# Functional programming and logic decrease the use of the most important part of our system

Thomas Arts

Ericsson
Computer Science Laboratory
Box 1505, 125 25 Älvsjö, Sweden
thomas@cslab.ericsson.se
http://www.ericsson.se/cslab/~thomas/

**Abstract.** Several years of experience with the functional language Erlang have learned Ericsson that it is highly beneficial to use this language for programming control software for large systems. Systems that could not be built before, have been constructed in less time and with fewer lines of code than one would need with conventional languages. The success of Ericsson in the business area of telephone switches is partly because of their solid fault tolerant architecture, both in hardware and in software. A lot of time and money have been invested in the development of this fault tolerant architecture, all to catch these errors that are overlooked in numerous tests. By using Erlang and its extensive libraries, the number of these uncaught errors decreases; the fault recovery mechanism of the system is used less. One saves on maintenance costs and the overall performance of a system increases. The additional use of formal verification aims on reducing even more the number of uncaught errors. Both theorem proving and model checking on Erlang code have been explored. We consider them as supplementary techniques, both used in the area they fit best. Aiming, thus, on using the fault tolerant part of the software even less.

## 1 Introduction

Development of software for a telecommunication system might be a bit more tricky than development of an arbitrary office application, because of the high demand on availability of the system where the software executes in. As in any other fault tolerant system, telecommunication systems are equipped with robust and redundant hardware. One hardware failure has at most a performance effect on the overall system. The hardware is controlled by a control system, which typically is a complicated piece of software. The control software monitors the hardware and configures the system for a certain task. These configuration changes might be frequent, e.g. when setting up a connection and tearing it down. Clearly, the task of control software is also to act in case of failing hardware. Redundant hardware may take over the role of the failing part automatically, but, even then, configuration data must be updated and the system as a whole should start acting in a different mode, e.g. giving priority to certain maintenance tasks and reducing the number of accepted calls.

It need not be surprising that the control software can be split in two parts, one dealing with fault tolerance and another with the actual functionality. Knowledge of how to write the fault tolerant and robust part is one of Ericsson's assets; building a switch is one thing, building a switch that never goes down, not even for a software update, is another. In the development of the functional language Erlang [1, 16] this knowledge is extensively used. Together with a well-chosen set of design principles, Erlang has been made very suitable for writing fault tolerant control software. In such software there might still be errors in the code. In fact, a philosophy behind the fault tolerant software is to assume that not only every piece of hardware, but also software may fail. Software is therefore divided in pieces, called components or blocks, that interact with each other. Whenever one software component fails, the rest of the software tries to recover from that. However, compared to the use of more conventional languages for writing control software, such as C and C++, the number of errors in comparably large projects is fewer when Erlang is used. In the rest of this paper it is described how the use of Erlang reduces the number of errors and how we have experimented with theorem proving and model checking on Erlang software in order to decrease the number of errors even more.

With fewer errors in the code, the software in the fault tolerant part of the system is used less, but not useless. Errors may still occur and a good architecture on trapping them remains necessary. What we gain, however, is an increase in performance (recovering from errors is computational expensive) and a decrease in maintenance costs.

## 2 Fewer errors in Erlang programs

Experience of using the functional language Erlang for control software learns that declarative languages have the potential of being very suitable for this task. Data on a real comparison of product development in both Erlang and a competitive other language is rare. Parallel developments of a large system is a very expensive experiment and conclusion one might draw from the results are discussible: Were the programming teams equally experienced? Did the design suit one of the implementation languages better? *etc.* On top of the problems of evaluating the results, one is also faced with requirements that evolve over time: the secondary requirements such as library support, available components, platform compatibility, debug possibilities *etc.*, and ternary requirements, like available skilled programmers, career possibilities for designers and project managers that try a new technology, *etc.*

Only a few times a small part of a system has been designed in two different languages for comparison reasons (cf. [2]). Therefore, we base our judgment on the opinion of experienced designers, programmers and project leaders (e.g. [15, 31]), that have seen many different projects, in which many techniques and languages have been used to implement the control software.

### 2.1 Programming declarative

The principal idea behind declarative languages is to separate out logic and control, so that the programmer is free to concentrate on the logic of a problem without being distracted by control considerations. Erlang is a declarative language, since it does support this uncoupling of logic and control with respect to computations. However, for fault tolerant issues, it is utmost important to keep control over which process runs on which machine and where critical data is stored. Therefore, the communication part allows control from the programmer. Processes have a unique identity and creation of processes is done explicitly. Messages are sent to an explicitly specified process and the programmer can determine at which moment the process is ready to read a message from its mailbox. Declarative in this setting is, though, that programmers need not concern about the memory a process or message uses; this is automatically assigned and garbage collected after use.

