Model-based Design of Cyber-Physical Systems

Christian Rinderknecht

28th November 2014

Abstract

We survey existing tools and methods for the simulation, testing and verification of concurrent communicating systems, and their pertinence for model-driven design and programming of cyber-physical systems. We also cover workflows and tests and conclude with two proposals.

1 Formal methods

In the context of our project, we define a *cyber-physical system* (CPS) as a wireless mobile network of embedded systems featuring environmental sensors and computational capability. Usually, an embedded system carries on specific control functions with real-time constraints, e.g., related to safety, on top of a real-time operating system (RTOS), but we will extend the notion to encompass general-purpose computers, for example, smartphones and Raspberry Pis running GNU/Linux.

Depending on the scale and the nature of the software applications, different network topologies, routing and synchronisation protocols may be used, for example centralised architectures (client-server) or decentralised ones, like peer-to-peer (P2P) or mobile ad hoc networks (MANET), where nodes are also routers. At the application layer, real-time constraints might be expected, as well as code (a.k.a. location) migration and security guarantees.

The *model-driven design* of CPS uses formal models, possibly including real-time features, which are checked with regards to some desired properties and tested through simulation, before implementations are derived.

Model checking is a formal method whereby the truth of a logical formula (e.g., of a temporal nature) is checked on a finite model of a system. It is different from *testing*, which consists in interacting with an implementation, whose state space may be infinite, and observing whether its responses conform to its model (if no stimulation of the system is performed, we have an instance of *passive testing*). Testing hence applies to implementations and can, at best, only prove the presence of errors, whilst model checking applies to models and conclusively proves the absence of design errors or finds a counterexample (with an execution trace, called *scenario* in telecommunications), thanks to

the finiteness imposed by construction. Clearly, these two approaches are complementary, and it even makes sense to "test" a model by *simulation*, that is, by animating runs of the model itself and thusly derive *test sequences* (of inputs and outputs), ranging from purely random walks to guided explorations, for example to cover all the transitions of a component. Model-based test generation is usually a heuristics which avoids the potential combinatorial explosion of the state space at the cost of incompleteness.

Another approach is *theorem proving*, which delivers the same guarantees as model checking (except in presence of a counterexample) but does not require the model to be finite, because it relies on induction. Some proof assistants, like Coq, can even generate implementations from proofs, thanks to the use of a constructive logic. Theorem proving can fruitfully be used in conjunction with model checking (Gastel et al., 2011).

Finally, *static analysis* consists in approximating sets of values and behaviours of implementations as they would occur at run-time, without instrumenting the source code and actually running it. A very general theoretical framework for static analysis is *abstract interpretation*, which is based on a sound abstraction of the semantics of a programming language and a symbolic execution of the program. More syntactical algorithms, like static type checking or inference, also work well in practice and both approaches are useful. Static analysis can be thought of as abstracting a model from a program, and is thus complementary to the techniques we have mentioned earlier.

In the following sections, we examine the models, logics and tools most widely used for model checking, and we assess whether they are suitable for designing and programming CPS and, more precisely, which features of CPS they support best. We finally outline different approaches to harness them to our project.

2 Model checking

Kaliappan (2008) offers a short survey on verification techniques from the software engineering perspective, with an emphasis on systems specified with UML. Zheng ('Introduction to Computer Aided Verification') provides a good tutorial.

In this section, we review and compare two verification systems, namely SPIN and UPPAAL, which are mature, maintained and used both in academy and industry. We proceed by considering translations between models, from model to implementation ("top-down") and from implementation to model ("bottom-up", or *model extraction*), in particular McErlang, a model checker for Erlang written in Erlang. But first, we need some notion about *temporal logics*.

2.1 Temporal logics

The properties or, put more properly, the claims to be checked on the models are expressed in some temporal logic, that is, a modal logic interpreted over time ('Temporal Logics'). The most common properties expected from a system are

- reachability, i.e., "Something good may happen;"
- liveness, i.e., "Something good eventually happens;"
- safety, i.e., "Something good always happens."

Reachability is a weak property, which is nonetheless useful to debug a model under construction, for example, by asserting that a process may access a given resource (a scenario can usually be automatically constructed). Safety is a strong property reminiscent of a global invariant in non-temporal logics, and it can be utilised for example to state mutual exclusion, that is, that, at any time, there will be no more than one process accessing a common resource, or the absence of deadlocks, that is, that, at any time, there will be no processes waiting transitively on each other for the release of a lock on a shared resource. Other interesting safety properties are the non-reception of unspecified messages and the absence of invalid end states.

In a temporal logic, doing nothing obviously realises safety, and this is where liveness comes into play, to require some progress. For example, liveness is useful for expressing the absence of starvation, in other words, the fact that a process will gain access to a resource. In the theory of programming languages, partial correctness, by which, if a program terminates, then its output satisfies some condition, would be considered a safety property in temporal logics, whilst termination would be considered a liveness property. A criterion to distinguish between safety and liveness is the following: safety can, in theory, be *disproved* in finite time, by simple observation of the states of the system execution, whereas liveness can be *proved* in finite time.

