Model-Based Design Languages: A Case Study

Ivan Cibrario Bertolotti
National Research Council of Italy – IEIIT
c.so Duca degli Abruzzi 24, I-10129 Torino, Italy
Email: ivan.cibrario@ieiit.cnr.it

Tingting Hu, Nicolas Navet
University of Luxembourg – FSTC
6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
Email: {tingting.hu, nicolas.navet}@uni.lu

***Abstract*—Fast-paced innovation in the embedded systems domain puts an ever increasing pressure on effective software development methods, leading to the growing popularity of Model-Based Design (MBD). In this context, a proper choice of modeling languages and related tools—depending on design goals and problem qualities—is crucial to make the most of MBD benefits. In this paper, a comparison between two dissimilar approaches to modeling is carried out, with the goal of highlighting their relative advantages and shortcomings. It focuses on a case study involving a well-known distributed agreement protocol, a choice motivated by the fact that embedded systems are nowadays quickly evolving towards distributed, fault-tolerant architectures.**

***Index Terms*—Model-driven development, Distributed agreement protocols, Embedded systems design and development.**

I. Introduction and related work

In recent years embedded systems permeated every aspect of life, placing software quality and the whole software development process under scrutiny even more than in the past. This process was once characterized by several distinct phases, often linked by means of an *informal* information flow, and was shown to be prone to errors, especially when system complexity grows. By contrast, the Model-Based Design (MBD) methodology stipulates that requirement analysis and design shall be performed by building a model of the system by means of a *formal language*. Ideally, the information contained in this model should then flow through all software development phases, ensuring their mutual consistency. However, as the variety of MBD-oriented languages grows, it becomes important to carefully contemplate their similarities and differences, in order to make an informed choice, depending on the design problem at hand. Even more importantly, the comparison must be done not only from a language theorist's point of view—but also by referring to case studies of *practical* interest.

General purpose modeling languages such as the Unified Modeling Language (UML) are suitable for requirement specification and generally meant for software system design alone. As a result, they do not provide sufficient facilities to support the modeling of hardware, which also plays an essential role in embedded systems. Matlab/Simulink® is a successful commercial model-based design tool. It is extremely powerful since its development environment not only allows modeling and simulation but also permits requirement traceability, code generation, as well as test case generation. However, it just focuses on the functional behavior of embedded systems. The Architecture Analysis and Design Language (AADL) [1], Chariot [2], as well as EAST-ADL are full-fledged Architecture Description Languages (ADL) which can be used to describe complex system architectures, covering both software and hardware aspects, and in the meantime deal with non-functional objectives. Furthermore, synchronous languages [3], including Lustre, Signal, and Esterel, are designed for safety-critical systems and generally offer adequate support for formal proof that eases the process of verification and certification. However, most of them share the same limitation as for architecture description languages, namely another language is still needed for the real implementation. In addition, most often they impose a very specific programming style that makes the initial learning curve unnecessarily steep for beginners. On the contrary, both Promela [4] and CPAL [5] build around C-like syntax, while in the meantime they provide useful constructs for convenient illustration of concepts particular to embedded systems. Table I summarizes their main features and more details will be provided in the next sections.

TABLE I
Main Features of Promela and CPAL Compared

Feature	Promela	CPAL
Concurrency model	— Multiple processes —	
Verification technique	**State space exploration** towards formal proof	Simulation and Schedulability analysis
Scheduling	All interleavings	**Specific algorithms**
Control flow	**Non-deterministic**	Sequential
Concept of time	Modeled explicitly	**Native**
Process structure	Free Format	Finite State Machine
Synchronization	Statement executability	Explicit
Communication	— Shared memory	and channels —
Model execution	Translator	**Interpreter**
Input-output	—	**I/O ports**

The paper is organized as follows. Section II outlines the Promela and CPAL languages, while Section III describes how the communication protocol used as a case study has been modeled in the two cases. Section IV highlights the most important considerations emerged during model development.

