time period and the condition to be verified were mixed within the same block. Therefore, it was impossible to have a stable library because of the combinatorial explosion of possibilities of associating conditions with time periods (a new block would have to be developed each time a new case occurs, which would be almost as frequent as when a new requirement is written). Therefore, EDF decided to develop a new requirement Modelica library called ReqSysPro that would clearly separate time periods from conditions, so that a requirement would be obtained by associating a block representing a time period to a block representing a condition (and not having the two aspects in the same block). The first idea was to use state machines to evaluate requirements: a requirement would pass through different states, starting from 'Untested' until it becomes either 'Satisfied' or 'Violated'. The problem with this solution is (1) that it was not possible to find a single state machine that would handle all possible requirements, (2) that it was not possible to combine requirements together to form more complex requirements such as 'Requirement1 and Requirement2'. Then the idea came to use Boolean logic instead and, more precisely, a 4-value Boolean logic called ETL (Extended Temporal Language) (Bouskela and Jardin 2018). A new version of ReqSysPro was developed successfully (the only difficulty was the handling of time periods that needs dynamic allocation of memory) that was able to handle the temporal and condition aspects of FORM-L and to evaluate requirements to one of the four values {undefined, undecided, false, true}. To handle the other aspects (except the probabilistic aspects), it became necessary to have a FORM-L compiler. A prototype of a FORM-L compiler was developed by Inria and Sciworks on an EDF contract with a first operational version released in 2021. It demonstrated the feasibility of having a FORM-L to Modelica compiler. However, the compiler suffers from the drawback that it must be modified each time a new function is added to FORM-L. To alleviate this problem, the specification of a new language CRML was released at the end of 2021 within the ITEA project EMBRACE (2019 – 2022). The idea was to add the notion of functions (called operators in CRML) to be able to build complex functions from a limited number of elementary native functions as described in the following sections.

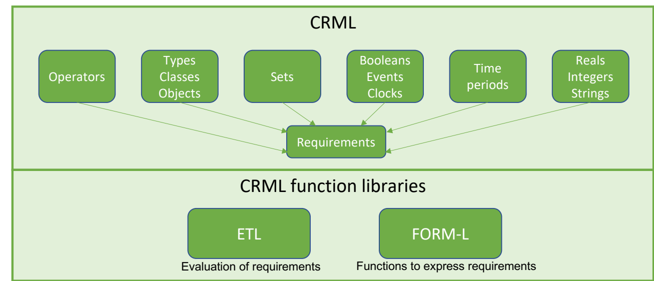

Figure 2. Architecture of the CRML language

3 The CRML Language

The language uses the concept of requirement made of four parts as introduced in the FORM-L language (Bouskela, Nguyen, and Jardin 2017; Nguyen 2019):

Spatial locator (WHERE): it defines the objects that are subject to the requirement. "Spatial" means that the objects are selected by some criteria on their properties that can be time-dependent.

Time locator (WHEN): it defines the time intervals when the requirements should be satisfied. A time interval is initiated when an event, called the opening event, occurs, and terminated when an event, called the closing event, occurs. An event occurs when a condition becomes true. A time locator can be composed of multiple time intervals that can overlap if several opening events occur before the closing event. In the following, the term 'time period' will be used as a synonym for 'time locator'.

Condition to be fulfilled (WHAT): it is the condition to be verified by the objects selected with the spatial locator within the bounds of the time periods selected by the time locator.

Probabilistic constraint (HOW_WELL): it defines a probabilistic constraint on the condition to be fulfilled.

The general architecture of the language is given in Fig. 2. Time periods and probabilistic constraints constitute the novelty of the approach. They are required to handle realistic requirements, because realistic requirements cannot be satisfied anytime at any cost. Time periods define when requirements are in effect and the time delay to satisfy them. Probabilistic constraints define some tolerance for the system to fail complying with the requirements. These two aspects have profound technical and economic impact on the design and operation of the system.

Classes, objects, sets and operators allow one to define the system structure and the objects properties, and enable to express generic requirements on sets of objects selected by their properties that can depend on time. All definitions can be stored in libraries for further reuse. There are two operator libraries provided with the language: the FORM-L library that implements the aspects of the FORM-L language that are related to requirement modelling (Nguyen 2019), and the ETL library that implements the low-level

DOI
10.3384/ecp204497

functions for requirement evaluation (Bouskela and Jardin 2018).

In the following, we present the salient aspects of CRML through the following example of a requirement: "During operation, the system should always stay within its operating domain. However, if the system fails to stay within its operating domain, then it should not stay outside of its operating domain for more than ten minutes more than three times per year, with a probability of success of 99.99%." First, we use a *mathematical notation* to explain the semantics of requirements. Then, starting from Section 3.3, we introduce the *CRML syntax*. The full CRML specification can be found at `http://crml-standard.org`.

3.1 Clocks and Time Periods

To express formally the example, we first define two Boolean variables. The first one called $b1$ is true when the system is in operation and false otherwise. The second called $b2$ is true when the system operates within its authorized operational domain and false otherwise. They are both external variables, which means that their values are given by an external behavioral model.

```
external Boolean b1;
external Boolean b2;
```

We then define the time period while the system is in operation. The time period consists of possibly multiple time intervals that start when $b1$ becomes true and end when $b1$ becomes false. The event $b1$ becoming true is denoted by $b1\uparrow$. The event of $b1$ becoming false is denoted by $b1\downarrow$. Thus $b1\downarrow = (\neg b1)\uparrow$. There are actually two time periods of interest to express when the system is in operation:

$P1 = [b1\uparrow, b1\downarrow]$

$P2 = [b1\uparrow, b1\downarrow[$

$P1$ includes the opening and closing events, whereas $P2$ includes the opening event, but excludes the closing event. Note that there are in general several occurrences of $b1\uparrow$ so that $P1$ and $P2$ are composed of multiple time intervals. The set of occurrences of an event is called a clock. Therefore, $b1\uparrow$ and $b1\downarrow$ are clocks.

In general, time periods P are sets of time intervals Δ_i that can overlap. This is denoted by $P = \{\Delta_i\}_{1\leq i\leq n}$. In the sequel, the opening and closing events of a time interval Δ_i are resp. denoted $\Delta_i\uparrow$ and $\Delta_i\downarrow$, and the clocks of opening and closing events of a time period P are resp. denoted $P\uparrow$ and $P\downarrow$.

3.2 Requirements and 4-valued Boolean Logic

The following expression combines condition $b2$ with time period $P1$ to form requirement $R1$ which states that $b2$ should be true at any time instant along $P1$.