Let us note that temporal logics assume infinite futures, and since the system is finite, its behaviour must then be cyclic, as this happens to be the case of many communication systems implementing sessions or interactive loops, like servers and command shells. This aspect is perhaps addressed more obviously by properties like *fairness*, which states that "something good will happen an arbitrary number of times," and can be conceived as a repeated liveness property. For example: "If access to the critical section is infinitely often requested, then access will be granted infinitely often." Fairness is most useful when dealing with the scheduling of processes.

2.2 SPIN

SPIN (Holzmann, 1997; Holzmann, 2003; Ben-Ari, 2008) is a system supporting the design, simulation and model checking of asynchronous systems.

Accordingly, sequential computations tend to be abstracted so the state of the system reflects mainly process interactions. These can be classified into several kinds: rendezvous primitives, asynchronous message passing through buffered channels, shared variables, and any combination of these.

The language for defining the models is Promela (process meta-language), from which the name SPIN is actually derived: "Simple Promela Interpreter." It is an imperative specification language with a C look and feel, except for the lack of pointers, dynamic memory allocation and functions returning values. The purpose of those restrictions, and others, is mainly to constrain the number of states in the model to be finite. In theory, this enables all correctness properties to become decidable, although, in practice, some engineering methods and techniques are needed to overcome resource limitations in the presence of very large state spaces.

The language used in SPIN to express temporal formulas is based upon the standard Linear Temporal Logic (LTL), where the future is denoted by the set of all possible paths that the system can take, and quantification in formulas is always universal, that is, it is not possible to isolate a strict subset of paths with respect to a property. The typical temporal operators used are

- $\bigcirc\varphi$, meaning that φ is true in the *next* moment in time;
- $\Box\varphi$, meaning that φ is true in *all* future moments;
- $\Diamond\varphi$, meaning that φ is true in *some* future moment;
- $\varphi \mathsf{U} \psi$, meaning that φ is true *until* ψ is true.

The usual connectors of classical logics are present as well in the LTL, like conjunction, negation etc. but time is *not* explicitly present, for example in the guise of clocks. As a consequence, time constraints, like timeouts, can not be modelled, but partial orderings of process interactions can.

Once a Promela model and temporal formulas in LTL have been written, SPIN generates a dedicated C program, whose run checks the claims against the specification. In case the formulas are found to be satisfied, there is no automatic generation of a skeleton of the implementation, which leaves a gap where errors can be introduced. In case the state space is too large, Promela allows for the embedding of C code as atomic sections whose scope is local and invisible to other processes (note: there is no hierarchy of processes). Of course, this is only feasible if the code in question does not deal directly with the control aspect of the protocol, otherwise its effects have to be observable in the global state space. Purely computational tasks can be abstracted this way, though. Pajic et al. (2012) remark:

> Due to the limitations of the verification process (e.g., restricted model size), some parts of the models used for verification represent over-approximations of the more "realistic" models. For example, for verification of Cyber-Physical Systems (CPS) that

feature a tight coupling between the real-time controller and (usually) continuous physical environment, it is necessary to model the closed-loop system as a whole.

By contrast, note that in the control system industry (e.g., automotive industry, aerospace, robotics, logic circuit design etc.) *synchronous languages* like Esterel, Lustre, Simulink etc. are used to express the models, and these allow automatic generation of the implementation. By contrast, the application domain of SPIN is that of dynamic asynchronous software systems, which allow the instanciation and scheduling of new processes at run-time, and where processes may perform complex computational tasks in addition to protocol related operations.

2.3 UPPAAL

Timed automata are a standard modelling formalism for real-time systems because they enable efficient system verification by model checking and testing (Bouyer, 2003; Bengtsson and Yi, 2004). Basically, a timed automaton is a finite-state machine extended with clock variables. The time denoted by the clocks is dense or continuous, in other words, it is a mathematical real number. Moreover, edges of the automata are threefold, in all generality: not only do they carry *send or receive* actions for synchronisation on unbuffered channels, but also *guards*, which are constraints on the clock values that must hold for the edges to be executable, and *clock resets* to zero. The semantics of a network of timed automata is their synchronous product. This kind of model is usually extended further with bounded discrete variables which are part of the system state and are used as if in an imperative language. In sum, the state of the system consists in the locations (vertices) of all the automata, the clock values and the discrete variable values.

The model checker UPPAAL (Amnell et al., 2000; Behrmann, David and Larsen, 2004) uses an extension of timed automata, called *timed safety automata* because the theoretical model is further extended with features which help in proving safety and liveness properties, like urgent and committed locations, urgent synchronisations and broadcast channels. User functions are allowed for computational tasks and have a C look and feel, except for the absence of pointers. Synchronisations are realised only with rendezvous on one action name: the sender is blocked until the receiver is in a state with an out-going executable edge receiving the same action; the sender is released after the receiver has accessed and perhaps modified the global state and reached the next local state (vertex). In other words, there is no parameter passing with actions: instead, shared variables and side-effects are used to exchange information (bidirectionally) during the rendezvous.