Declarative programming reduces the number of errors in a system, in particular serious errors that originate from programmers doing the memory management themselves, e.g. memory leaks, pointer errors, *etc.* A declarative program contains fewer lines of code than a similar imperative program. For Erlang, we experience a factor four to ten reduction in the number of lines of code in a large system [31]. At the same time we also experience that the errors per lines of code ratio is roughly the same, which seems strange if classical, memory related, errors are removed. This, though, should be seen in a development process, where one starts counting the errors when the system is shipped for testing, i.e., a reasonably stable system. This reasonably stable system is obtained earlier when using Erlang.

Thus, we get roughly four to ten times fewer errors in a system developed in Erlang. Furthermore, error corrections do usually not give rise to a whole series of side-effects, introducing and visualizing more errors. This is in line with what one expects from using a declarative language.

### 2.2 Following the implicit model

A programming language that supports an implementation approach that intuitively reflects the real problem has a strong advantage over a language that lacks a direct mapping of problem into implementation. Erlang's model of concurrency, with the possibility of having thousands of light-weight concurrent processes at the same time, resembles closely to the description of the connections that have to be controlled. Every connection is specified as an independent entity and as such they can be programmed in Erlang. Without the support for having so many light-weight processes, it is possible to gather a bunch of concurrent activities in one thread, but this comes with the risk of introducing extra errors, more complicated debugging and above all, a more difficult system to make fault tolerant.

If event based communication is assumed by the hardware specifications, one can implement this by using synchronous communication, but the system will get unnecessary complicated. Hence, Erlang's asynchronous communication fits very well in this setting.

Erlang's concurrency model fits very well the kind of problems that one is modeling in control software of highly concurrent systems. This helps in simplifying the programs and therefore reduces the number of errors one expects to find.

### 2.3 Component based development

Libraries, ready-to-use components and design principles play a key factor in efficient software development. For Erlang this was realized from the beginning and a lot of effort has been put in the production of a large set of libraries, referred to as *the Open Telecom Platform*, OTP for short [30]. These libraries include functions on data structures, but also complicated components, such as servers, supervisors and finite state machines. A few design principles describe how the libraries and components are used best.

The typical control software consists for the larger part of processes that implement a server or finite state machine. Such components that occur very frequent in a software system can better be isolated and implemented once and for all. Clearly, one can only provide a kind of generic implementation that needs to be instantiated with the specific functionality, but that is then also exactly what has been done. For example, there is a concept of a generic server, which provides synchronous and asynchronous communication, debugging support, error and timeout handling, and other administrative tasks. In order to obtain the required server functionality, the programmer provides an instantiation for this generic server. Since only the code for the instantiation has to be provided and since the generic part of the code is very well tested, the use of these components reduces the number of errors drastically. Moreover, if an error is still present in the generic part, it appears for example in all finite state machines in the system and is therefore relatively easy to find and in particular easy to fix for all state machines at the same time.

The supervisors are generic components used for fault tolerance. All processes in a system occur somewhere as a node in a supervision tree, and are monitored by a supervisor process. The supervisor nodes are implemented in a very simple, straight-forward way such that no unnecessary errors are introduced.

Software components are for a large software project what a declarative language is for a small program. The programmer can concentrate on functionality, not being bothered by details. Unfortunately, literature on components for functional languages is rather poor. It seems that only the small scale components like `map`, `foldl`, `farm`, `parallel` and such are commonly accepted. One reason might be that specifying communication between components depends on the underlying framework to guarantee referential transparency.

### 2.4 Short design cycles

Notwithstanding the (in academia) popular rules on how to design a system: starting from an abstract model and refining it to specifications that are only in the very end implemented in a programming language, systems used to be built differently in practice. The implementation of a system is generally started as early as the design. One of the main reasons for this is that models lack a lot of information, e.g. on performance, memory usages and realizability of the design. As important, probably, is the fact that a running system, even a buggy one, is better understood by the customers, who then can decide upon a drastic change of the requirements.