$R1 = ensure(b2 \otimes P1)$

$R1$ is a Boolean that is true when $R1$ is satisfied and false when it is not satisfied. The sign $\otimes$ denotes that condition $b2$ is combined with time period $P1$. The precise meaning of $\otimes$ and *ensure* will be given in the sequel.

$R2$ ensures that the number of failures of $R1$ should not exceed 3 over a sliding time period $P3$ of one year, that continuously shifts over the time period while the system is in operation. The failures of $R1$ corresponds to the events $R1\downarrow$. $P3$ is modelled as a time period composed of time intervals of one year that start at each occurrence of $R1\downarrow$. Note that $R1\downarrow$ cannot occur when the system is not in operation. The formal expression of $R2$ states that if a failure occurs at time t, there should not be more than two additional failures before $t+1$ year, and that this condition should be ensured at any time instant t while the system is in operation. Thus, time intervals are not continuously created, but only when they are needed to verify the requirement (i.e., when $R1$ fails).

$P3 = [R1\downarrow, R1\downarrow +1\,year[$

$R2 = ensure(((count(R1\downarrow, P3) <= 3) \otimes P3) \otimes P2)$

$R3$ states that if $R1$ is not satisfied at time instant t, then $R1$ should be satisfied at $t+10$ mn, unless the end of $P1$ occurs before $t+10$ mn.

$R3 = ensure(((\neg R1 \wedge b1) \implies R1 \otimes [R1\downarrow, R1\downarrow +10mn]) \otimes P1)$

Expressions are evaluated at each time instant t. $R3$ is active within $P1$. A new time interval is created at each occurrence of $R1\downarrow$ while $R3$ is active. The purpose of adding $\wedge b1$ to the precondition $\neg R1 \wedge b1$ is to avoid $R3$ being undecided if $P1$ ends before 10 mn after a failure of $R1$ (the meaning of undecided is given in the sequel).

$R4$ is the non-probabilistic version of the final requirement.

$R4 = R1 \wedge R2 \wedge R3$

$R5$ is the final requirement that corresponds to the probability of success of $R4$. The probability is evaluated at $b1\downarrow$ (when the system is stopped). The *checkAtEnd* function evaluates the probabilistic condition at the end of $P1$ (thus at $b1\downarrow$).

$R5 = checkAtEnd((prob(R4, b1\downarrow) > 99.99\%) \otimes P1)$

The reason why the satisfaction of $R5$ is evaluated at $b1\downarrow$, and not before, is because the probability of success of $R4$ is not defined before $b1\downarrow$.

We have so far only used logical functions with some elementary arithmetic to formally express the requirement. The question now is the mathematical type of a requirement and the meaning of $\otimes$. To clarify this, let us temporarily take a simpler example of a requirement R: "The project report must be completed before the end of the project". Before the start of the project, the requirement is *undefined*, which means that the requirement is not applicable. After the start of the project and before the end of the project, the requirement is *undecided* which means that the requirement is applicable, but its outcome is uncertain until either the report is completed before or at the end of the time period (thus before the deadline or just in time), in such case the requirement is *satisfied*, or not completed until the end of the time period, in such case the requirement is *not satisfied* (the report is late or canceled). Therefore, the requirement can take any value in the set $\mathbb{B} = \{true, false, undecided, undefined\}$.

The question now is whether variables of $\mathbb{B}$ comply with the usual Boolean algebra. If the answer to this question is positive, then requirements can be combined using the usual Boolean operators. It is very natural to assign true to the satisfaction of a requirement, and therefore false to the non-satisfaction of a requirement. To find the mathematical meaning of undecided and undefined, let us assume that R is a logical combination of two requirements $R1$ and $R2$: $R = R1\,op\,R2$, where op is a binary logical operation. $R1$ is undefined means that it is not applicable, and that its value should not affect the outcome of the logical operation. Therefore, undefined is a neutral element for any logical operation: $R = R1\,op\,R2 = R2$ for any $R2$ if $R1$ is undefined.

$R1$ is undecided means it is not known whether it is true or false. Therefore, if $R1$ is undecided then $R = R1 \wedge R2$ is false if $R2$ is false and undecided if $R2$ is true or undecided, and $R = R1 \vee R2$ is true if $R2$ is true and undecided if $R2$ is false or undecided. The same argument applies to find the value of $\neg R$: if R is undefined or undecided, then $R = \neg R$. Of course, the standard Boolean algebra applies if $R1$ and $R2$ are true or false.

It is then easy to verify that this algebra verifies the De Morgan's laws (such as $\neg R1 \wedge \neg R2 = \neg(R1 \vee R2)$) and all other Boolean laws (including $\neg(\neg R) = R$), except the complements law (because $R \wedge \neg R = false$ is not verified if R is undecided or undefined). Therefore, this 4-valued algebra can be considered as a Boolean algebra that accommodates all standard logical operators with the usual meaning (with some precaution regarding the complements law). Then the logical implication operator is defined as $R1 \implies R2 \equiv \neg R1 \vee R2$.

The main difference between the 4-valued and 2-valued Boolean algebras is that time is taken into account with the former: the requirement is undefined as long as the time period it is associated with has not begun, and undecided while inside the time period and before the decision can be made whether it is satisfied or not. This event is called the decision event. After the decision event, the requirement is true (satisfied) or false (not satisfied). Let us now assume that the completion of the project report is modeled by a Boolean C that tells whether the report has been signed or not. Then C is a 2-valued Boolean that takes the values true (the report is signed) or false (the report is not signed). Obviously, at a given time instant t, the value of the requirement R can be different from C, as R can take the additional values undefined and undecided. As explained before, the value of R results from a combination of C with the time period P that corresponds to the delay granted for the satisfaction of C. This is denoted $R = C \otimes P$. Therefore, mathematically speaking, a requirement is a function that associates a couple (C,P) to R, where C is the condition of the requirement, P is the time period of the requirement, and R is the value of the requirement. The definition domain of this function can be extended to conditions C that are 4-valued Booleans without any difficulty. It is then possible to formally express

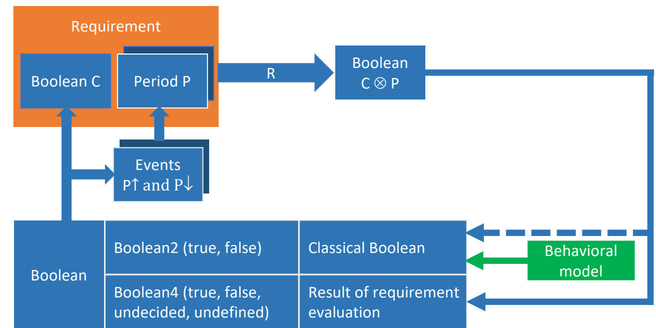

Figure 3. The requirement factory

requirements on requirements, such as if requirement $R1$ fails, then requirement $R2$ should be satisfied within a given time delay $P : R = R1 \wedge (\neg R1 \implies R2 \otimes P)$. Fig. 3 summarizes how requirements are built.