The query language is based on *Temporal Computational Temporal Logic* (TCTL), a modal extension of *Computational Temporal Logic* (CTL) with

time. The concept of future in the TCTL is that of a tree of all possible traces from the present moment. The objects of the logical formulas are paths (in the tree) and individual states. Quantifications are universal or existential: $\mathsf{A}\,\varphi$, which means that "for all paths, the formula φ holds," and $\mathsf{E}\,\varphi$, which means that "there exists a path such that φ holds." Furthermore, we find the following modal operators:

- $\mathsf{X}\,\varphi$, meaning: "In the next state, φ is true;"
- $\mathsf{F}\,\varphi$, meaning: "In a future state, φ is true;"
- $\mathsf{G}\,\varphi$, meaning: "Globally in the future, φ is true;"
- $\varphi\ \mathsf{U}\ \psi$, meaning: "$\varphi$ is true *until* ψ is true."

For example (Raimondi, 2008): $\mathsf{AG}(\varphi \rightarrow (\mathsf{EF}\,\psi))$ means: "It is globally the case that, if φ holds, then there exists a path such that, at some point in the future, ψ holds as well."

UPPAAL is actually a toolkit made of a graphical user interface and a built-in model checker. The graphical interface is convenient and helps understanding the model through simulation. There is no automatic generation of a skeleton of the implementation, which presents a risk of introducing errors.

There is an extension of model checking which attracts a lot of attention nowadays: *statistical model checking* (David, 2012; Legay and David, 2012). Simply put, the timed automata are extended with probabilities to become *Priced Timed Automata*, and a stochastic semantics is used. The temporal logic TCTL is also extended conservatively to support probabilistic queries. The main interest of these extension is to have model checking go beyond worst case analyses, like the worst case response time of a recurrent task following a given schedule, and assess the average case behaviour of real-time systems. In other words, statistical model checking decides whether a system statisfies a property *with some degree of confidence*. There are two extensions to UPPAAL supporting this new paradigm: UPPAAL-TRON (Larsen, Mikačionis and Nielsen, 2009) and UPPAAL-SMC (Bulychev et al., 2012).

2.4 Comparison

Perhaps the most obvious differences between the modelling languages of SPIN and UPPAAL are the absence of real-time (i.e., clocks) in the former and the absence of asynchronicity (i.e., buffered channels) in the latter.

Promela allows several kinds of synchronisations, whilst UPPAAL only offers binary rendezvous (except for broadcast) and the sharing of variables. In other words, Promela features synchronisation by message passing, whilst UPPAAL uses rendezvous and shared variables. In particular, this means that the exchange of information in UPPAAL can be bidirectional. There has been a recent attempt at a workaround in UPPAAL by Boudjadar et al. (2013), who define a new class of automata called *communicating timed automata*,

which are timed automata extended with the concept of "call (of another automaton) with parameters." These automata are defined by translation into standard UPPAAL automata. See also Chandrasekaran and Mukund (2006) and their *timed message-passing automata*.

As far as data types are concerned, both Promela and UPPAAL have primitive and structured data types, including arrays, although Promela does not allow arrays to be passed as a parameter during synchronisation.

Buffered channels can be modelled explicitly in UPPAAL as timed automata with an array of messages as a local variable (Chandrasekaran and Mukund, 2006), but this may quickly overload the state space, as all permutations of the messages are observable. In Promela, channels are first-class objects, that is, they are values and they can be sent from one process to another in order to create a private communication channel. By contrast, in UPPAAL, channels are not a concept and they have to be inferred on the basis of the synchronisations, i.e., if two automata can synchronise, an implicit channel exists between them.

There have been some proposals (Tripakis and Courcoubetis, 1996; Bošnački and Dams, 2002; Nabiałek, Janowska and Janowski, 2008) to extend Promela with real-time features, but they are not widely used.

It is possible in Promela to create at most 255 processes at run-time, whilst UPPAAL requires a statically fixed number of automata (so-called template instanciations). The idea of Boudjadar et al. (2013) to extend UPPAAL with dynamic automaton creation is not entirely convincing here because it relies implicitly on some unstated static analysis to determine an upper bound on the maximum number of automata active simultaneously.

Some industrial applications have been carried out with the side-effect or main intent to compare SPIN and UPPAAL (Jensen, Larsen and Skou, 1996; Lingegowda, 2006; Nabiałek, Janowska and Janowski, 2008; Günther, Milius and Möller, 2012; Alzahrani and Georgieva, 2013). Also worth of notice is the extensive dual tutorial by Klüppelholz (2012). Therefore, to a certain extent, Promela and UPPAAL can simulate each other (it is an open question as to what are the largest subsets of these languages that are equivalent by bisimulation), but, in practice, the models of the former often use asynchronous channels and the models of the latter often rely prominently on clocks, which makes the two languages incommensurable in general.

