# From UML State-Machine Diagrams to Erlang[*]

**Ricardo J. Rodríguez, Lars-Åke Fredlund, Ángel Herranz**

Babel Group, DLSIIS, Facultad de Informática
Universidad Politécnica de Madrid, Spain
{rjrodriguez, lfredlund, aherranz}@fi.upm.es

**Abstract:** The Unified Modelling Language (UML) is a semi-formal modelling language useful for representing architectural and behavioural aspects of concurrent and distributed systems. In this paper we propose a transformation from UML State-Machine diagrams to Erlang code. Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing, and with good support for concurrency and distribution. The contribution of this transformation is twofold: it can reduce development time, and moreover it permits us to validate UML diagrams at an early development stage through the use of Erlang-based model checking techniques.

**Keywords:** UML, Erlang, automatic code generation

## 1 Introduction

The Software Development Life-Cycle (SDLC) [GJM02] is a process methodology that defines the phases involved for developing and maintaining software systems and software code. One of these phases is the *deployment and maintenance* phase, where the software is deployed into production and it is maintained to cope with possible faults or minor changes on current requirements. To verify the correctness of the developed software before it is deployed into production can save efforts as problems are detected before arise. A widely used approach for the formal verification of software is model checking [CGP99, BBF$^+$10]. Model checking automatically provides complete proofs of correctness, and explains using counter-examples why a software system or software code is not correct.

In this paper, we provide an approach to be able to verify a software design during design phase through model-checking techniques (more precisely, by using Erlang-based model-checking techniques). This early verification step would improve the SDLC, as design problems can be detected during design phase and not at the end of the process, thus saving efforts and economic losses due to redesign and/or reimplementation. This approach is based in a transformation from UML diagrams (namely, UML State-Machine diagrams (UML-SMs)) to Erlang code. Since a number of effective verification and validation tools are available for Erlang, e.g., the Quviq QuickCheck random testing tool [AHJW06] and the McErlang model checker [FS07], we can use these tools to verify the transformed UML diagram. The well-known *dining philosophers* problem is used as a motivation example in this paper.

[*] This work was partially supported by ARTEMIS Joint Undertaking nSafeCer under grant agreement no. 295373 and from National funding.

The UML (Unified Modelling Language) [OMG05] is a standard *de facto* modelling language, commonly used by software engineers for high-level system design. UML provides several types of diagrams which allow to capture different aspects and views of the system. A UML model of a system consists of several diagrams which represent the functionality of the system, its static structure, the dynamic behaviour of each system component and the interactions among the system components.

Erlang [Arm97] is a functional and concurrent programming language commonly used for the development of distributed, fault-tolerant, soft-real-time, and non-stop applications [Wik94, AVWW96, Lar09].

The contribution of this transformation is twofold: firstly, it can minimise development time as it automatically generates Erlang code from UML-SMs, and secondly it enables us to apply Erlang-based model-checking techniques to UML specifications. This transformation can save efforts and future costs produced by a redesign and/or reimplementation of a designed software at the end of its SDLC. Besides, to the best of our knowledge, our proposal is the first one the that provides an automatic generation of Erlang code from UML-SMs.

The remainder of this paper is as follows. Section 2 introduces some concepts in more detail, such as UML-SMs and the Erlang programming language. Then, Section 3 introduces our approach to transform a UML-SM into Erlang code. Section 4 discusses some approaches in the literature relevant to our work. Lastly, Section 5 provides concluding remarks and future research directions.

## 2 Background

In this section we provide some background needed to follow the rest of this paper. In first place, we introduce the UML State-Machine diagrams (UML-SMs). Finally, the Erlang programming language is introduced.

### 2.1 UML State-Machine Diagrams

The Unified Modelling Language (UML) [OMG05] provides several types of diagrams that allow to capture different aspects and views of the system. UML diagrams can be categorised in: *static diagrams*, which are intended to model the structure (logical and architectural) of the system; *behavioural diagrams*, which are intended to describe system dynamics (its dynamic behaviour and the interactions among the system components); and *organisational diagrams*, which allows to reduce the complexity of the system by organising it in a modular way.