We are now interested in giving a formal definition for the value of $C \otimes P$. Let us consider first a time period $P = \{\Delta_1\}$ composed of a single time interval Δ_1. The meaning of $R = C \otimes P$ is that the decision whether C is satisfied or not is made as soon as possible within Δ_1, i.e., at the decision event. The decision event for $C \otimes P$ is denoted $\delta(C,\Delta_1)$. From the decision event, if C is true or false, then $C \otimes P$ is true or false (it cannot be undefined or undecided) and keeps its value until the next decision event if any. For instance, if the condition C is that the number p of events $e\uparrow$ within Δ_1 should be less than a fixed integer n, i.e. if $C \equiv (p = count(e\uparrow,\Delta_1) \leq n)$, then $\delta(C,\Delta_1) = (p > n)\uparrow \vee \Delta_1\downarrow$ where $a\uparrow \vee b\uparrow$ denotes the clock containing the occurrences of events $a\uparrow$ and $b\uparrow$: $C \otimes P$ is false as soon as p is larger than n within Δ_1, otherwise $C \otimes P$ is true at the end of Δ_1. For the condition $C \equiv (p > n)$, the same decision event applies but the outcome is different: $C \otimes P$ is true as soon as p is larger than n within Δ_1, otherwise $C \otimes P$ is false at the end of Δ_1. Note that because the counter starts from $p = 0$, while $p \leq n$, C is false but $C \otimes P$ is undecided, thus in general $C \otimes P \neq C$. Note that when writing $(C,\Delta_1) = a\uparrow \vee b\uparrow$, there are in fact two decision events, $a\uparrow$ and $b\uparrow$. Unless it is something desirable, one must be careful that if $b\uparrow$ occurs after $a\uparrow$, the decision over $C \otimes P$ made at $a\uparrow$ is not reversed when $b\uparrow$ occurs. If the decision over C can be made at any time instant within Δ_1, then $\delta(C,\Delta_1) = (C \vee \neg C)\uparrow \vee \Delta_1\downarrow$. This is the case if C is the satisfaction of another requirement R' (i.e., if $C \equiv R'$): $C \otimes P$ is true or false as soon as C becomes true or false (if the decision over the satisfaction of a requirement is not reversed). If the decision over C cannot be made before the end of Δ_1, then $\delta(C,\Delta_1) = \Delta_1\downarrow$.

The function $checkAtEnd(C \otimes P)$ used in the example means that $\delta(C,\Delta_1) = \Delta_1\downarrow$. $ensure(C \otimes P)$ means that C must be true all along Δ_1, thus that it should never be false within Δ_1. This can be expressed as $ensure(C \otimes P) = (count(C\downarrow,\Delta_1) <= 0) \otimes P \wedge C \otimes [\Delta_1\uparrow,\Delta_1\uparrow]$. Within P, the value of the requirement is undecided until the condition is not verified, in such case it turns to false and stays false,

DOI
10.3384/ecp204497

or until the end of P if the condition is always satisfied, in such case it turns to true.

To evaluate $C \otimes P$ at time instant t, we consider two temporal Boolean operators:

- A filter $\times$ with the following properties: $a \times b = b$ if a is true, $a \times b = undecided$ otherwise.
- An accumulator $+$ with the following properties: $a + b = b + a$; $a + a = a$; $undefined + a = a$; $undecided + a = a$ if a is true or false; $true + false = false$;

The filter filters out all events that are not decision events. The accumulator computes the value of $C \otimes P$ at instant t taking into account the history of $C \otimes P$. It ensures that if C is false after the decision event, $C \otimes P$ will stay false whatever the future values of C. For a fixed integer i and $P = \Delta_i$, the value of $C \otimes P$ at t is

$$C \otimes \{\Delta_i\}(t) = undefined + \int_{\tau=\Delta_i\uparrow}^{min\{\Delta_i\downarrow,t\}} \delta(C,\Delta_i)(\tau) \times C(\tau) d\tau$$

The integral operator accumulates all values of the Boolean integrand for all time steps $d\tau$ within Δ_i until t. For t occurring before $\Delta_i \uparrow$, the value of $C \otimes \{\Delta_i\}$ is undefined. For t occurring after $\Delta_i \uparrow$, the value of $C \otimes \{\Delta_i\}$ is undecided until the integrand is true or false. This equation is generalized to a multiple time period $P = \{\Delta_i\}_{1 \le i \le n}$ by taking the logical conjunction of all $C \otimes \{\Delta_i\}$:

$$C \otimes P(t) = \bigwedge_{i=1}^{n} C \otimes \{\Delta_i\}(t)$$

This equation states that the condition must be satisfied for all time intervals of the time period. Then, provided that for any condition C and time period P, we know how to express $\delta(C,\Delta_i)$, we can evaluate $C \otimes P$, and consequently we can evaluate any temporal CRML expression. The operator $\otimes$ is expressed using elementary CRML temporal operators on 4-valued Booleans that are implemented in the ETL library using truth tables. This is why there is no built-in operator in CRML for $\otimes$, and also no built-in type for requirements. The operator $\otimes$ is used in high-level operators such as 'while' 'check count', 'while' 'ensure' or 'while' 'check at end' that express different kinds of decision events and constitute the FORM-L library. Such high-level operators are used in Section 3.5. It is possible to have user-defined types for requirements derived from the built-in type Boolean. Types are introduced in Section 3.3, and operators are introduced in Section 3.4. The following sections detail these constructs with the CRML syntax.

3.3 Types

User-defined types can be created from built-in types. For instance, the types Requirement and Assumption can be created from the type Boolean.

```
type Requirement is Boolean;
type Assumption is Boolean;
```

User-defined types can be used to define physical or monetary units, the syntax of which is not detailed here. They enable users to write expressions involving units such as

```
Pressure P is 1 bar + 1 mbar;
```

The value of P is computed in the unit system defined by the type Pressure. An error is raised if the unit is wrong or omitted.