As for the temporal logics, LTL versus TCTL, the timed nature of TCTL makes then trivially incomparable. The untimed fragment of TCTL, which is CTL, has *not* the same expressive power as LTL. A CTL formula and a LTL formula are said equivalent if, for all models expressed as Labelled Transition Systems, they have the same truth value. It can be shown that there are formulas in CTL which have no equivalent in LTL, and vice versa, although there exists equivalent formulas. For a discussion of their respective merits, on practical and theoretical grounds, see Vardi (2001) and the tutorials by Katoen (2007) and Krug (2010).

2.5 Translations

We briefly consider here three sorts of translations involving the models of SPIN and UPPAAL, either as source or target: from one model to another, from a model to an implementation and from an implementation to a model.

An example of translation from UPPAAL to another modelling language is given by Abel (2009), who does not build the product automaton before translation but uses a stepwise algorithm based on the semantics of timed automata. Another example of intermodel translation is that of Nabiałek, Janowska and Janowski (2008), whom we mentioned earlier because their source is a discrete-time extension of Promela. Their target is a general language of timed automata, compatible with that of UPPAAL.

There have been a few studies on the automatic generation of implementations from models, for example from Promela and UPPAAL to C or C-like languages ('Automatic Translation from UPPAAL to C'; Löffler, 1996; Hendriks, 2011).

As far as model extraction is concerned, we would mention a recent translation from C to Promela by Jiang (2009). Another way to proceed is that of McErlang (Earle and Fredlund, 2010; Earle and Fredlund, 2012) with a timed semantics of Erlang programs. With this system, Erlang programs need to use a library providing time-related functions, for example, to obtain or record the current time, and a dedicated run-time must be used instead of the standard virtual machine. More precisely, the run-time is an interpreter for Erlang in Erlang. Time is discrete and, by default, there is no global clock ticks: the difference between two successive timeouts defines a time slice. The vertices of the state graph consist of the virtual time elapsed since the start, and the edges are annotated by send, receive or timeout statements. Some restrictions ensure that the graph is always finite, so model checking is possible. McErlang can also be integrated with QuickCheck, a popular tool for unit testing ('QuickCheck and McErlang integration'; 'QuickCheck-McErlang Integration Tutorial').

3 Testing

Statistical model checking is closely related to the use of model checkers for testing (Larsen, Mikučionis and Nielsen, 2004), because randomness is often needed by the heuristics deriving test sequences according to a given objective, like the coverage of an embedded component (that is, a part of a system under test which has no direct connection with the tester).

The specificity of testing (an implementation derived from) an UPPAAL model is that real-time constraints must be taken into account when deriving the test suite; in fact, the test system becomes itself a real-time system. One interesting technique proposed by Hessel et al. (2008) consists in adding a timed automata (an *observer automaton*) to a given UPPAAL model, as

well as annotations to assign test objectives, coverage and verdicts. These annotations are then translated into pure UPPAAL in such a way that model checking itself will generate the test sequences. More precisely, a formula ψ is devised so that it specifies the desired structural criterion, like coverage of some part of the control flow graph, and we check for "always not ψ": the counterexample given by the model checker is the test case. The caveat, as usual with model checking, is that the state space must be finite.

Another closely related issue is the matching of scenarios with time constraints against a given model. These scenarios can be conceived as test sequences, with a positive or a negative expected verdict. Chandrasekaran and Mukund (2006) propose an extension of the classic UPPAAL timed automata, called *timed message-passing automata*, and Timed Message Sequence Charts, a timed extension of MSC. They show how to translate these into UPPAAL, at the cost of some restrictions on expressiveness and efficiency.

4 Proposals

This section outlines two research proposals for the programming of CPS.

4.1 Model-based design for CPS

We assume that we have a library of Erlang combinators for tasks and workflows, inspired by the iTasks framework in Clean. Its usage is eased by an extension of Erlang with infix operators. Moreover, we suppose that we have at our disposal some static analyses applicable to Erlang programs, either dedicated by means of RefactorErl, or generic by means of McErlang.

Different model-driven designs should be supported by our framework.

1. *Design with combinators.* If the workflow of the CPS application fits the scope of the combinators' library, we may simply want to directly and only use combinators.

2. *Model checking with SPIN.* If model checking is required or deemed valuable, we may want to model the CPS in Promela. The limitation is that the model must be finite, in particular, a maximum of 255 processes can be active simultaneously, and no real-time constraints are possible.

   (a) *Generation of Erlang.* If this is not an issue, model checking can be done and a mapping from processes to network (Erlang) nodes shoud be provided. Some research would be then necessary to translate Promela to Erlang, perhaps targeting the combinators' library or the OTP behaviour `gen_fsm`.