Erlang in combination with OTP bridges the gap between refinement and implementation, one could say. Because of the high level of abstraction in the components and the language, one easily implements a rather abstract design. In typical Erlang projects designers move around, working on different parts of the software and integration of these parts takes place on a daily basis. This way of working has great similarity to the so called *Extreme Programming* [8]. The advantage of this way of working is that many design errors already are eliminated in an early stage in the project.

Short cycles between design and test secure systems for design errors, and at the same time, a lot of small errors in the code already are removed by the many tests that are performed during development.

## 3 Towards error free Erlang programs

The advantages of the use of a language like Erlang in combination with design principles and components have been shown in many projects (cf. [15]). Nevertheless, the produced software is not error free and a lot of time and money is spent on testing the software and fixing errors that are found.

In order to reduce in particular the maintenance costs, we strive to develop tools that assist in finding errors as early as possible. However, these tools may not interfere with the rapid development of the software. The choice between early on the market and lower maintenance costs is nowadays always in favor of the first alternative; too late on the market implies cancellation of the project and hence no maintenance at all.

In this context, we focus in particular on errors that are hard to find with testing, such as synchronization errors. The more expensive it is to repair or find a certain error, the more interesting it becomes to develop a technique to find it.

It is legitimate to wonder why Erlang is dynamically typed, whereas many functional languages have advanced static type systems to help in detecting errors in an early stage. There have definitely been attempts to add a type system to the language (e.g. [28]). However, adding a suitable type system to a language that is developed with dynamic typing in mind is a challenge. Whereas we see a dazzling amount of papers appear on new type systems or minor changes to existing type systems, there seems poor academic interest in developing a type system for an industrial functional language. In this paper we do not discuss the attempts within Ericsson to develop a suitable type system, but we focus on two other verification techniques.

For the purpose of finding the most tricky errors in an implementation, both model checking and theorem proving have been explored. For the first technique we have been able to use the CÆSAR/ALDÉBARAN development package [17] to develop a kind of push-button verification tool [3], whereas for the second technique Ericsson developed in cooperation with the Swedish Institute of Computer Science (SICS) a special theorem prover that is aware of the operational semantics of Erlang [5, 29, 19]. We aim to use existing tools and techniques where possible, but even though many good tools are available, there is normally a lot of work left before a tool is incorporated in an existing development environment. Using a modeling language, for example, is out of the question, since the disadvantages outweigh the benefits. Feedback of the tools should be presented in a way that Erlang programmers can relate it immediately to their programs. When errors are presented, they need to be real errors and not false positives. The time spent on using a tool should be marginal to the time it would take to find the error in another way (for example by manual code inspection).

## 4 Model checking Erlang programs

Using model checking for the formal verification of software is by now a well known field of research. Basically there are two branches, either one uses a specification language in combination with a model checker to obtain a correct specification that is used to write an implementation in a programming language, or one takes the program code as a starting point and abstracts from that into a model, which can be checked by a model checker. In either case, the implementation is not proved correct by these approaches, but when an error is encountered, this may indicate an error in the implementation. As such, the use of model checking can be seen as a very accurate debugging method.

For the first approach, one of the most successful of the many examples is the combination of the specification language Promela and model checker SPIN [24]. The attractive merit of Promela is that this language is so close to the implementation language C, that it becomes rather easy to derive the implementation from the specification in a direct, fault free way. However, even this popular approach has tendency to be overtaken by the approach of translating C to Promela [25]. For this second approach of translating source code to a modeling language there are many examples, among which PathFinder [23] and Bandera [13] starting from Java code. Our approach could be added to this list, probably with the difference that we use the knowledge of the occurring design patterns used in the Erlang code to obtain smaller state spaces (cf. [7]). We follow a similar approach to the translation of Java into Promela, checked by SPIN [23]; however, we translate Erlang into $\mu$CRL [21] (a process algebra with data [20]) and model check properties in alternation free $\mu$-calculus, by using CÆSAR/ALDÉBARAN [17].

There is also another approach to perform model checking on Erlang code, presented by Huch [26]. The major differences between Huch's and our approach are found in the abstraction. We only abstract according to the given semantics of the design patterns, whereas Huch does not take design patterns into account. Huch's approach is based on abstract interpretations, abstracting from both data and control. This abstraction is sound, but might result in properties that do not hold for the abstraction, even though they hold for the Erlang program. Moreover, the presented approach is restricted to Erlang programs that are tail recursive, but this has been extended recently [27].