3.4 Operators and Sets

Anything in CRML is a set or an element of a set. An element of a set has a fixed value, or a value returned by an operator. Therefore, the CRML syntax is entirely based on the notions of sets and functions. A CRML operator is a standard mathematical function. Two syntaxes are possible: the traditional mathematical syntax and the natural language syntax. In the natural language syntax, the names of user-defined operators are divided into snippets enclosed between singe quotes, and the input arguments are placed in front or after each snippet. Names of built-in operators or user-defined operators in the mathematical syntax are not enclosed within quotes. As an example, we define two operators using the natural language syntax: one that generates the set of occurrences of a Boolean becoming true (this set is a clock), and one that defines the logical implication operator (the keyword *Template* can be used when all input and output arguments are Booleans).

```
Operator [ Clock ] Boolean b 'becomes true'
     = Clock b;
Template b1 'implies' b2 = not b1 or b2;
```

The above operators are invoked as follows:

```
Clock c is b 'becomes true';
Boolean b3 is b1 'implies' b2;
```

A CRML set is a standard mathematical set. It is possible to apply unary operators to a set. This amounts to applying these operators to all elements of the set as follows:

```
Boolean {} b is { n1, n2, n3, n4 } < p;
```

is equivalent to

```
Boolean b {} is { n1 < p, n2 < p, n3 < p,
    n4 < p };
```

In this example, the unary operator is $x \mapsto x <$ p. It is also possible to apply a binary operator to a set:

```
Boolean b is and { n1 < p, n2 < p, n3 < p,
    n4 < p };
```

is equivalent to

```
Boolean b is n1 < p and n2 < p and n3 < p
    and n4 < p;
```

It is possible to construct subsets by filtering set elements depending on their properties, as defined for instance in classes of objects.

3.5 Classes and Objects

CRML is equipped with the notion of class in order to express requirements on complex objects, which can be physical subsystems like cooling or heating systems, physical components like vessels, pumps or heat exchangers, or abstract objects that capture generic notions. For the sake of the example, we define the abstract class Equipment that carries requirement R5, which is rewritten using the CRML syntax.

```
partial class Equipment is {
  external Boolean b1;
  external Boolean b2;
  Integer n is 3;                  // Default
      value
  Time dt is 10 mn;      // Default value
  Periods P1 is [b1 'becomes true', b1 '
      becomes false'];
  Periods P2 is [b1 'becomes true', b1 '
      becomes false'[;
  Periods P3 is [b1 'becomes true' or R1 '
      becomes false', 1 year[;
  Boolean R1 is 'while' P1 'ensure' b2;
  Boolean R2 is 'while' P2 'ensure'
    ('while' P3 'check count' (R1 'becomes
        false') '<=' n);
  Boolean R3 is 'while' P1 'ensure'
    ((not R1 and b1) 'implies' ('while' [R1
        'becomes false',
    R1 'becomes false' + dt] 'check' R1));
  Boolean R4 is R1 and R2 and R3;
  Requirement R5 is 'while' P1 'check at
      end'
    (('probability' (R4) 'at' b1 'becomes
        false') > 99.99%);
};
```

A partial class cannot be instantiated because it carries partial information that is not sufficient to instantiate objects. Class Equipment is partial because it is not possible to provide values for b1 and b2 without having some knowledge about the type of the equipment. However, requirement R5 will be automatically applied to all instances of the classes derived from Equipment that are not partial. Let us now create a new class Pump that derives from class Equipment. Class Pump will inherit all attributes of Equipment. It is however possible to redefine (or redeclare) within class Pump the attributes of Equipment that are not suitable to pumps. The operational domain for a pump corresponds to the non-cavitation of the pump: pumps should never cavitate while they are in operation. This can be enforced by redeclaring attribute n to be a constant integer equal to zero in the class Pump definition.

```
class Pump is {
  redeclare n constant Integer n is 0;
  redeclare R5 Requirement no_cavitation;
} extends Equipment;
```

In the above statement, requirement R5 is renamed to no_cavitation for better legibility. Then pumps can be instantiated with the statement

```
Pump pump is Pump ();
```

The following statement expresses that all pumps in the system should never cavitate, no matter the number or types of the pumps. The fact that class System extends Equipment means that requirement R5 is applicable to the system as a whole (provided that the values of b1 and b2 for the whole system can be obtained from a behavioral model).

```
class System is {
  Pump {} pumps;
  Requirement no_cavitation is and pumps.
      no_cavitation;
} extends Equipment;
```

In the above statement, the global no-cavitation requirement for all pumps in the system is obtained by taking the logical conjunction of the no-cavitation requirement for all individual pumps, that can be of different nature, and do not need to be known when this statement is written. Class extension and attribute redefinition can be used in combination to add and refine requirements in the course of system design. For instance, if during detailed design the chosen type for pumps is centrifugal, and if centrifugal pumps come with their own set of requirements, then a new class DetailedSystem can be derived from the class System that redefines the set of objects from class Pump to be a set of objects from a new class CentrifugalPump that extends class Pump and that carries the additional requirements for centrifugal pumps.

4 Implementation of the CRML toolchain

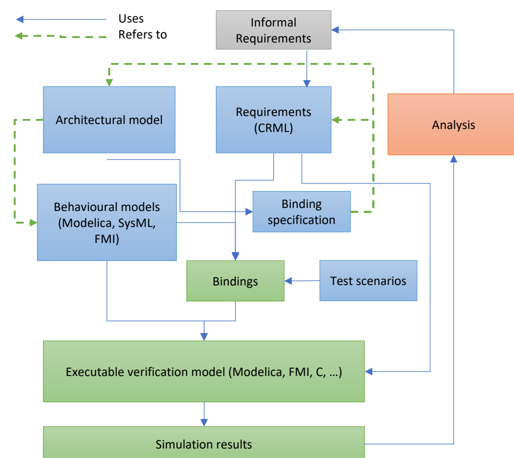

Figure 4. Architecture of the verification model

As illustrated in Fig. 4, the requirements are connected to the architectural and behavioural models through bindings. The compiler developed in the project translates CRML to Modelica. This also enables the use of Functional Mockup Itnerfaces (FMI) - a standard for exporting models for co-simulation supported by many tools. This means that any behavioural model that can be exported as

DOI
10.3384/ecp204497

a Functional Mockup Unit (FMU) can be connected to the requirements model.