   (b) *Model refinement.* If Promela proves to be too restrictive, or if we had to hide some behaviours from the state space due to memory

limitations, we may consider a model refinement whereby the system is divided into one global model and several local (sub)models, each being the refinement of a Promela process considered as a stub or abstract, and being mapped to one network node. When creating submodels which, collectively, define the behaviour of one Promela process (one network node), we must make sure that we indeed make a refinement, so we do not invalidate any assumption at the global level (the observable timed traces of the submodels must be included in the timed traces of the corresponding abstract Promela process). From there, different routes open.

i. *Generation of Erlang.* As in step 2a, Erlang could be produced from the local Promela models.

ii. *Translation to UPPAAL.* If real-time constraints are important, we could opt for another way and translate our model such that the global model is asynchronous and written in Promela, and the local models are synchronous and written in UPPAAL. (In control systems, this kind of design is called "Globally Asynchronous, Locally Synchronous" or GALS.) The simpler way to achieve that is to rely on Promela declarations of unbuffered channels and translate the process involved into UPPAAL. Each remaining local asynchronous communication, for example with physical sensors, would be simulated by one timed automata (one per Promela channel) holding a bounded array of messages as a local variable.
The translation from Promela to UPPAAL should introduce real-time constraints. One solution would perhaps define a Domain Specific Language (DSL) as a superset of Promela, which includes real-time annotations. These annotations would be stripped every time standard Promela is required. Another way is to revive some of the real-time extensions to Promela proposed by some researchers. It would be interesting also to consider whether UPPAAL textual models could be directly embedded in (deterministic) atomic sections in an extension of Promela, differing to UPPAAL all analyses about these. Yet another angle would be to consider an existing DSL for specifying program transformations, like in the Spoofax workbench, and both inject and remove real-time constraints, akin to parsing and pretty-printing, the latter being useful when the untimed design has to be modified. After the model checking and simulation of the UPPAAL submodels, some research is needed to automatically translate these into Erlang, as we envisaged it in step 2a for Promela models.

4.2 Model-based testing of CPS

4.2.1 TTCN-3

TTCN-3 (*Testing and Test Control Notation version 3*) is a programming language strongly and statically typed, specialised for the conformance testing of telecommunication systems, but it is also a language used to specify interfaces for the test infrastructures. It is used for large projects in telecommunication, for instance mobile telephony, rather than software engineering in general. (By the way, it would be interesting to determine if this difference is cultural or technical in nature.)

From the point of view of programming theory, this language presents many interesting features which definitely merit attention. For instance, TTCN-3 features

- a standardised operational semantic,
- an expressive type system,
- pattern matching on expressions,
- dynamic creation of processes,
- multiple paradigms of communication, as asynchronous exchange of messages, RPC (synchronous) etc.

Some of these characteristics are also found in the functional programming language Erlang, used for implementing distributed, fault tolerant systems. Others aspects, not mentioned here (like *snapshot semantic*), are very specific to the domain of application of TTCN-3. Nevertheless, the resemblance suggests that a semantic of TTCN-3 could be defined by translation to Erlang. Moreover, since TTCN-3 can be used to test distributed systems and embedded systems, it makes sense to use it on CPS.

4.2.2 Testable workflows

A different approach would be to generate a testable workflow instead of a workflow for operational deployment. This would entail the creation of additional tasks whose purpose would be to *trace*, via pipes and taps, the communication of the other tasks. Beyond pure tracing, we could envisage also *passive testing*, whereby the testers would carry test objectives and stop when a verdict is reached, without disturbing the whole system while it is running. Some test tasks would also define *Message Sequence Charts* (MSC), or some equivalent formalism, to be matched against the exchanges of a set of communicating tasks, distributed in general on different network nodes.

In other words, we propose to create tasks, using the same framework of combinators, whose sole purpose would be to support tracing, passive testing

and conformance testing (with respect to MSC). The specification of a testing workflow would be provided at the same time as the operational workflow, hence meeting the requirement of "Task and Test Driven Development".

4.3 Workflows for business processes

Many papers have been written about the design of business processes by means of patterns and workflows and recent years have seen the emergence of standard catalogs which are used as benchmarks by researchers and software companies alike. The combinator approach of the CPS project seems to share some common tasks with the business workflows, although the languages (BPEL, YAWL) appear very specific to the business framework. The checking of these models has attracted an increasing attention, in particular with the model checker SPIN. It would be interesting to assess how far these business workflows overlap with our task-oriented worflows for CPS and how the published patterns of translation to Promela can be reused for model checking.

4.4 Functional Reactive Programming

Reactive Programming is a paradigm for programming communicating processes in the manner of synchonous data-flow languages. In particular, this type of programming, or programming languages, is seen as an attractive alternative to the pervasive use of callback functions, which tend to obscure the control flow. When reactive programming is supported by means of a DSL embedded in a functional language, it is said *Functional Reactive Programming* (FRP). The other approach consists in designing a full-fledged FRP, but it requires more work, although it offers potentially more options for static analysis of the reactive behaviour. Moreover, with the DSL, the features of the embedding language can be used, allowing more expressivity for the logic part.