II. Model-Based Design languages

A. Promela and The SPIN Model Checker

SPIN [4] is one of the most prominent *model checkers*, aimed at the formal verification of complex concurrent systems. Model checkers have as input a model of the system under analysis and one or more properties to be verified, both specified by means of a formal language. In the case of SPIN

the modeling language is Promela and properties are specified with Linear Temporal Logic (LTL) formulas [6]. Informally speaking, SPIN's goal is to prove that the properties of interest hold by exploring every possible state the system can reach along its execution. Possible computation steps, leading from one state to another, are specified by the model.

A large number of possible sequences of steps, and hence, system states are usually possible, leading to the well-known *state space explosion* issue. Although research in model checking was able to devise fully automatic model optimization techniques, keeping state space explosion under control may still require designers to artificially ply the model. This activity requires a lot of expertise and makes the model less natural and expressive.

Since the focus of this paper is on expressiveness and features of modeling languages, LTL will not be discussed further. Instead, this section provides more information on Promela, especially when it departs significantly from conventional programming languages. Generally speaking, the syntax of Promela is similar to the C programming language most embedded system programmers are familiar with. The main additions to its semantics, which significantly contribute to language expressiveness, are: support for *concurrent execution*, statements *executability*, and *non-deterministic choices* in the execution flow.

For what concerns concurrent execution, C-language support is left to libraries that are not part of the language proper, like in the POSIX standard. On the contrary, processes are first-class citizens in Promela and encompass all executable statements of a model. A process `P` can be declared as follows:

```
active [n] proctype P(params) {
  body
}
```
(1)

where `params` is a list of formal parameters and `body` is a sequence of statements. Processes can be instantiated statically (by prepending **`active`** `[n]` to their declaration, where `n` is the number of instances) or dynamically (by means of the **`run`** operator). There is no built-in support for periodic processes or activation conditions. Both must be explicitly modeled if needed, taking into account that Promela by itself has no concept of time, unless it is suitably extended [7].

A significant departure from traditional programming languages is that, in Promela, statements may or may not be *executable* depending on a Boolean condition that, in turn, depends on the statement itself. For instance, an expression (classified as a kind of statement in Promela) is executable if it evaluates to **`true`** or, equivalently, to a non-zero value. Whenever a process encounters a statement that is not executable at the moment, it *blocks* until the statement becomes executable. This concept, together with the atomic execution of a sequence of statements (specified by means of the **`atomic`** keyword) provides a very concise and effective way to represent interprocess synchronization.

Regarding execution flow the most peculiar feature of Promela are non-deterministic choices, better illustrated with an example. Let us consider the following fragment of code that resembles a conventional "if–then–else if" statement:

```
if
:: i == 1 -> k=1
:: j == 1 -> k=2
fi
```
(2)

The `::` keyword introduces an execution alternative, while **`->`** separates the predicate or *guard* (on its left) from the list of statements associated with it (on its right). When the execution flow is *sequential*, the first guard that evaluates to true triggers the execution of its list of statements and execution continues after the conditional statement as a whole. Instead, when more than one guard is true in Promela, a *non-deterministic choice* exists among the corresponding lists of statements, and all possibilities are considered upon verification. In the example above, when both `i` and `j` are 1, `k` can be set to *either* 1 or 2, whereas it would invariably be set to 1 in a traditional programming language.

On the other hand, inter–process communication takes place in a more conventional way. Besides global variables, which are still popular despite being considered questionable programming practice [8], Promela supports multi-point communication channels. The following statement defines a channel `ch` that holds up to `n` messages. Individual messages consists of fields belonging to the specified list of `type`.

```
chan ch = [n] of {type, ..., type}
```
(3)

Messages can be sent to, and received from, channels by means of a rich variety of operators similar to those available in the CSP language [9]. The basic synchronous send (denoted as **`!`**) and receive (**`?`**) operations block the invoking process when the channel is full or empty, respectively, but polling variants also exist. Other flavors of the receive operation can read from the channel in a non-destructive way and/or access it in non-FIFO order.