**Case study** Inspired by an access manager algorithm in one of the large switches programmed in Erlang [10], we started to develop in parallel a verified version of this algorithm[1] and a tool to automatically translate this kind of code into $\mu$CRL. We used the generic server design principle, higher-order functions, supervision trees and many other features that are typical for the kind of code we find in products [3]. We used tools developed by the Research Institute for Mathematics and Computer Science (CWI) for generating the labeled transition system and hiding (abstraction from) internal actions that were not of interest for the property at hand [11]. We wrote a small program to translate the labels in the transition system back to Erlang terms, such that it appears as the state space of an Erlang program. The CÆSAR/ALDÉBARAN toolset contains many easily accessible features of which we used the state space visualization, the minimization with respect to all kinds of bisimulation, the simulator and, of course, the model checker. The model checker was used to check that the Erlang program guaranteed: mutual exclusion of the resources that were requested exclusively, validity of all kind of assertions in the code, possibility of sharing a resource if requested shared access, priority of exclusive access over shared access, *etc.*

**Results** After the development of the translator from Erlang to $\mu$CRL had been finished (at least for the part of Erlang that we used in our examples), model checking became almost push-button technology. The technique is limited to finite transition systems, but, luckily, we were dealing with these in our case study. The program itself contains unbounded data structures, like lists, but when the number of clients is fixed, the lists have an natural upperbound. Clearly, we could only verify the properties for a fixed number of clients, where we used the supervision tree to generate the right configuration automatically. Checking properties in this way for several configurations, from one up to five clients, turned out to be relatively easy. The main problems that are still open deal with the formulation of the properties:

[1] The code for this verified program is available in an extended version of this paper.

– it is in general hard to ensure that one specifies a requirement in the right way,
– some properties (such as *no starvation*) cannot satisfactory be specified in the given logic, and
– for every configuration, a slightly different property needs to be specified (thus, properties are far from re-usable).

Other problems are of a more technical matter, such as dealing with transition systems of around one gigabyte, speeding up the generation of the transition system and the model checking (the latter for example by using a parallel model checker [12]).

We have been able to verify a combination of four Erlang modules, together containing about 200 lines of advanced code, containing pattern matching, library functions, higher-order functions, and several data structures (lists, records, *etc.*). The $\mu$CRL specifications we obtained from this code were about 1000 lines long. This inconvenient expansion is mainly caused by the fact that the untyped Erlang terms are all translated in the same $\mu$CRL sort, which then has to be supplied with a very large definition of equality. The datatype primitives, therefore, are responsible for this large size of the $\mu$CRL specification. Obtaining the transition system from $\mu$CRL takes a few minutes up to half a day. Model checking is in general faster than creating the transition system, even though it could take several hours for a transition system of a few hundred megabytes.

## 5 Proving correctness of critical code

There are four reasons not to be satisfied with only a model checker tool for Erlang programs. First of all, one wants to reason about an unbounded number of processes, about dynamically generated processes and about data structures that have no natural maximum size. Second, the logic for model checking is normally restricted and there are properties that one could express in a more powerful logic, but not in the logic supported by a model checker. Third, we use model checking as a technique for finding errors, not for proving correctness. For real safety critical parts of the software, you might want to be more demanding and prove the code to be correct w.r.t. a certain property. Fourth, for real systems, one expects not to be able to use model checking, because of a state space that grows simply too large. A theorem prover that supports compositional reasoning could then be a powerful tool to split the problem in sub-problems that can on their turn be checked by the same theorem prover or a model checker.

For the above mentioned reasons, a theorem prover for Erlang has been developed; a classical proof assistant requiring users to intervene in the proof processes at critical steps, such as stating program invariants. The tool offers considerable support for automatic proof discovery through higher-level tactics tailored to the particular task of the verification of Erlang programs. In addition, a graphical interface permits easy navigation through proof tableaux, proof reuse, and meaningful feedback. Rather than working with some abstract model of the Erlang system, this verification approach is directly based on the code: we show that a concrete Erlang program satisfies a set of properties, formalized in the $\mu$-calculus. The (full) $\mu$-calculus semantically subsumes the alternation free $\mu$-calculus, CTL, CTL* and LTL. The theorem prover supports reasoning which is parametric on components and supports inductive and co-inductive reasoning about recursively defined components and properties [19].