In this paper, we focus on UML State-Machine diagrams (UML-SMs) that model the dynamic behaviour of a system through finite state-transitions. We assume that such a system is composed of several components intercommunicated among them by sending asynchronous messages. The UML-SM semantics is the one proposed by Harel in [Har87] for *state-charts*. Graphically, a UML-SM is a directed graph in which nodes denote states and connectors denote state transitions. A state is represented by a rounded rectangle labelled with a state name, while transitions are represented by arrows labelled with the triggered events followed (optionally) by effects. A transition can optionally have a guard, indicated between brackets. The initial transition origi-

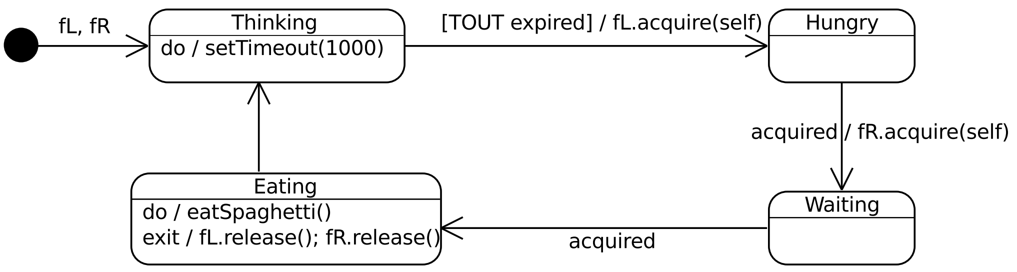
fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

Figure 1: *Dining philosophers*: A UML State-Machine diagram (UML-SM) of a philosopher.

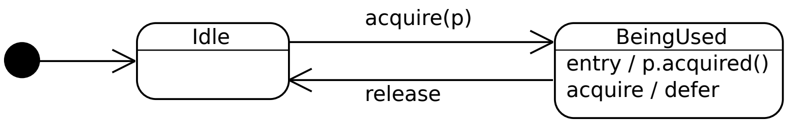

Figure 2: *Dining philosophers*: A UML State-Machine diagram (UML-SM) of a fork.

nates from a solid circle and sets the default state where system begins. The final transition ends in a solid circle surrounded by an empty circle.

Figures 1 and 2 show UML-SMs describing the behaviour of a philosopher and a fork of the “dining philosophers” example. We assume that the thinking time of each philosopher is equal to 1000 units of time. We also assume that each philosopher first grabs his left fork, and then he waits holding it until his right fork can be acquired. Note that this “eating protocol” leads to a deadlock, i.e., a situation where philosophers are waiting for some event which will never happen.

The initial transition of a philosopher in Figure 1 needs two parameters, `fL` and `fR`, which represents respectively his left and right forks. Note that in the corresponding UML class diagram `Philosopher` class would have a double association with `Fork` class, thus this initial parameters would be not longer needed. The first state of a philosopher is *Thinking* state. This state has a do action, which sets a timeout of a duration equal to 1000 units of time. The single exit transition of this state has a guard that indicates the timeout has expired, and an effect of sending an asynchronous `acquire(self)` message to `fL`. After sending such a message, a philosopher moves to *Hungry* state. From this state he moves to *Waiting* state when an event `acquired` is received, and sends also an `acquire(self)` message to `fR`. Once the `acquired` event is received, a philosopher enters to *Eating* state, where starts the `eatSpaghetti()` do activity. When this activity has finished, the `release()` message is sent to both `fL` and `fR` before existing the state and move to *Thinking* state, starting the cycle again.

The behaviour of a fork, depicted in Figure 2 is much simpler. It starts in *Idle* state, and moves to *BeingUsed* when it receives an `acquire(p)` event. This parameter `p` is the philosopher who triggers the event. When entering in the *BeingUsed* state, it sends an `acquired` event to `p`, as indicated by the `entry` activity. The statement `acquire / defer` in *BeingUsed* indicates that all `acquire` events received in this state are deferred, i.e., stored for later handling. When the `release` event is received, a fork moves to *Idle* state, starting its cycle again. Note that a deferred event could trigger immediately the transition to the `BeingUsed` state.