For what concerns *model execution*, besides verification mode SPIN also implements a *simulation* mode, meant to help designers understand and debug their models interactively. However, it is unsuitable for targeting embedded real-time systems, also because Promela does not directly support I/O functions and networking, whereas both of them are crucial to interface software modules with real equipment. In other words, Promela and SPIN implement a very high-level abstraction of the computing platform, and cannot generate any executable code for an actual target, like a microcontroller. Although some attempts at translating Promela from and into the Java and C languages have been carried out in the past [10], [11], they may limit their scope to a subset of Promela and/or neglect part of its semantics.

B. The CPAL Language

Cyber-Physical Action Language (CPAL) is a multi-purpose language designed for modeling, simulating, and programming typical embedded systems in a unified way [5]. It was partly inspired by Promela in emphasizing language expressiveness and simplicity, while natively supporting concurrent, real-time programming concepts. The similarities between them become evident when we compare, for instance, the following fragment

of CPAL code with its Promela counterpart (1). Both declare and instantiate a process.

```
processdef P(params) {
  body
}
process P: inst[period,offset][cond](args);
```
(4)

The same example also shows how CPAL addresses several key areas of interest, especially when considering embedded or more in general Cyber-Physical Systems (CPS) with real-time constraints. In particular, CPAL processes can be either *periodic* or *event-triggered*. Periodic instances are scheduled for execution with a certain `period` and an optional `offset`. It is also possible to specify a Boolean activation condition `cond`. In this case, a given process instance is scheduled for execution only if its activation condition is true at release time. Processes are *event-triggered*, when they are instantiated with just the activation condition. In this case, the instance is released as soon (and as long) as the condition evaluates to true. As we can see, time is a native concept in CPAL and enjoys full language-level support.

Moreover, unlike Promela which considers every possible interleaving among processes during verification, CPAL offers several predefined scheduling models and algorithms that can be configured by means of *annotations* to the code. For the time being, CPAL considers only *non preemptive* algorithms—in which processes run to completion once scheduled—as they represent the commonest case in most time-critical systems. Annotation offers a convenient way to experiment with different scheduling strategies in simulation mode, possibly with information fed back from the real execution environment, and then enforce the same strategy when the model executes on the target embedded system. For the First–In, First–Out (FIFO) algorithm, schedulability analysis through worst-case execution time determination is supported [12]. It worth noting that other execution-related timing information can be specified through annotation as well, such as execution time and jitter, and can be performed at different level of granularity (e.g. process-level, state-level, or transition-level). Generally speaking, the annotation mechanism provides a clean separation between functional and non-functional properties of a program.

The dissimilar approaches followed by Promela and CPAL represent a trade-off between full exploration of all possible behaviors or being focused on a few specific, consistent behaviors of practical interest. On one side, Promela aims at a formal proof of code correctness across the whole state space of the system, which may come at the expense of huge computational resources that may even make the problem intractable. Alternatively, designers may be forced to artificially streamline the model to make it amenable to verification, usually achieved by abstracting away some implementation details. However, this way of doing may easily contradict the purpose of a formal proof. In other words, it may be possible to prove that a model is flawless, but without knowing exactly how close the model is to the real system it is supposed to represent. On the contrary, in order to avoid any gap between the model and the real system, CPAL supports *direct model execution* by means of an interpretation engine (available on various platforms such as Windows, Linux, Raspberry Pi and Freescale FRDM-K64F), without any intervening translation. Hence, the system model coincides with the actual executable program.