4.1 CRML to Modelica Compiler

The compiler consists of two parts, first a grammar is specified using ANTLR and this is used to generate a parser for the compiler in Java. This grammar can also be used to generate parsers for compilers in other languages and can serve as a more formal specification of the language syntax.

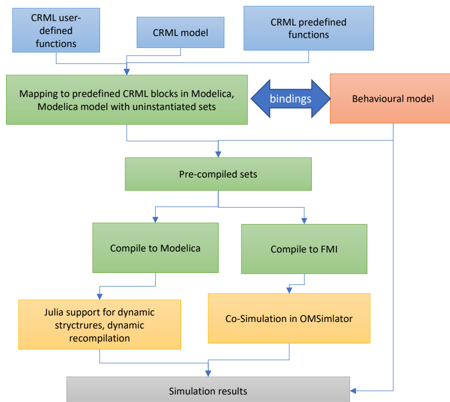

Figure 5. Architecture of the CRML to Modelica compiler

In a second step, the generated abstract syntax tree (AST) is then traversed to generate Modelica. Many elements of the language can be translated in a straightforward way, but in this section we will focus on the main CRML elements that require special consideration (Fig. 5).

4.1.1 4-valued Booleans

In Modelica booleans are two valued, therefore all references to booleans and boolean operators need to be converted to a special Modelica type Boolean4. To simplify, the definitions of this type and of the fundamental logical operators are given in a library and then calls to this library are directly generated (Fig. 5).

```
import CRML.ETL.Types.Boolean4;
model Bool1
  Boolean4 b0 = CRML.ETL.Types.Boolean4.
      true4;
  Boolean4 b1 = CRML.ETL.Types.Boolean4.
      false4;
  Boolean4 b2 = CRML.ETL.Types.Boolean4.
      undecided;
  Boolean4 b4 = CRML.Blocks.Logical4.and4(
      b1,b2);
end Bool1;
```

4.1.2 Time dependent functions, templates and operators

Another big difference in CRML and Modelica is that in CRML functions can be dependent on time while in Modelica they cannot. Therefore CRML functions need to be mapped to Modelica blocks. Built-in functions are mapped to the predefined blocks in the CRML library and user-defined Operators and Templates generate new Modelica blocks.

For example the user defined

```
Template R1 'xor' R2 = (R1 'or' R2) and not
     (R1 and R2);
```

is translated to the following Modelica snippet:

```
import CRML.ETL.Types.Boolean4;
model "xor"
  input Boolean4 R1, R2;
  output Boolean4 out;
  "or" "or0"(R1 = R1,R2 = R2);
equation
  out = CRML.Blocks.Logical4.and4(_or0.out,
        CRML.Blocks.Logical4.not4
          (CRML.Blocks.Logical4.and4(R1,R2)
              ));
end "xor";
```

And the call to the function is translated to an instantiation of the corresponding block and corresponding connectors. Quotation marks are used around operator names to distinguish between user defined keywords in CRML and Modelica keywords.

4.1.3 Sets

CRML is built around the concept of sets which are not present in Modelica. We distinguish between event sets that change in size throughout the execution of the program, and object sets that are either statically specified or calculated during the binding process.

Object sets are translated to arrays that are instantiated during the binding. This can either be done in a semi-automated manner or manually depending on whether the behavioural model is in Modelica or FMI.

Since the number of events can potentially be unlimited, event sets need to be mapped to dynamic structures, also known as Variable Structured Systems that increase in size as needed.

We have developed OpenModelica.jl (Tinnerholm et al. 2021), a modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl and supporting Variable Structured Systems. We extend the Modelica language with several new operators to support continuous time mode-switching and reconfiguration via recompilation at runtime. Our compiler supports the Modelica language as well as these aforementioned extensions.

A special type, a dynamic event array is defined that relies on the recompilation primitive to grow (or shrink the array size at runtime). This model is extended to store specific events.

```
partial model EventArray
  Event events[N];
  parameter Integer N=10; // size
  Integer i (start = 1); // index
equation
  when (i = N) then
    recompilation(N, N + 100);
  end when;
```

```
end EventArray;
```

4.2 Integrating CRML with Architectural and Behavioral Models

4.2.1 The Binding Mechanism

The most obvious way to bind variables is to fill in their full pathnames in a connect statement. However, this requires a large amount work from the user if a large number of external variables are involved. Also, if the models that provide the values are restructured, the pathnames must be changed accordingly, inducing work overhead. To simplify the work of binding, the idea is to prepare as much as possible binding at the class level. Manual binding is then done at the instance level by connecting together objects instead of variables. The final binding of individual variables is performed automatically from the information provided at the class and instance levels. As information provided at the class level can be reused for any model, the amount of manual work to be done for a given model is on average divided by the number of objects carrying the external variables. It is also easier to manually bind objects than variables because objects are much easier to spot than variables. Therefore, it always requires less effort to bind objects than variables, even in the worst case of having no more than one external variable per object.

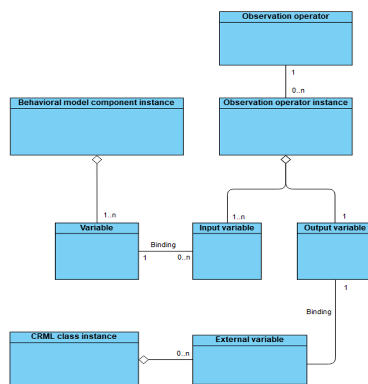

Figure 6. Binding output information

The output information to be produced to bind external variables of the requirement model to variables of the behavioral models is given in Fig. 6. There are two kinds of variable bindings: (1) the connections of the input variables of the observation operators instances to the variables of the behavioral models, and (2) the connections of the output variables of the observation operators instances to the external variables of the requirement model.

The information to be provided by the user in order to automate binding as much as possible is given in Fig. 7. It shows that the only information to be provided at the instance level is the correspondence between the objects in the requirement models and the objects in the behavioral or architectural models. The only assumption that makes this procedure possible is that the behavioral and architectural models are expressed using an object-oriented methodology, where objects are instances of classes. No assumption is made on the way connections between variables are expressed. Binding input and output information can be provided in a relational database. Connections between variables can be expressed using Modelica statements for white-box Modelica behavioral models, or FMI statements for black-box behavioral models equipped with an FMI interface.