### 2.2 The Erlang Programming Language

Erlang [AVWW96] is a functional concurrent programming language created by the Ericsson company in the 1980s. Ericsson is still maintaining the main Erlang implementation, but it is available as open source since 1998. The chief strength of the language is that it provides excellent support for concurrency, distribution and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. Concurrency is achieved by lightweight processes communicating through asynchronous message passing. Although Erlang is not a new language, it has experienced considerable growth in users in recent years. This is due in most part to its focus on message passing instead of variable sharing as the main communication mechanism, which enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

Today Erlang is used by Ericsson and many other companies (T-Mobile (UK), and many smaller start-up companies such as e.g. Interoud in Spain and Klarna in Sweden) to develop industrial applications, often implementing crucial internet server-side applications. Examples include a high-speed ATM switch developed at Ericsson with over a million lines of Erlang code which had to meet very challenging requirements on software reliability and overall system availability [BRA$^+$99, WAB02], parts of Facebook chat, Apache CouchDB – a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API, etc.

## 3 A Transformation Approach

This section introduces the transformation approach from UML State-Machine diagrams (UML-SMs) to Erlang code. The aim of this section is to show how our approach works. Basically, for each UML-SM, an Erlang module is generated following the steps described in Algorithm 1.

The transformation algorithm of the approach is described in Algorithms 1 and 2, as pseudo-code. As input, it receives a UML-SM $\mathscr{D}$. As output, it returns an Erlang module $\mathscr{E}$ with the same behaviour as $\mathscr{D}$.

Let us recall how this transformation algorithm works by means of the running example as defined by the UML-SMs depicted in Figures 1 and 2. The Erlang code generated by Algorithms 1 and 2 is listed in Code 1 and 2.

First, let us consider the UML-SM in Figure 1 that describes the behaviour of a philosopher. In the translation, the two initial parameters `fL`, `fR` of the initial transition are stored in a set $\mathscr{P}$. The set $\mathscr{P}$ is passed along as a parameter to `start` and of all other Erlang functions. In the set $\mathscr{E}$ we collect all the events of the UML-SM that is being transformed. This is needed because, unlike Erlang messages, UML events sent between different UML-SMs are lost if they are not handled in the current state of a UML-SM. Then, the **`module`** and **`export`** Erlang compiler attributes are output. Each UML-SM transformed to Erlang has a dedicated function `start` which is used to create a new Erlang process executing the (translated) state machine.

The first state is the *Thinking* state. An Erlang function is defined with that name, and receiving the parameters contained in $\mathscr{P}$. The `do` activity `setTimeout` and the conditional exit trigger `TOUT expired` are assumed to represent a timeout activation and expiration in UML-SMs, and generate a **`receive`** clause **`after`** with a timeout corresponding to the parameter

```
Input: A UML-SM $\mathscr{D}$
Output: An Erlang source file $\mathscr{E}$
Store in $\mathscr{P}$ the parameters of the initial transition in $\mathscr{D}$
Create a module named as $\mathscr{D}$ and an export declaration of start function
Create a start function with parameters $\mathscr{P}$ and invoke the first state $\mathscr{S}$ of $\mathscr{D}$
Create a set of events $\mathscr{E}$ containing all trigger events in $\mathscr{D}$
forall the state $\mathscr{S}$ of $\mathscr{D}$ do
    Create a function $\mathscr{F}$ with parameters $\mathscr{P}$ named as state $\mathscr{S}$
    Execute Algorithm 2 with entry activity as input (if any), and invoke the result
    Execute Algorithm 2 with do activity (if any)
    if state $\mathscr{S}$ has not a do activity called setTimeout then
        Execute Algorithm 2 with do activity, and invoke the result
    end
    if state $\mathscr{S}$ has output transitions then
        if some output transition has a guard and/or a triggered event then
            Create an Erlang receive skeleton
            forall the output transition $\mathscr{O}$ of $\mathscr{S}$ triggered by an event do
                Map the event of $\mathscr{O}$ to an Erlang message reception
                Create a when clause if output transition $\mathscr{O}$ has a condition guard
                Execute Algorithm 2 with output transition $\mathscr{O}$, and invoke the result
                Invoke the Erlang function that represents the exit action (if any)
                Invoke the function $\mathscr{F}'$ representing the destination of $\mathscr{O}$
            end
            if exists some event in $\mathscr{E}$ not deferred in $\mathscr{S}$ and not handled then
                Create an Erlang code to ignore such an event by invoking $\mathscr{F}$
            end
            if state $\mathscr{S}$ has a do activity called setTimeout and an output transition $\mathscr{O}$ is
            triggered when the timeout has expired then
                Create an after clause
                Execute Algorithm 2 with output transition $\mathscr{O}$, and invoke the result
                Invoke the Erlang function that represents the exit action (if any)
                Invoke the function $\mathscr{F}'$ representing the destination of $\mathscr{O}$
            end
        end
        Invoke the Erlang function that represents the exit action (if any)
        Invoke the function $\mathscr{F}'$ representing the destination of $\mathscr{O}$
    else
        Invoke the Erlang function that represents the exit action (if any)
    end
end
```