Another important aspect CPAL deviates from Promela (and other languages as well) is the *process structure*. The following fragment of code elaborates further how the process `body` shown in (4) must be organized internally.

```
processdef P(params) {
  state state_1 {
    statements ...
  }
  on (cond) { trans_code } to state_2;
  after (time) { trans_code } to state_3;
  ...
}
```
(5)

As shown above, CPAL processes are structured as Finite State Machines (FSM) and expressed in terms of *states* and *transitions*. A set of *transitions*, which govern how a process goes from one state to another, can be specified after the state code and introduced by the **on** or the **after** keyword. In particular, the latter specifies a time-triggered transition. An execution step of a process can be summarized as follows: upon activation, a process is assumed to be in a certain state. According to Mealy FSM semantics, all outgoing transitions from that state are evaluated first, according to their order of declaration and the first one evaluated to true is taken. After that, the process executes the (optional) transition code (`trans_code`), enters the target state and executes the block of code associated to it. With respect to the completely free structure of Promela processes, standardizing a well-defined and well-known implementation logic as done in CPAL improves code readability and makes it easier to understand, especially when a model is shared among different programmers.

Regarding executable statements, CPAL capabilities are remarkably close to what Promela provides. In addition, CPAL offers a richer set of looping constructs, which not only improves code readability, but also reduces the probability of programming mistakes. The only looping construct provided in Promela is **do**/**od**, with its syntax identical to the **if**/**fi** statement shown in (2), except for keywords. Semantics are similar, too, the main difference being that the evaluation of execution alternatives is repeated indefinitely, until the loop is broken by a **break** statement. By contrast, besides C-like **for** and **while** loops, CPAL also provides the ability to loop over a collection of items (e.g., an array) by means of an *iterator* (e.g. `it`) associated with operators such as `it.current` (current element) and `it.index` (index of the current element).

```
loop over array with it {
  body
}
```
(6)

Inter-process communication is another area where CPAL and Promela take similar approaches. In CPAL processes may exchange information through global variables—although their use is discouraged for the same reasons as outlined in Section II-A. CPAL natively supports the `channel` data type, with two sub-types (namely, `stack` and `queue`) inherited from it and implementing a Last-In First-Out (LIFO) and First-In

```
typedef buffer_t {
  bool agent[N]
}

typedef report_t {
  byte i;          /* Number of reports */
  bool value[N-1] /* Their value */
}

#define nil   2

typedef result_t {
  byte agent[N]
}

typedef results_t {
  result_t opinion[N]
}
```

Fig. 1. Promela Data Type Definitions.

First-Out (FIFO) insertion and extraction policy, respectively. As is common in structured type hierarchies, either a stack or a queue can be used whenever a channel is expected.

```
queue <eltype> : q[n];

processdef P(in channel <eltype> : c) {
  ...
}

process P: p[1s](q);
```
(7)

The above code declares a queue `q` of `n` elements of type `eltype`, which is then passed as argument to an instance of process `P`. Similarities with Promela (3) are evident. One difference worth noting is that in Promela the channel insertion and extraction policy depends on the *operator* being used, whereas in CPAL it is determined by the channel *data type*. For instance, in CPAL insertion and extraction always take place by means of the **push** and **pop** operators, but their exact semantics depend on the data type they work upon.

One more feature of CPAL, extremely important from the practical point of view, especially when considered in combination with model execution, is the possibility to specify *real* I/O operations. More specifically, on embedded targets, global variables can be mapped to I/O registers, for instance, General-Purpose Input-Output (GPIO) ports to read the value of, or write values to, the hardware I/Os.

III. An Interactive Consistency Protocol

A. Protocol Description

The protocol being considered in this paper has been originally proposed in [13] for the SIFT fault-tolerant aircraft control computer. It enables a group of n agents A_i to share a value V_i of their choice, by communicating through a perfect network, that is, a network that never drops, alters, or duplicates messages. Although up to m agents (with $n \geq 3m+1$) may fail and send forged messages, the remaining $n-m$ non-faulty agents still reach an agreement on the values all other agents chose. More specifically, each non-faulty agent is able to build an n-element vector. These vectors are all identical and each element corresponding to a non-faulty agent A_i is equal to V_i. In the following we focus on the case $n = 4$ and $m = 1$, in which the protocol consists of two rounds.