**Case studies** The theorem prover for Erlang has been developed over several years and we have worked with many small and a few larger case studies, including a mobile billing agent [14], a safety critical database lookup manager [4], sorting algorithms [22], persistent sets [18], a concurrent server that creates new processes to handle incoming tasks [19], and currently still ongoing, the same access manager program as described in the previous section.

For the properties we have verified we often needed the power given by the full $\mu$-calculus, not being able to express it in alternation free $\mu$-calculus or another less powerful logic.

**Results** For safety critical parts of software that are reasonably stable, i.e. are no longer subject to change, it can be beneficial to use the theorem prover approach. The tasks of proving a property of advanced code is rather labour intensive and takes months. The constructed proofs can easily contain several thousands of intermediate goals, of which a large part is solved automatically. However, the level of automation is insufficient for leaving other than experts use the tool.

Without too much expertise, the theorem prover approach can also be used for advanced debugging of Erlang code. One can try to prove a property that basically requires a symbolic evaluation of an Erlang program. Working on such a proof gives insight in the execution behavior of the program, not limited to fixed initial parameters. Moreover, one might learn about errors in the program by not being able to proof the property, deriving an unfinished proof tree reflecting a symbolic trace [9].

The possibility to decompose proofs in smaller tasks has not been fully explored yet and we believe to get better results in the future. Our hope is that we can combine model checking and theorem proving, having the user roughly sketching the proof and use a model checker for sub-goals that deal with finite state spaces. An indirect way of decomposing proofs by just assuming properties of certain components has been implemented for generic servers [7]. This results in smaller proofs, but some more work on the automation of this technique is needed to make it practically applicable.

## 6 Conclusions

The way Erlang programs are developed at the moment is rather successful. More support for finding errors in the code is, though, a frequently demanded issue. Whatever method or tool is developed to help in this, it should not interfere too much with the established way of working.

**Software Development** We aim on extending the test server that is presently used during product development with formal verification features. Designers use the test server to run overnight tests on their code. In addition to compiling and testing the code, a few properties will automatically be verified on this server. The properties are planned to be manually constructed from the requirement specifications, which only needs to be done once in the beginning of the project (cf. [25]). While the code is constantly changing, the same properties are checked over and over again. Encountered errors are presented to the programmers by means of an Erlang trace leading to the error state. Model checking in combination with our translation tool can support this.

For the most critical parts in the software, for example library functions that are added to the Erlang distribution, we plan to use the theorem prover to make sure that certain properties hold for these parts. These properties then also serve as a very precise documentation of the behavior of the parts.

**Code optimizations** The translation from Erlang to $\mu$CRL and generation of labeled transition systems is, apart from debugging, also very useful for checking equivalence of two implementations. Quite often a program is optimized after that a first, working version has been produced. It is possible to use bisimulation techniques on labeled transition systems to check whether two implementations behave the same in a certain environment. After defining which equivalence one is trying to prove, this technique can be used to show correctness of an optimization [6].

## References


1. J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall International, 2nd edition, 1996.
2. T. Aronsson and J. Grafström, *A Comparison between Erlang and C++ for Implementation of Telecom Applications*, master thesis, University of Linköping (LiTH/IDA), Sweden, 1995.
3. T. Arts and C. Benac Earle, Development of a Verified Erlang Program for Resource Locking, In *Proc. of Formal Methods in Industrial Critical Systems (FMICS2001)*, Paris, France, July 2001.
4. T. Arts and M. Dam, Verifying a Distributed Database Lookup Manager Written in Erlang, In *Proc. of the world congress on Formal Methods (FM'99)*, LNCS 1708, p. 682-700, Springer-Verlag, Berlin, 1999
5. T. Arts, M. Dam, L-å. Fredlund, and D. Gurov, System Description: Verification of Distributed Erlang Programs. In *Proc. 15th Int. Conf. on Automated Deduction*, LNAI 1421, p. 38–42, Springer Verlag, Berlin, 1998.
6. T. Arts and I.A. van Langevelde, Correct Performance of Transaction Capabilities, In *Proc. of the 2nd Int. Conf. on Application of Concurrency to System Design*, Newcastle upon Tyne, UK, June 2001.
7. T. Arts and T. Noll, Verifying Generic Erlang Client-Server Implementations. In *Proc. 12th Int. Workshop on the Implementation of Functional Languages (IFL2000)*, LNCS 2011, p. 37–53, Springer Verlag, Berlin, 2000.