**Algorithm 1:** A transformation from UML State-Machine diagrams to Erlang code.

```
Input: A state $\mathscr{S}$ and an activity $\mathscr{A}$
Output: A piece of Erlang code
1 if $\mathscr{A}$ concerns another state machine $\mathscr{SM}'$ then
2     Send a message with the mapped $\mathscr{A}$ (and parameters) to the Erlang process
      corresponding to state machine $\mathscr{SM}'$
3 else
4     Map $\mathscr{A}$ to Erlang code
5 end
```
**Algorithm 2:** Transformation of any activity to Erlang code.

Code 1: Erlang code generated from basic UML-SM in Figure 1.

```
-module(philosopher).
-export([start/2]).

start(FL, FR) ->
  spawn(fun() -> thinking(FL, FR) end).

thinking(FL, FR) ->
  receive
    X -> thinking(FL, FR)
    after 1000 -> FL!{acquire, self()}, hungry(FL, FR)
  end.

hungry(FL, FR) ->
  receive
    acquired -> FR!{acquire, self()}, waiting(FL, FR)
  end.

waiting(FL, FR) ->
  receive
    acquired -> eating(FL, FR)
  end.

eating(FL, FR) ->
  eatSpaghetti(),
  FL!release,
  FR!release,
  thinking(FL, FR).
```

Code 2: Erlang code generated from basic UML-SM in Figure 2.

```
-module(fork).
-export([start/0]).

start() ->
   spawn(fun() -> idle() end).

idle() ->
  receive
    {acquire, P} -> beingUsed(P);
    X -> idle()
  end.

beingUsed(P) ->
  P!acquired(),
  receive
    release -> idle()
  end.
```

of the `setTimeout` activity. As there are events that are not handled in this state and it has output transitions triggered by events, a **`receive`** clause receives any value and recurses. The effect `fL.acquire(self)` is transformed to a signal `cquire, self()acquire,` **`self`**`()` sent to `fL` in Erlang code. Finally the function that represents the *Hungry* state is called. This state has only an exit transition triggered by the `acquired` event; the construct is translated into a **`receive`** clause receiving `acquired`. The effect of this event sends a signal `cquire, self()acquire,` **`self`**`()` to `fR` and calls the function that represents the *Waiting* state. As previously, a **`receive`** clause is output where receiving `acquired` results in a call to the function corresponding to the *Eating* state. Finally, the `do` and `exit` activities are transformed to their equivalent Erlang code. More precisely, the exit activity that involves invocations of `release()` method in `fL` and `fR` are transformed to `FL!release` and `FR!release` in Erlang.

Let us consider now the UML-SM of the behaviour of a fork (Figure 2). The set of parameters of the initial transition (none) is stored in $\mathscr{P}$. Then, `module(fork)` and `export([start/0])` compiler attributes are created in Erlang code. After this, the `start` function spawns a new process which behaves as the initial state of the state machine (the *Idle* state). As before, all the events of this UML-SM are stored in a set $\mathscr{E}$ (in this case, `acquire` and `release`).

The UML-SM depicted in Figure 2 has two states, *Idle* and *BeingUsed*. Let us focus first on the *Idle* state. A function `idle()` is created, without parameters, representing this state. This state does not have an `entry` activity, nor a `do` activity. It has just an output transition triggered by an event `acquired(p)`, that is mapped to the reception of a message with a parameter `P`. When received, it invokes the function `beingUsed()`, which represents the other state of $\mathscr{D}$. As the event `release` is not handled in this state, an clause that handles any message is output invoking itself.

The `entry` action of *BeingUsed* is mapped to a sending of an `acquired` message to `P`, as identified by Algorithm 2). As before, it has an output transition triggered by the `release` event. When received, it invokes the `idle()` function, returning to the initial state. In this case, as the `acquire` event is deferred, there is no need to include a clause receiving any other message.