Informally speaking, in the first round, each A_i sends to the others its V_i. Hence, each agent gets one report about each of the other agents' value. In the second round, each A_i forwards to the others the reports it received during the first round. Eventually, each agent has 3 reports about each V_j, $i \neq j$. As proved in [13], each A_i can build its interactive consistency vector R_i according to the following procedure: $R_{i,i} = V_i$, and $R_{i,j}$, with $i \neq j$, is obtained by majority voting among the 3 reports about V_j that A_i received. If there is no majority, $R_{i,j}$ is set to the reserved unique value *nil*.

```
inline second_round() {
    d_step {
        i=0;
        do
        :: (i<N) -> {
                buffer.agent[i] =
                  ((i==id) -> false : report[i].value[0]);
                i++
            }
        :: else break
        od
    }

    dest=0; skip_id(id, dest);
    do
    :: (dest<N) -> {
           c2[dest] ! id, buffer;
           dest++; skip_id(id, dest)
       }
    :: else break
    od;

    atomic {
        i=0;
        do
        :: (i<NB) -> {
                c2[id] ? source, buffer;

                j=0;
                do
                :: (j<N) -> {
                        if
                        :: (j==source || j==id) -> skip
                        :: else add_report(j, buffer.agent[j])
                        fi;

                        j++
                    }
                :: else break
                od;

                i++
            }
        :: else break
        od
    }
}
```

Fig. 2. Second Protocol Round in Promela.

B. Promela Model

The Promela model of the protocol is based on previous work [14]. The main data types relevant to the protocol are defined are shown in Fig. 1. Namely:

- `buffer_t` represents a buffer of n values; it is used, for instance, during the second round of message exchanges.
- `report_t` is a structure that collects reports about the value held by a certain agent.
- `result_t` and `results_t` represent results obtained by *one* agent, and *all* agents, respectively.

This excerpt also provides the opportunity to exemplify the modeling trade-off discussed in Section II-A. Although the protocol itself supports any kind of V_i (as long as they can be compared for equality), to reduce verification times only the simplest Promela data type has been considered in the model, that is, the Boolean (**bool**) data type. This approach has been used when defining `buffer_t` and `report_t`. However, since

```
struct Value {
  uint8: value;
};

const Value: NIL = { uint8.LAST };

struct Vector {
  Value: value[N];
};

struct Reports {
  uint8: n_reports;
  Value: report[N-1];
};

struct All_Reports {
  Reports: agent[N];
};
```

Fig. 3. CPAL Data Type Definitions.

nil is also a possible outcome of the protocol, the **byte** data type has been used for results.

The second round of message exchanges has been modeled as shown in Fig. 2. Each agent first prepares the message to be sent (lines 3–11). These steps are enclosed within a *deterministic step* block (**d_step**) in an effort to reduce verification time, again at the expense of model accuracy. In fact, this approach forces Promela to consider all statements enclosed in the block as non-blocking and indivisible, thus neglecting their interleaving with other parts of the model. Messages are then sent through the appropriate channels (lines 14–21) held in the array `c2`. It is worth noting that the processing of incoming messages (lines 24–45) requires two nested **do/od** loops, which hinders readability, and is enclosed within an **atomic** block for the same reasons described previously—although **atomic** implies a weaker assumption with respect to **d_step** and lets Promela consider interleaving upon blocking. As a last remark, the code relies on two functions not shown in the figure: `skip_id` is used by agents to skip their own identifier and avoid sending messages to themselves, and `add_report` adds a report to a `report_t` data structure.

C. CPAL Model

As done in Section III-B, the description of the CPAL model starts from data type definitions shown in Fig. 3, which are the counterpart of Fig. 1. Although a **uint8** (8-bit unsigned integer) was used as `Value` (the data type of V_i,) in the figure, any other data type can be used without impacting simulation performance, unlike in Promela. In this way, it was also possible to remove the artificial distinction between Promela's `buffer_t` and `report_t` data types, by reserving the highest value of a **uint8**, denoted as **uint8**.`LAST`, as *nil*.