Roughly speaking, in FRP, certain variables, called *signals* or *behaviours*, have time-varying values. The difference with side-effects is that the mutation is implicit and all expressions depending on behaviours are automatically updated. (In case of a FRP standalone language, the run-time must analyse the data flow to minimally propagate the changes.) The other distinctive feature of FRP is *events*, which are timed values denoting physical events, like a mouse button being pressed, or predicates. Depending on the semantics, signals and events can be first-class values or not. Many behaviours can be naturally expressed in terms of reactions to events, giving rise to reactive behaviours. Since time is not present in the semantic of functional languages, signals and events are defined as abstract data types which are created and destructured by functions, e.g., events may be specified in terms of boolean functions of continuous, implicit time or by, in non-strict functional languages,

signals can be modelled by infinite (lazy) streams of values. A library is provided to *lift* (convert) static values (constants) into behaviours and it is convenient if the language offers syntactical to implicit lift static values.

FRP is a good theoretical framework for the CPS project, because it can account for concurrency, migration (that is, code mobility) and dynamic creation of processes. We would have to add features like fault tolerance, either by extending a FRP library or by implementing some abstract types with Erlang during code generation. Recently, several backends have been developed for different functional languages, which produce JavaScript in order to implement web services. There have been some work to translate JavaScript to Scheme, hopefully it would help to translated it to Erlang as well.

Agda is a system providing a programming language with dependent types which are leveraged by a theorem prover. Recent works have shown the equivalence of Linear-time Temporal Logic (LTL), used in the model checker SPIN, and FRP. A reactive library has been written in Agda, using the type checker to verify LTL properties on the reactive (client) programs. The possibility to have an infinite space state, thanks to the availability of induction, makes Agda an appealing option, although the learning curve is steep. There is also a backend to the Agda compiler, which produces JavaScript and therefore would allow us to translate automatically the Agda programs (not the theorems) to Erlang.

References

[1] Andreas Abel. *From UPPAAL to Slab*. Bachelor thesis. Faculty of Natural Sciences and Technology I, Saarland University, Germany, 2009.

[2] Mohammed Alzahrani and Lilia Georgieva. 'A comparative analysis of time-sensitive web applications using SPIN and UPPAAL'. In: *Proceedings of the 20th Automated Reasoning Workshop (ARW)*. School of Computing, University of Dundee, United Kingdom, Apr. 2013.

[3] Tobias Amnell et al. 'UPPAAL – Now, Next, and Future'. In: *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP)*. Vol. 2067. Lecture Notes in Computer Science. Nantes, France, June 2000, pp. 99–124.

[4] Anonymous. 'QuickCheck and McErlang integration'. 3 pages.

[5] Anonymous. 'UPPAAL 4.0: Small Tutorial'. 11 pages. Nov. 2009.

[6] Anonymous. 'Verifying Real-Time Systems – The UPPAAL Model Checker'. 28 pages.

[7] Michael Balser et al. 'Interactive Verification of UML State Machines'. In: *Proceedings of the conference on Formal Methods and Software Engineering*. Vol. 3308. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 434–448.

[8] Gerd Behrmann, Alexandre David and Kim G. Larsen. 'A Tutorial on UPPAAL 4.0'. In: *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time (SFM-RT)*. Vol. 3185. Lecture Notes in Computer Science. Bertinoro University, Bertinoro, Italy: Springer-Verlag, Sept. 2004, pp. 200–236.

[9] Gerd Behrmann and Kim Larsen. 'Intro to UPPAAL'. BRICS & Aalborg University, 23 pages.

[10] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[11] Johan Bengtsson and Wang Yi. 'Timed Automata: Semantics, Algorithms and Tools'. In: *Lectures on Concurrency and Petri Nets*. Ed. by Jörg Desel, Wolfgang Reisig and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 87–124.

[12] Abdeldjalil Boudjadar et al. 'Extending UPPAAL for the Modeling and Verification of Dynamic Real-time Systems'. In: *Proceedings of international conference on Fundamentals of Software Engineering*. Tehran, Iran, Apr. 2013.

[13] Patricia Bouyer. 'Timed Automata – From Theory to Implementation'. Tutorial given in Chennai (India), 94 slides. Jan. 2003.

[14] Dragan Bošnački and Dennis Dams. 'Enhancing State Space Reduction Techniques for Model Checking'. In: PhD thesis. Eindhoven University Press, 2002. Chap. Integrating Real Time into SPIN: A Prototype Implementation.

[15] Peter Bulychev et al. 'UPPAAL-SMC: Statistical Model Checking for Networks of Priced Timed Automata'. In: *Proceedings of the 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*. 2012, pp. 1–16.

[16] Mats Carlsson and Lars Johansson. 'Formal Verification of UML-RT Capsules using Model Checking'. 114 pages. MA thesis. University of Gothenburg, Göteborg, Sweden: Chalmers University of Technology, June 2009.

[17] Prakash Chandrasekaran and Madhavan Mukund. 'Matching scenarios with timing constraints'. In: *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems*. Vol. 4202. Lecture Notes in Computer Science. 2006, pp. 98–112.

[18] Alexandre David. 'Checking and Distributing Statistical Model-Checking'. Tutorial, 37 pages. 2012.