Finally, note how the *absence* of an `aquire / defer` statementin *idle* gives rise to an Erlang **`receive`** clause `X ->idle()` in `Idle`, whereas the *occurrence* of an `aquire / defer` statement in the state `BeingUsed` results in the absence of the corresponding receive clause `X ->beingUsed()`. Thus a signal `cquire,P'acquire,P'` that arrives in the state `BeingUsed` is not thrown away but *deferred*, i.e., stored and later handled in the `Idle` state.

The Erlang code resulting from the transformation can be checked using the `McErlang` model checker [FS07] without modifications. We can for example compose a system of `N` philosophers in Erlang using the function call `run(N)` below (located in the module `dining`):

```
run(N) ->
  Forks = lists:map (fun (_) -> fork:start() end, lists:seq(1,N)),
  lists:foreach
    (fun ({L,R}) -> philosopher:start(L, R) end, adjacent(Forks)).

adjacent([])      -> [];
adjacent([X|Xs]) -> lists:zip([X] ++ Xs, Xs ++ [X]).
```

Invoking `McErlang` to check whether a configuration of two philosphers is deadlock free has the following result:

```
$ erl
> mce:start
    (#mce_opts{program=fun () -> dining:run(2) end,
               monitor=mce_mon_deadlock}).
...
*** Monitor failed
monitor error:
deadlock
```

As we commented in Section 1, the dining philosophers problem modelled as illustrated in Figures 1 and 2 has a deadlock problem which evidently was detected by the `McErlang` model checker.

## 4 Related Work

Automatic code generation from formal specifications is considered one of the silver bullet of software engineering. In this respect, state machines is one of the most successful formalism since its semantics is directly executable. UML state-charts is a variation of state machines so it is not strange that countless automatic code generation approaches can be found in the literature.

The single-threaded approach is simple and the *inherent synchronous* view of operations is relatively straightforward to compile into ordinary programming language constructs: case statements and function calls [HG96].

Here we restrict the consideration of related works to the more complex problem of automatic code generation for state-charts with multi-threaded behaviour and asynchronous communication (signals).

Most of such works rely on the implementation of a general interpreter that captures the executable semantics of a particular specification language. Given a set of concrete state machines written in a specification language, a translator compiles them into structures to represent states, nodes and transitions that the general interpreter operates on. Relevant works following this road are for example [PM03, AFL$^+$10] (C code from extended finite state machines with a synchronous semantics for the moment), [NT03, KNNZ00] (asynchronous semantics with a state table to reacts to events), [TKUY01] (based on design pattern forms implementing state machines), [KM02] (with a Java thread per state-chart and Java objects to represent event queues).

All these works relies on the ability of the interpreter to manage multi-threading and event queues, two aspects where the Erlang programming language is particularly strong. As a consequence of the excellent support available in an Erlang run-time system for multiprogramming, in our approach no such interpreter is needed. Instead state-machine concepts such as machines, state, nodes and transitions are directly compiled into Erlang efficient constructs: processes, call by value function calls, efficient message passing, and event queues. As a result we expect our implementation of UML state-machines in Erlang to be very high-performing with regards to performance criteria such as the maximum number of messages sent per seconds, the maximum number of concurrent state machines supported, etc. We aim at performing some experiments for comparing performance results as future work.

Other works that worth to be mentioned transform state-charts to other models that are verifiable, such as [GLM99, LMM99, CEC12]. In [GLM99], a branching time model-checking approach is presented for automatically verify the correctness of UML-SMs using JACK as verification environment. The work in [LMM99] introduces an approach where linear-time model checking of UML-SMs is addressed by translating them to PROMELA, the modelling language of the SPIN model-checker. In [CEC12], UML class diagrams, UML-SMs and UML Communication diagrams are considered and automatically transformed to a model that can be verified by means of Maude LTL model checker.

## 5 Conclusions

The Unified Modelling Language (UML), a *de facto* standard as semi-formal modelling language, provides several types of diagrams that can capture different aspects and views of a modelled system. For instance, UML is useful for representing architectural and behavioural aspects of concurrent and distributed systems.

This paper explores the idea of automatic transformation from UML State-Machines diagrams (UML-SMs), which describe the dynamic behaviour of a system, to the Erlang programming language. Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing, and with good support for concurrency and distribution. The contribution of this transformation can be summarized:

- It can reduce development time by automatically generating Erlang skeleton code from UML diagrams; and

- it enables validation of UML-SMs at an early development stage through the use of Erlang-based model checking and testing techniques.