Coming down to the CPAL model of the second round of message exchanges, shown in Fig. 4, besides noting the similarity with Fig. 2, it is also important to underline the better readability of the **while** and **loop over** constructs with respect to their **do/od** counterpart, even when they are nested to handle incoming messages (lines 29–40). It also worth remarking how the FSM organization enforced by CPAL (the `Round_2_Tx` state handles transmission, while `Round_2_Rx` processes incoming messages) allows designers to model the protocol concisely without impairing readability in any way.

```
state Round_2_Tx {
  var Round_2_Msg: msg;

  msg.sender_id = id;
  loop over msg.other_values.value with it {
    if(it.index != id) {
      msg.other_values.value[it.index] =
        st.all_reports.agent[it.index].report[0];
    }

    else {
        msg.other_values.value[it.index] = NIL;
    }
  }

  loop over round_2_chan with it {
    if(it.index != id) {
      it.current.chan.push(msg);
    }
  }

  st.received = 0;
}
on (true) to Round_2_Rx;

state Round_2_Rx {
  var Round_2_Msg: buf;

  while(round_2_chan[id].chan.not_empty()) {
    buf = round_2_chan[id].chan.pop();

    loop over buf.other_values.value with it {
      if(it.index != buf.sender_id) {
              add_report(it.current,
                      st.all_reports.agent[it.index]);
      }
    }

    st.received = st.received + 1;
  }
}
on (st.received == N-1) to Result;
```

Fig. 4. Second Protocol Round in CPAL.

IV. Comparison and Discussion

A. Model-based verification versus model-based development language

Promela is a language meant for model-based verification, that is the verification on models of correctness properties. In that regard it can perfectly be used within a model-based design flow typically involving specifications in UML as in [15]. But Promela is not an implementation language, i.e. a language to develop software or systems. In particular, it lacks important features such as input/output capabilities or floating point support. More essentially, non-determinism is a behavior at the core of the language, while this is most often something to be avoided in real systems, especially in critical systems. In addition, Promela is, in certain aspects, a low-level programming language, that comes without libraries or domain-specific frameworks, and thus would not be a productive development environment for real applications.

By contrast, CPAL, as a domain-specific language for CPS, has been conceived with productivity as a design objective and, to that aim, provides high-level abstractions suited to express domain-specific properties or patterns of behaviors. Although CPAL allows some forms of verification by timing-accurate simulation, monitoring and schedulability analysis, it is also an implementation language, and specifically a language to implement systems for which the timing behaviors of their components matter. Time is indeed a central concept of the language, with time units parts of the language, the **after** transition in FSMs, process activation and scheduling features.

B. Logical versus temporal correctness verification

On the contrary to CPAL, quantified time is something which is absent in Promela. A Promela model is for the formal study of all possible evolutions/trajectories of the modeled system, and perform value domain and logical-order verification across the entire space of possible evolutions. This is made possible by model-checking with SPIN. Although, in practice, model-checking comes often at the price of using simplified models because of the search space explosion problem. CPAL, on the other hand, does not support model checking. A CPAL program is usually about studying and enforcing specific real-time behaviors. Indeed, in real-time systems, the goal of the designer is in most cases not to identify all possible trajectories of a system but only the ones possibly leading to "feasible" systems.

Underlying CPAL, there is the idea, inspired from the design of interlocking systems [16] in use at the French national railway company, to decouple the execution platform (i.e., the interpreter) from the application and verify the correctness of both independently. Promela also abstracts away the execution platform, it is simply assumed that it will perform correctly (e.g., delivering messages in the right order, etc.). But it is not possible to configure the platform, typically the real-time QoS we expect from it through the use of scheduling policies, while this is possible in CPAL. Although it may be possible in Promela to, for instance, implement an EDF scheduler for the processes, it would be cumbersome and not in the spirit of the language which is based on non-determinism.