[19] Alexandre David and Paul Pettersson. 'UPPAAL Tutorial (Modeling Patterns)'. Tutorial, 11 pages. 2005.

[20] Alexandre David et al. 'Statistical Model Checking for Networks of Priced Timed Automata'. In: *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems*. Vol. 6919. Lecture Notres in Computer Science. Aalborg, Denmark: Springer-Verlag, Sept. 2011, pp. 80–96.

[21] Clara Benac Earle and Lars-Åke Fredlund. 'McErlang - A tool for model checking Erlang programs in Erlang'. Tutorial, 39 pages.

[22] Clara Benac Earle and Lars-Åke Fredlund. 'McErlang: A Tutorial'. Universidad Politécnica de Madrid, Spain. Oct. 2010.

[23] Clara Benac Earle and Lars-Åke Fredlund. 'Verification of Timed Erlang Programs Using McErlang'. In: *Proceedings of the joint conference on Formal Techniques for Distributed Systems*. Vol. 7273. Lecture Notes in Computer Science. Stockholm, Sweden, June 2012, pp. 251–267.

[24] Wan Fokkink, Allard Kakebeen and Jun Pang. 'Adapting the UPPAAL model of a distributed lift system'. In: *Proceedings of the 2nd Symposium on Fundamentals of Software Engineering*. Vol. 4767. Lecture Notes in Computer Science. Tehran, Iran: Springer, Aug. 2007, pp. 81–97.

[25] Bernard van Gastel et al. 'Deadlock and starvation free reentrant Readers-Writers – A case study combining model checking with theorem proving'. In: *Science of Computer Programming* 76.2 (Feb. 2011). Selected papers from the workshops on Formal Methods for Industrial Critical Systems, pp. 82–99.

[26] Bernard van Gastel et al. 'Reentrant Readers-Writers – A case study combining model checking with theorem proving'. In: *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. Ed. by D. Cofer and A. Fantechi. Vol. 5596. Lecture Notes in Computer Science. L'Aquila, Italy: Springer-Verlag, Sept. 2008, pp. 85–102.

[27] Jens Godskesen and Olena Gryn. 'Modelling and Verification of Security Protocols for Ad Hoc Networks Using UPPAAL'. In: *Proceedings of the 18th Nordic Workshop on Programming Theory*. Oslo, Norway, Dec. 2006.

[28] Zonghua Gu and Kang G. Shin. 'Synthesis of Real-Time Implementation from UML-RT Models'. In: *Proceedings of the 2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES)*. Toronto, Ontario, Canada, May 2004.

[29] Henning Günther, Stefan Milius and Oliver Möller. 'On the Formal Verification of Systems of Synchronous Software Components'. In: *Procedings of the 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Vol. 7612. Magdebourg, Germany, Sept. 2012, pp. 291–304.

[30] Ferdy Hanssen, Angelika Mader and Pierre Jansen. 'Verifying the Distributed Real-Time Network Protocol RTnet Using UPPAAL'. In: *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*. 2006, pp. 239–246.

[31] Martijn Hendriks. *Translating UPPAAL to Not Quite C*. Tech. rep. Nijmegen, The Netherlands: University of Nijmegen, Mar. 2011.

[32] Anders Hessel et al. 'Testing Real-Time Systems Using UPPAAL'. In: *Proceedings of the conference on Formal Methods and Testing*. Vol. 4949. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 77–117.

[33] Gerard J. Holzmann. 'The Model Checker SPIN'. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997).

[34] Gerard J. Holzmann. *The SPIN Model Checker – The Primer and Reference Manual*. Addison-Wesley, Sept. 2003.

[35] Henrik Ejersbo Jensen, Kim G. Larsen and Arne Skou. *Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL*. Tech. rep. BRICS RS-96-24. 23 pages. University of Aarhus, Aarhus, Denmark: BRICS, Department of Computer Science, July 1996.

[36] Ke Jiang. 'Model Checking C Programs by Translating C to Promela'. MA thesis. Uppsala University, Uppsala, Sweden, 2009.

[37] Prabhu Shankar Kaliappan. 'Model based verification techniques – State of the Art'. Brandenburg University of Technology, Cottbus, Germany. May 2008.

[38] Joost-Pieter Katoen. 'CTL, LTL and CTL*'. Tutorial, 33 slides. 2007.

[39] Sasha Klüppelholz. 'Tutorial on SPIN and PROMELA'. Institut für Theoretische Informatik, 164 pages. Nov. 2012.

[40] Alexander Knapp and Stephan Merz. 'Model checking and code generation for UML state machines and collaborations'. In: *Proceedings of the 5th Workshop on Tools for System Design and Verification*. Vol. 11. Springer, 2002, pp. 59–64.

[41] Alexander Knapp, Stephan Merz and Christopher Rauh. 'Model Checking Timed UML State Machines and Collaborations'. In: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*. Vol. 2469. Lecture Notes in Computer Science. Springer-Verlag, 2002, pp. 395–414.