Besides, this transformation also provides an alternative behavioural code pattern complementing frequently used Erlang code patterns (libraries) such as e.g. the generic server library `gen_server` and the finite-state machine library `gen_fsm`.

As future work, we plan also to significantly extend the scope of the transformation to provide support for additional UML-SM constructs. For example, we would like to provide support for preemptive UML-SM activities in Erlang. Moreover, supporting UML-SM constructs such as e.g. sub-states, entry, exit or alternative points, and call actions would be highly beneficial. Lastly, we aim at developing a plugin for some UML CASE tool (such as Papyrus, MagicDraw or ArgoUML) to package the transformation.

## Bibliography

[AFL$^+$10] D. Arney, S. Fischmeister, I. Lee, Y. Takashima, M. Yim. Model-Based Programming of Modular Robots. In *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. Pp. 66–74. 2010.

[AHJW06] T. Arts, J. Hughes, J. Johansson, U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (ERLANG)*. Pp. 2–10. ACM, New York, NY, USA, 2006.

[Arm97] J. Armstrong. The development of Erlang. *ACM SIGPLAN Notices* 32(8):196–203, Aug. 1997.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[BBF$^+$10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[BRA$^+$99] S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, G. Wicklund. AXD 301: A new generation ATM switching system. *Computer Networks* 31(6):559–582, 1999.

[CEC12] W. Chama, R. Elmansouri, A. Chaoui. Model Checking and Code Generation for UML Diagrams using Graph Transformation. *International Journal of Software Engineering & Applications (IJSEA)* 3(6):39–55, December 2012.

[CGP99] E. M. Clarke, O. Grumberg, D. Peled. *Model Checking*. The MIT Press, 1999.

[FS07] L.-Å. Fredlund, H. Svensson. McErlang: a model checker for a distributed functional programming language. *ACM SIGPLAN Notices* 42(9):125–136, 2007.

[GJM02] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[GLM99] S. Gnesi, D. Latella, M. Massink. Model Checking UML Statechart Diagrams using JACK. In *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*. Pp. 46–55. 1999.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8(3):231–274, June 1987.

[HG96] D. Harel, E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*. Pp. 246–257. IEEE Computer Society, Washington, DC, USA, 1996.

[KM02] A. Knapp, S. Merz. Model checking and code generation for UML state machines and collaborations. *Proceedings of 5th Workshop on Tools for System Design and Verification, Technical Report* 11:59–64, 2002.

[KNNZ00] H. J. Köhler, U. Nickel, J. Niere, A. Zündorf. Integrating UML diagrams for production control systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Pp. 241–251. ACM, New York, NY, USA, 2000.

[Lar09] J. Larson. Erlang for Concurrent Programming. *Communications of the ACM* 52(3):48–56, March 2009.

[LMM99] D. Latella, I. Majzik, M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing* 11(6):637–664, 1999.

[NT03] I. A. Niaz, J. Tanaka. Code Generation From Uml Statecharts. In *Proceedings of the 7th IASTED International Conference on Software Engineering and Application (SEA)*. Pp. 315–321. 2003.

[OMG05] OMG. Unified Modelling Language: Superstructure. Object Management Group, July 2005. Version 2.0, formal/05-07-04.

[PM03] G. Pintér, I. Majzik. Automatic Code Generation Based on Formally Analyzed UML Statechart Models. In Tarnai and Schnieder (eds.), *Formal Methods for Railway Operation and Control Systems (Proceedings of the FORMS-2003 Conference)*. Pp. 45–52. LH́armattan, Budapest, Hungary, May 15-16 2003.

[TKUY01] T. Tomura, S. Kanai, K. Uehiro, S. Yamamoto. Object-Oriented Design Pattern Approach for Modeling and Simulating Open Distributed Control System. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA)*. Pp. 211–216. IEEE, 2001.

[WAB02] U. Wiger, G. Ask, K. Boortz. World-class product certification using Erlang. *SIGPLAN Not.* 37:25–34, December 2002.

[Wik94] C. Wikström. Distributed Programming in Erlang. In *Proceedings of the 1st International Symposium on Parallel Symbolic Computation (PASCO)*. Pp. 412–421. 1994.