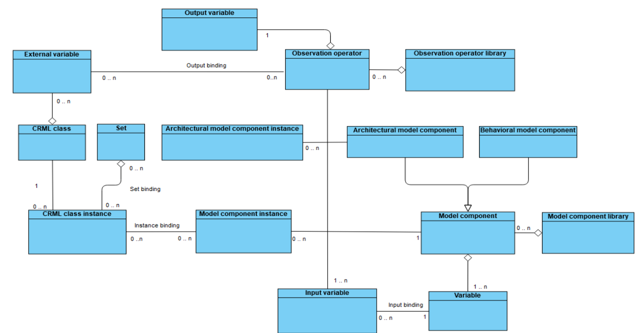

Figure 7. Automatic binding input information

4.2.2 Bindings to Behavioral Models

For Modelica behavioral models, the CRML model is automatically translated to a Modelica model by the CRML compiler. If the behavioral model is a Modelica model, then it is possible to have bindings using Modelica statements. Then the verification model (cf. Fig. 1) can be automatically generated as a Modelica model from the binding output information (cf. Fig. 6) (which itself can be automatically generated from the binding input information).

For black box models we can use the System Structure and Parametrization standard (SSP) to connect to models defined in other formalisms provided they can also be exported as FMUs as illustrated in Fig. 4. SSP connections can be generated automatically or manually based on binding specifications.

One difference with when generating bindings for FMUs is that there is no notion of classes in FMI, only simple types. Therefore it is impossible to automatically calculate sets of object of a certain type and they need to be specified manually.

4.2.3 Bindings to Architectural Models

If the architecture model is given in SysMLv2, bindings can also be defined in SysMLv2. The idea (shown below on a simplified example) is to use CRML as the constraint language to formalize requirements over the *attributes of the requirements element*. These attributes can be bound to attributes of the subject in specialized requirements, essentially defining the binding between variables of the architecture and of the requirement. A compiler can then generate observation operators and requirements models based on this information whenever the SysML model

DOI
10.3384/ecp204497

is translated to a computable model for verification (e.g. Modelica).

4.2.4 Execution of verification models

Having the requirements defined separately and bound through a technology-agnostic syntax, means that different approaches of verifying the requirements agains Modelica or FMU models are possible. For instance probabilistic requirements can be verified through Monte-Carlo simulation, ensuring for example that failure rates are below a certain threshold. Currently, test scenarios are generated by hand but in the future, generating test values based on the behavioural envelopes specified by the requirement models will also be investigated.

```
requirement def SpeedLimiterReq {
  attribute speed : SpeedValue;
  attribute limit : SpeedValue;
  attribute on : Boolean;
  require constraint { language "CRML" /*
      during on ensure speed < limit; */ }
}
requirement <r1> citySpeedLimiter :
    SpeedLimiterReq {
  subject vehicle : Vehicle;
  attribute :>> speed = vehicle.
      currentSpeed;
  attribute :>> limit = 50[km/h];
}
```

5 The Intermediate Cooling System Use Case

As explained in Section 1, the goal of the CRML initiative is to support the engineering of complex CPS that are very often over-constrained by a set of numerous (even conflicting) requirements. The idea is to rely on a shared evaluation toolset that helps stakeholders find the best trade-offs and foster innovative solutions. The CRML language and its underlying methodology make the traditional V-model fully executable and should act as an enabler for taking appropriate decisions at each step of engineering projects. Inspired from (Azzouzi et al. 2022), the methodology should be seen as an iterative approach whose main steps are summarized in Fig. 8.

Nuclear power plants are divided into approx. 200 subsystems. One of them is called the Intermediate Cooling System (ICS). The goal of this section is to illustrate how CRML can help justifying that the ICS properly fulfills its missions.

Step 0: Identification of System's Missions The ICS missions sum up to:

- **Evacuate the heat produced by a served systems** when they are in operation (served systems are auxiliary equipment such as the alternator or pumps);
- With the **use of demineralized water** (because water flowing directly from the cold source -sea or river- could damage equipment);
- At an **acceptable availability** rate (the plant must be shut down if the ICS is unavailable).

Step 1: Formalize the System's Environment and Its Interfaces To fulfill its missions, the ICS should physically interact with the different served systems to be cooled but also with (Fig. 4):

- A source of demineralized water (SED) to provide "clean" water to the ICS;
- A source of cold water (SEN) to cool the ICS itself (here a sea or a river);
- A drain sewer (SEK) in case of ICS leaks;
- A means to communicate with the plant operator (OP).

Although the ICS is not a large system, it involves 9 stakeholders coming with their own data and requirements.

For the engineer in charge of the ICS design, this makes his/her work even more challenging as it multiplies the information channels, the sources of potential changes during the project and the types of verification studies he/she needs to produce (i.e. one stakeholder being interested only by the achievements of its own goals). Formalizing the different constraints using CRML appears as a means to gather all these sources of data in a more rigorous and reproductible way than relying on a one-person expertise. It also enables the automation of verification tasks and hence provides more flexibility when input data changes and design studies must be rerun to assess the impact on the current solutions.

This formalization step should be performed for each system (or stakeholder) in interface with the ICS. For the sake of conciseness, let us focus on two different interfaces: the one between the ICS and the served system and the one between the ICS and the cold-water source.

Contract between the ICS and the served systems Each served system should have a physical interface with the ICS to be cooled by its refreshed demineralized water. In practice, there should be some physical means such as a valve at the inlet of a heat exchanger to "activate" cooling when necessary. In order to leave as many design options open as possible, we set the expectations on the served systems as follows: The cooling service should be provided as soon as the minimum temperature is reached (because too cold water could damage equipment). Requirements being common to all served systems, they are modelled in a generic class "Served_system" as follows:

```
class Served_system is {
  [...]
  // Mission: When the served system is
     operating, all the heat produced
     should instantly be evacuated by the
     ics
```

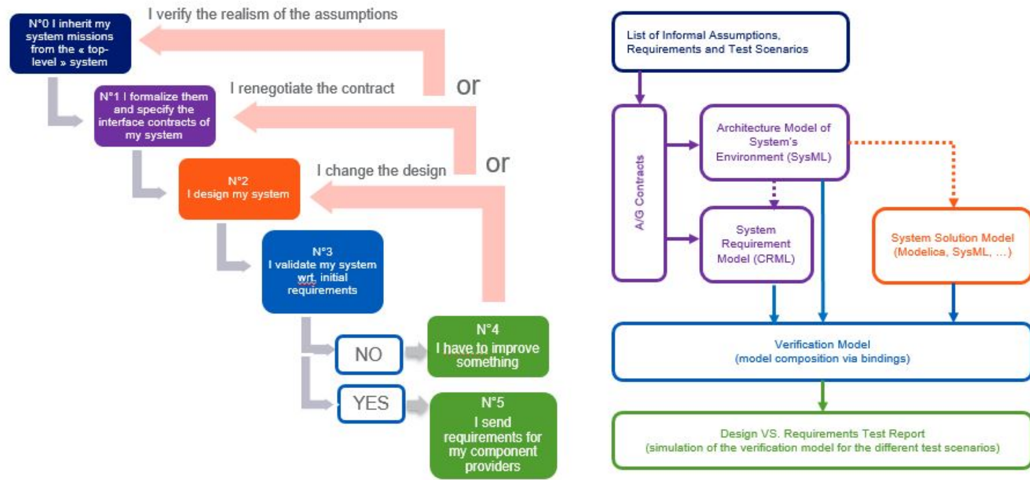

Figure 8. The main steps of the CRML methodology (left) leading to a structured set of models (right)