C. A continuum of verification from behavioral to temporal properties

Considering the development cycle, Promela, in our opinion, is targeted at the design stage, to build coarse-grained models abstracting the implementation issues. CPAL, on the other hand, covers the entire development cycle until deployment with a more accurate modeling of the execution platform. In terms of verification, what is the true need of the designer at the design stage? Is it formal verification on a simplified model (with respect to the implementation) or simulation of the actual implementation, or, at least, of a model close to it? We believe the two possibilities should not be opposed. Promela appears to us best suited for the early design phase, to decide the architecture of a system, identify and prove its main behavioral properties. Then, the next stages can be performed with CPAL, resulting in a system proven correct in its timing dimension too and providing its implementation. This continuum of verification, based on increasingly refined and platform-accurate models, obeys the principle of specifying and verifying one concern at a time, from logical and value-domain verification to timing verification.

V. Conclusion

In this paper we compared two MBD languages—Promela and CPAL—in terms of language constructs, verification capabilities, and role in the software life cycle. Both languages are straightforward to learn, unlike many other formal languages, whose steep learning curve and complex formalism can discourage practitioners. However, Promela aims at the verification of behavioral properties and is not meant to provide or directly support an implementation, leading to a discontinuity in the design flow. On the other hand, CPAL enables designers to write a model that can be directly used for the implementation, thus alleviating this discontinuity while still supporting some properties verification.

Together, these languages target complementary correctness properties and may offer a continuum of verification from behavioral to temporal properties, provided proper model refinement techniques are developed. Ideally, those refinements could be automated and span from the Promela model down to CPAL and C code generation, and then RTOS integration. Even on systems for which model interpretation is not efficient enough, this would still provide a development flow where the properties checked at higher levels of abstraction are guaranteed to hold at the lower levels.

References

[1] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems," in *2006 IEEE Conference on Computer Aided Control System Design*, Oct 2006, pp. 1206–1211.

[2] S. Pradhan, A. Dubey, A. Gokhale, and M. Lehofer, "CHARIOT: A domain specific language for extensible cyber-physical systems," in *Proc. Workshop on Domain-Specific Modeling (DSM)*, 2015, pp. 9–16.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[4] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[5] N. Navet and L. Fejoz, "CPAL: High-level abstractions for safe embedded systems," in *Proc. of the ACM International Workshop on Domain-Specific Modeling (DSM)*, 2016, pp. 35–41.

[6] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annual Symposium on Foundations of Computer Science*, Nov. 1977, pp. 46–57.

[7] D. Bošnački and D. Dams, "Discrete-time Promela and Spin," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, vol. 1486. Springer Berlin Heidelberg, 1998, pp. 307–310.

[8] W. Wulf and M. Shaw, "Global variable considered harmful," *ACM SIGPLAN Notices*, vol. 8, no. 2, pp. 28–34, Feb. 1973.

[9] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[10] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[11] A. Sharma, "A refinement calculus for Promela," *Proc. 18th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 75–84, 2013.

[12] S. Altmeyer, S. Manikandan Sundharam, and N. Navet, "The case for FIFO real-time scheduling," University of Luxembourg, Tech. Rep., 2016. [Online]. Available: http://orbilu.uni.lu/bitstream/10993/24935/1/FIFO_scheduling_TR.pdf

[13] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[14] T. Hu and I. Cibrario Bertolotti, "Model checking," in *Digital Avionics Handbook*, 3rd ed., C. R. Spitzer, U. Ferrell, and T. Ferrell, Eds. CRC Press, Taylor & Francis Group, Sep. 2014, ch. 42.

[15] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations," *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 3, pp. 357–369, 2001.

[16] M. Antoni, "Formal validation method and tools for computerized interlocking system," Presentation at the 18th International Symposium on Formal Methods (FM 2012), Industry Day, August 2012, available at http://fm2012.cnam.fr/fm2012/ID2012-Marc-Antoni.pdf.