[42] Alexander Knapp and Jochen Wuttke. 'Model Checking of UML 2.0 Interactions'. In: *Proceedings of the conference on Models in Software Engineering*. Vol. 4364. Lecture Notes in Computer Science. Springer-Verlag, 2007, pp. 42–51.

[43] Jesper Kristensen, Arne Mejlholm and Søren Pedersen. 'Automatic Translation from UPPAAL to C'. 15 pages.

[44] Ingolf H. Krueger. 'Modeling SW-Architectures using UML-RT and UML 2.0'. Tutorial, 43 slides.

[45] Robert Bellarmine Krug. 'CTL vs. LTL'. Tutorial, 40 slides. May 2010.

[46] Kim G. Larsen, Maris Mikačionis and Brian Nielsen. *UPPAAL TRON User Manual*. CISS, BRICS and Aalborg University, Denmark. June 2009.

[47] Kim G. Larsen, Marius Mikučionis and Brian Nielsen. 'Online Testing of Real-time Systems Using UPPAAL'. In: *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES)*. Vol. 3395. Lecture Notes in Computer Science. Linz, Austria, Sept. 2004, pp. 79–94.

[48] Axel Legay and Alexandre David. 'Statistical Model Checking'. Tutorial, 21 pages. Feb. 2012.

[49] Vasu Hossaholal Lingegowda. 'Building Graphical Promela Models using UPPAAL GUI'. 41 pages. MA thesis. Aalborg University, Aalborg, Denmark: Department of Computer Science, Mar. 2006.

[50] Siegfried Löffler. *From Specificiation to Implementation: A PROMELA to C Compiler*. Tech. rep. Paris, France: École Nationale Supérieure des Télécommunications, 1996.

[51] B. Meenakshi. 'A tutorial on SPIN'. Honeywell Technology Solutions Lab, Bangalore, India (39 pages).

[52] Oliver Möller. 'Structure and Hierarchy in Real-Time Systems'. BRICS Dissertation Series, 247 pages. PhD thesis. Department of Computer Science, University of Aarhus, Denmark, Apr. 2002.

[53] André L. N. Muniz, Aline M. S. Andrade and George Lima. 'Integrating UML and UPPAAL for designing, specifying and verifying component-based real-time systems'. In: *Innovations in Systems and Software Engineering* 6.1-2 (2010), pp. 29–37.

[54] Wojciech Nabiałek, Agata Janowska and Paweł Janowski. 'Translation of Timed Promela to Timed Automata with Discrete Data'. In: *Fundamenta Informaticæ – Concurrency Specification and Programming (CS&P)* 85.1-4 (Sept. 2008). IOS Press, Amsterdam, The Netherlands, pp. 409–424.

[55] Miroslav Pajic et al. ‘From Verification to Implementation: A Model Translation Tool and a Pacemaker Case Study’. In: *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2012, pp. 173–184.

[56] Jun Pang, Bart Karstens and Wan Fokkink. ‘Analyzing the redesign of a distributed lift system in UPPAAL’. In: *Proceedings of the 5th Conference on Formal Engineering Methods (ICFEM)*. Vol. 2885. Lecture Notes in Computer Science. Singapore: Springer, Nov. 2003, pp. 504–522.

[57] ‘Quasimodo Handbook’. In: The Quasimodo Project (Quantitative System Properties in Model-Driven Design of Embedded Systems), 2013. Chap. A First Introduction to UPPAAL.

[58] F. Raimondi. ‘The temporal logic CTL – A gentle introduction’. Tutorial, 35 slides. 2008.

[59] Abhik Roychoudhury. ‘SPIN Model Checker’. 15 pages.

[60] Theo C. Ruys. ‘SPIN Beginners’ Tutorial’. SPIN Workshop, Grenoble, France. Apr. 2002.

[61] Kaarel Saal. ‘Proving properties with UPPAAL’. Technical University of Tallinn, Tutorial, 12 pages. 2004.

[62] *Model Checking UML State Machines and Collaborations*. Electronic Notes in Theoretical Computer Science 55.3 (Oct. 2001): *Workshop on Software Model Checking*.

[63] Gerardo Schneider. ‘Promela Semantics’. Tutorial, 44 pages.

[64] Hans Svensson. ‘QuickCheck-McErlang Integration Tutorial’. 8 pages.

[65] Stavros Tripakis and Costas Courcoubetis. ‘Extending Promela and SPIN for real time’. In: *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 1996, pp. 329–348.

[66] Moshe Vardi. ‘Branching vs. Linear Time: Final Showdown’. In: *Proceedings of the 7th conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2031. Lecture Notes in Computer Science. Genova, Italy: Springer-Verlag, Apr. 2001.

[67] Hao Zheng. ‘Introduction to Computer Aided Verification’. Tutorial, 41 pages.

[68] Hao Zheng. ‘Temporal Logics’. Tutorial, 20 slides.

[69] Einat Zuker. ‘UPPAAL – Verification of real-time systems’. Tutorial, 39 pages.