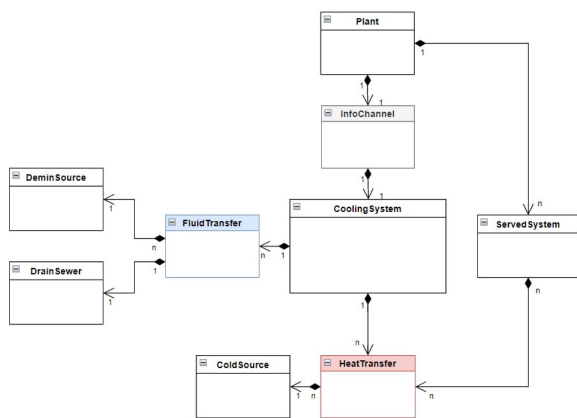

Figure 9. Architecture model of the ICS environment

```
constant Real epsilon is 0.001; //
    tolerance
Boolean cool_served_system_0 is
  during inOperation ensure 0 <= ics.W -
      W <= epsilon
// Requirement: The served system should
    not be supercooled by the ics (to
    avoid thermal stresses).
Boolean cool_served_system_1 is
  during ics.tRW <= (tRWMin - 0.1 Celsius
      ) ensure not open; // 0.1 Celsius
      is a design margin (tolerance)
// Requirement: The served system should
    be cooled by the ics as soon as the
    ics water temperature is above the
    minimum acceptable.
Boolean cool_served_system_2 is
  during ics.tRW >= tRWMin ensure open;};
```

"inOperation" is a Boolean indicating whether the served system is operating, "W" is the power of the served system, "tRW" is the ICS temperature, "tRWMin" is the minimum temperature acceptable by a served system, "open" is a Boolean stating whether the cooling service should be provided. These requirements are then automatically instantiated when the different served systems are themselves defined and parameterized.

Contract between the ICS and the cold-water source The ICS should have a physical interface with the cold source such as sea or river. To avoid environmental damages on the fauna and flora, regulations are enacted by a local authority regarding thermal effluents. Failure to comply with the regulations forces the plant to be shut down. In CRML, these requirements are modelled as follows:

```
class Cooling_system is {
 [...]
 // Requirement: When the ICS is operating,
     the temperature of the cold water
    source should
 // be below its acceptable maximum (no
    overheat of the sea or river).
 Boolean coldW_ics_1 is
   during not (state_stopped or
       state_stopping)
   ensure sen.tCW <= sen.tCWMax;
 // Requirement: When the ICS is operating,
     the temperature increase of the cold
    water source should
 // be below its acceptable maximum.
 Boolean coldW_ics_2 is
   during not (state_stopped or
       state_stopping)
   ensure tWW < (sen.tCW + sen.deltaTMax);
       };
```

"state_stopped" and "state_stopping" are Booleans stating whether the ICS is respectively stopped or in a stopping state, "tCW" is the temperature of the cold-water source, "tCWMax" is the maximum acceptable temperature of the cold-water source, "deltaTMax" is the maximum increase of temperature of the cold-water source.

Step 2: System Design After having identified and formalized the different requirements and constraints, the ICS engineer imagines some design alternatives. Fig. 10 represents one possible solution modeled in Modelica

with the use of the open source ThermoSysPro[1] library.

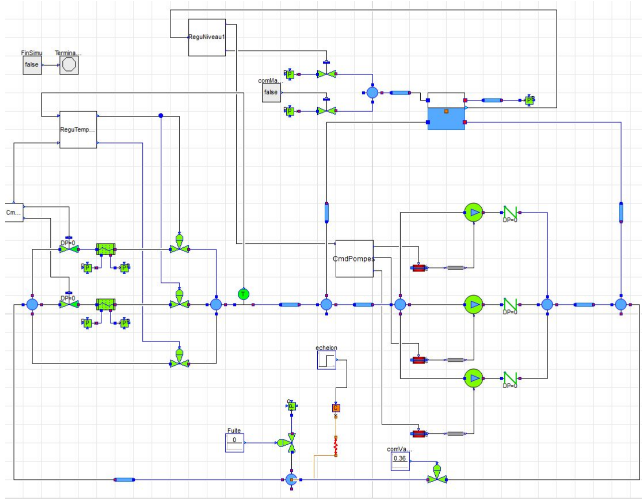

Figure 10. Behavioral model of the ICS

During this step, the ICS engineer could add some specific performance goals such as maintaining the circuit temperature around a setpoint (here 17 ° C) to optimize thermal transfer in heat exchangers. In CRML, this is expressed by adding the following requirement in the "CoolingSystem" class:

```
class Cooling_system is {
  [...]
  // Kpi: When operating, ics temperature
     should be around 17 Celsius
  constant Temperature tolerance is 0.5
     Celsius;
  Boolean kpi_1 is
    during inOperation
    ensure tRW >= (17 Celsius - tolerance)
       and tRW <= (17 Celsius + tolerance)
       ;
};
```

Step 3: System Verification Using the concept of bindings, the ICS engineer is then able to compose the different models to build a verification model. The purpose is to check by simulation whether the requirements are satisfied.

Fig.11 shows the simulation results obtained to monitor the 4-valued Boolean variable "kpi_1". The green curve is the evolution of "tRW" given by the behavioral model. "kpi_1" is computed by the CRML model. One can easily see that the "kpi_1" is not achieved (is false) as soon as "tRW" goes beyond its tolerance interval.

Although this seems like quite a simple performance target, it involves the dynamics of the system, so that possible violations of constraints are difficult to spot and interpret by visual inspection of continuous curves such as "tRW". The final verdict given by "kpi_1" alleviates this difficulty.