8. K. Beck, *eXtreme Programming explained*, Addison-Wesley Pub Co, 1999.
9. C. Benac Earle, Symbolic Program Execution using the Erlang Verification Tool, In *Proc. of the 9th Int. Workshop on Functional and Logic Programming (WFLP'2000)*, Benicassim, Spain, September 2000.
10. S. Blau and J. Rooth, AXD 301 – A new Generation ATM Switching System. *Ericsson Review*, no 1, 1998.
11. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol, $\mu$CRL: A Toolset to Analyse Algebraic Specifications, In *Proc. of the 13th Int. Conf. on Computer Aided Verification*, LNCS 2102, p. 250–254, Springer Verlag, Berlin, 2001.
12. B. Bollig, M. Leucker, and M. Weber, Local Parallel Model Checking for the Alternation Free $\mu$–Calculus. tech. rep. AIB-04-2001, RWTH Aachen, March 2001.
13. J. Corbett, M. Dwyer, L. Hatcliff, Bandera: A Source-level Interface for Model Checking Java Programs. In *Teaching and Research Demos at ICSE'00*, Limerick, Ireland, 4-11 June, 2000.
14. M. Dam, L.-Å. Fredlund, and D. Gurov, Toward Parametric Verification of Open Distributed Systems, In *Compositionality: The Significant Difference*, LNCS 1536, p. 150-185, Springer-Verlag, Berlin, 1998
15. B. Däcker, *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction*, Licenciate Thesis, Department of Teleinformatics Royal Institute of Technology, Stockholm, Sweden, November, 2000.
16. Open Source Erlang, `http://www.erlang.org/`, 1999.
17. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireau. CADP (CÆSAR/ALDÉBARAN development package): A protocol validation and verification toolbox. In *Proc. of the 8th Int. Conf. on Computer Aided Verification*, LNCS 1102, p. 437–440, Springer Verlag, Berlin, 1996.
18. L.-Å. Fredlund, D. Gurov, A Framework for Formal Reasoning about Open Distributed Systems, In *Proc. of ASIAN'99*, LNCS 1742, p. 87-100, Springer Verlag, Berlin, 1999.
19. L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov, A Verification Tool for Erlang, Accepted for publication in: *International Journal of Software Tools for Technology Transfer*, 2001.
20. W. Fokkink, *Introduction to Process Algebra*, Texts in Theoretical Computer Science, Springer Verlag, Heidelberg, 2000.
21. J. F. Groote, The syntax and semantics of timed $\mu$CRL. tech. rep. SEN-R9709, CWI, June 1997. Available from `http://www.cwi.nl/`.
22. D. Gurov and G. Chugunov, Verification of Erlang Programs: Factoring out the Side-effect-free Fragment, In *Proc. of Formal Methods in Industrial Critical Systems 2000*, GMD Report, no. 91, p. 109-122, 2000.
23. K. Havelund and T. Pressburger, Model checking JAVA programs using JAVA PathFinder. *Int. J. on Software Tools for Technology Transfer*, Vol 2, Nr 4, p. 366–381, March 2000.
24. G. Holzmann, *The Design and Validation of Computer Protocols.* Edgewood Cliffs, MA: Pretence Hall, 1991.
25. G. Holzmann, From Code to Models, In *Proc. of the 2nd Int. Conf. on Application of Concurrency to System Design*, Newcastle upon Tyne, UK, June 2001.
26. F. Huch, Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proc. of the International Conference on Functional Programming*, Sept. 1999.

27. F. Huch, Model Checking Erlang Programs – Abstracting the Context-Free Struckture. In *Proc. of the 10th Int. Workshop on Functional and Logic Programming (WFLP'2001)*, Kiel, Germany, September 2001.
28. S. Marlow and Ph. Wadler, A Practical Subtyping System For Erlang, In *Proc. of the Int. Conf. on Functional Programming*, Amsterdam, The Netherlands, 1997.
29. T. Noll, L.Å. Fredlund, and D. Gurov, The Erlang Verification Tool, In *Proc. of TACAS'01*, LNCS 2031, p. 582-585, Springer-Verlag, Berlin, 2001
30. Open Telecom Platform, `http://www.erlang.org/`.
31. U. Wiger, Four-fold Increase in Productivity and Quality; Industrial-Strength Functional Programming in Telecom-Class Products, In *Proc. of Workshop on Formal Design of Safety Critical Embedded Systems (FEmSYS2001)*, Munich, Germany, March 2001.