[1]URL: https://www.thermosyspro.com/

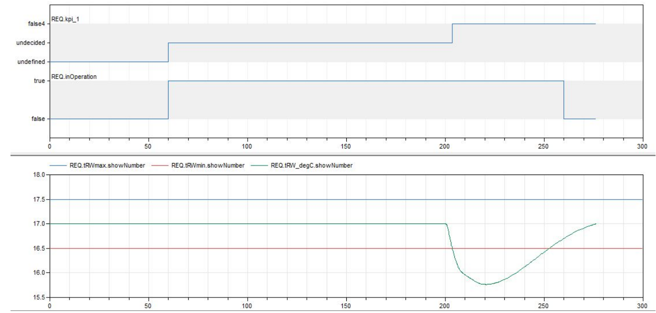

Figure 11. Simulation results of the verification model to monitor "kpi_1"

Step 4: Design Report Test reports can be customized. Below is an example that filters the requirements associated to the served systems only.

```
model GlobalReports is {
  // Conjunction of all requirements on
     Served_system
  Boolean globalReport_ForServedSystems is
      and flatten filter
    (flatten filter sriRequirements
      (type element == Served_system)) (
         type element == Boolean);};
```

6 Conclusion and Future Work

A new formal language called CRML (Common Requirement Modeling Language) for the modeling and simulation of requirements has been presented. The goal is to release CRML as an international standard interoperable with other standards such as SysML, Modelica, FMI and SSP. The purpose of CRML is to enable different stakeholders in different disciplines in charge of the design and operation of complex cyber-physical systems to reach a formal common agreement in terms of contracts made of formal constraints, so that they can successfully build, modify and operate such systems.

The salient innovative features of the language are its ability to capture all possible constraints on the real system, including real-time dependent constraints complemented with probabilities for failure, because no requirement can be fulfilled at any time at any cost (the lower the probability for failure, the higher the cost of the system). The language includes the possibility to structure the system using objects as instances of classes and build libraries of standard requirements that can be customized when used to express constraints on particular systems. A formal definition for the satisfaction of requirements has been given that uses a 4-valued Boolean algebra. This algebra provides the framework to combine requirements together to form high-level generic constraints that can be stored in libraries for further reuse.

Design can be verified against requirements by coupling, via so-called bindings, requirements models with behavioral models that capture the dynamic behavior of the physical system under study. To that end, a CRML to

Modelica compiler is being developed in the OpenModelica framework that automatically produces the executable verification models from the CRML models and the binding specifications (i.e., the way to associate the variables to be monitored in requirements to the variables that represent the states of the physical system).

Ways to use CRML within SysML v2 are being explored. It is believed that CRML will efficiently bridge the semantic gap between SysML and physical modeling and simulation, so that digital twins will be more efficiently used for the engineering and operation of cyber-physical systems. The reason is that CRML provides a formal way to translate functional concepts embedded in requirements models to the various concepts (such as physical concepts) embedded in behavioral models.

The way to use CRML for the design of a subsystem of a nuclear power plant has been presented. Future work will essentially bear on providing a graphical design methodology that will enable designers to specify systems without explicitly writing CRML code. Such graphical work will also improve the understandability of CRML by non-experts and empower the spread of CRML across different engineering teams.

Acknowledgements

This work has been supported by the ITEA3 EMBrACE project, as well as National Research, Development and Innovation Fund of Hungary, financed under the [2019-2.1.1-EUREKA-2019-00001] funding scheme.

References

Alur, Rajeev (1999). "Timed automata". In: *International Conference on Computer Aided Verification*. Springer, pp. 8–22.

Azzouzi, Elmehdi et al. (2022). "A Model-Based Engineering Methodology for Stakeholders Coordination of Multienergy Cyber-Physical Systems". In: *IEEE Systems Journal* 16.1, pp. 219–230. DOI: 10.1109/JSYST.2021.3071231.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of model checking*. MIT press.

Bouskela, Daniel and Audrey Jardin (2018). "ETL: A new temporal language for the verification of cyber-physical systems". In: *2018 Annual IEEE International Systems Conference (SysCon)*, pp. 1–8. DOI: 10.1109/SYSCON.2018.8369502.

Bouskela, Daniel, Thuy Nguyen, and Audrey Jardin (2017). "Toward a rigorous approach for verifying cyber-physical systems against requirements". In: *Canadian Journal of Electrical and Computer Engineering* 40.2, pp. 66–73.

Henzinger, Thomas A (2000). "The theory of hybrid automata". In: *Verification of digital and hybrid systems*. Springer, pp. 265–292.

Kanso, Bilal and Safouan Taha (2013). "Temporal Constraint Support for OCL". In: *Czarnecki K., Hedin G. (eds) Software Language Engineering. SLE 2012. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg.* 7745.

Mattsson, Sven Erik, Hilding Elmqvist, and Martin Otter (1998). "Physical system modeling with Modelica". In: *Control Engineering Practice* 6.4, pp. 501–510.

Nguyen, Thuy (2014-03). "FORM-L: A MODELICA Extension for Properties Modelling Illustrated on a Practical Example". In: *Proceedings of the 10th International Modelica Conference*. Linköping Electronic Conference Proceedings. Lund, Sweden: Modelica Association and Linköping University Electronic Press, pp. 1227–1236. DOI: 10.3384/ecp140961227.

Nguyen, Thuy (2019). "Formal Requirements and Constraints Modelling in FORM-L for the Engineering of Complex Socio-Technical Systems". In: *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pp. 123–132. DOI: 10.1109/REW.2019.00027.

OMG (2017). *Unified Modeling Language (UML) version 2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 2023-05-13).

OMG (2019). *Precise Semantics of UML State Machines (PSSM)*. formal/19-05-01.

OMG (2021). *SysML Extension for Physical Interaction and Signal Flow Simulation (SysPhS)*. formal/21-05-03.

OMG (2023). *Systems Modeling Language (SysML) version 2*. URL: https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/2-OMG_Systems_Modeling_Language.pdf (visited on 2023-06-08).

Otter, Martin et al. (2015-09). "Formal Requirements Modeling for Simulation-Based Verification". In: *Proceedings of the 11th International Modelica Conference*. Linköping Electronic Conference Proceedings. Versailles, France: Modelica Association and Linköping University Electronic Press, pp. 625–635. DOI: 10.3384/ecp15118625.

Tinnerholm, John et al. (2021-09). "OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl". In: *Proceedings of the 14th International Modelica Conference*. Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, pp. 109–117. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181109.