Applicability of the BLAST Model Checker: An Industrial Case Study*

Emanuel Kolb[1], Ondřej Šerý[2], Roland Weiss[1]

[1] Industrial Software Systems, ABB Corporate Research,
ABB AG, Forschungszentrum Deutschland,
Wallstadter Str. 59, D-68526 Ladenburg, Germany
{emanuel.kolb, roland.weiss}@de.abb.com

[2] Charles University in Prague
Malostranske nam. 25, 118 00 Prague 1, Czech Republic
ondrej.sery@dsrg.mff.cuni.cz

Abstract. Model checking of software has been a very active research topic recently. As a result, a number of software model checkers have been developed for analysis of software written in different programming languages, e.g., SLAM, BLAST, and Java PathFinder. Applicability of these tools in the industrial development process, however, is yet to be shown. In this paper, we present results of an experiment, in which we applied BLAST, a state-of-the-art model checker for C programs, in industrial settings. An industrial strength C implementation of a protocol stack has been verified against a set of formalized properties. We have identified real bugs in the code and we have also reached the limits of the tool. This experience report provides valuable guidance for developers of code analysis tools as well as for general software developers, who need to decide whether this kind of technique is ready for application and suitable for their particular goals.

Keywords: Software analysis, Model checking.

1 Introduction

Is model checking of software ready for application in the industrial development process? The answer to this question is not simple. In some situations, where short time to market and initial budget is more important than correctness, the benefits in the sense of software quality might never outweigh the additional costs of model checking both in time and money. On the other hand, in some very specific scenarios, software model checking is already being used for quite a time, e.g., during development of Windows device drivers [1]. Nevertheless, it is yet to be shown that software model checking can be applied also outside these specific domains.

* This work was funded in the context of the Q-ImPrESS research project (http://www.q-impress.eu) by the European Union under the ICT priority of the 7th Research Framework Programme and partially supported by the Grant Agency of the Czech Republic project 201/08/0266.

There are potential industrial users. Unfortunately, they often miss the information on which to base their decision of using this technique. Unlike in other fields (e.g., theorem proving and HW model checking), there is no widely accepted collection of problems for software model checking that would allow for comparison of model checking tools against each other and also assessing the strength of the technique as a whole. Therefore, a potential industrial user has to make his/her mind based on few available case studies. Moreover, many of these are conducted by the tool's authors in order to point out benefits of a particular technique they use, which gives little information on general applicability. Often, the tools are used to find already discovered errors (e.g., in [14]). This is definitely useful to show that finding a particular type of error is at least possible. However, when manual code simplifications are necessary to do so, it is unclear whether those might have been applied even without knowing the error in advance.

We believe that both potential industrial users and authors of the model checking tools would benefit from more case studies showing applicability and emphasizing limitations of the tools, which are currently available.

1.2 Goal and Structure of the Paper

In this paper, we document a case study in which we have employed the state-of-the-art software model checker BLAST in analysis of an industrial strength C implementation of the OPC UA [13] protocol stack. We have discovered a number of real defects with a reasonable rate of false positives. Our original input was the C source code and a set of informally stated properties to be verified.

The rest of the paper is structured as follows. The OPC UA protocol stack and its C implementation are briefly presented in Section 2. The BLAST model checker is described in Section 3, where we also advocate its choice. In Section 4, the experiment and its results are discussed. In Sections 5 and 6, the related work is listed followed by our conclusions.

2 Case Study: OPC UA Protocol Stack

The OPC Unified Architecture (OPC UA) is a platform-independent standard through which various kinds of systems and devices can communicate by sending messages between clients and servers over various types of networks. OPC UA is applicable to manufacturing software in application areas such as Field Devices, Control Systems, Manufacturing Execution Systems and Enterprise Resource Planning Systems. These systems are intended to exchange information and to use command and control for industrial processes.

It is expected that, over the next years, OPC UA will replace the "classic" OPC protocols, like OPC DA (Data Access), OPC A&E (Alarm and Event) and OPC HDA (Historical Data Access). The classic OPC protocols are widely used in industrial automation, but – due to their specification based on Microsoft's COM and DCOM technology – they can only run on Windows computers. Since OPC UA is based on

the web-service paradigm, the protocol is able to run also on non-Windows systems, like Linux or VxWorks. The OPC UA specification is expected to be released by the OPC Foundation [17] in the 1st quarter of 2009.

The specification of the OPC UA protocol is not based on a specific programming language or technology. To access the OPC UA protocol from a specific programming language, a binding of the protocol to the language must be provided. This allows OPC UA applications written in different languages to communicate with each other. Although the OPC Foundation does not specify bindings for programming languages, it makes binding implementations (so called OPC UA Stacks) for C/C++, Java and .Net available. An OPC UA Stack implements the serialization, security and transport of messages exchanged between different UA Applications.

A typical architecture of an OPC UA communication system is depicted on Figure 1. There are two main roles for OPC UA applications: OPC UA client and OPC UA server. An OPC UA client builds up a connection to an OPC UA server and accesses/manages the data that is made available by the server.

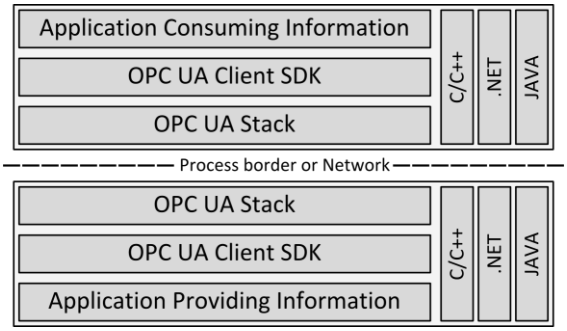

Figure 1. A typical architecture of an OPC UA communication system

In this paper, we are particularly interested the OPC UA C-stack, which implements the OPC UA protocol binding for the C programming language. The OPC UA C-Stack is programmed in ANSI C (about 150 KLOC) and is split into a platform independent and a platform dependent part (also called platform layer). The platform layer contains the platform specific code that is needed for porting the C-Stack to a specific Operating System. Currently platform layers for Windows and Linux are available. The component is designed for usage in both PC-based and embedded systems.

The internal structure of the OPC UA C-Stack is visualized in Figure 2. Responsibilities of the particular modules are as follows:

- **Server Stub** provides the OPC UA API for OPC UA server applications. Its main functions are: “Managing communication endpoints”, “Entry points for OPC UA services” and “Service infrastructure functions”.
- **Client Proxy** provides the OPC UA API for OPC UA client applications. Its main functions are: “Managing connections” and “Calling OPC UA Services in synchronous or asynchronous mode”.

- **Secure Channel** manages the security layer of the OPC UA protocol. The security layer is on top of the transport layer "Tcp Transport". Its main functions are: "Managing secure connections on client and server side", "Managing the secure data stream" and "Managing security policies".
- **Tcp Transport** is responsible for the binary TCP channel of the OPC UA protocol. Its main functions are: "Managing TCP connections" and "Managing the TCP data stream".
- **Stack Core** contains the core functionality of the C-stack. Its main functions are: "Providing Binary encoders and decoders", "Providing the OPC UA built-in types", "Providing basic cryptographic functions", "Basic stream and connection handling", "Providing the message types used in the OPC UA services" and "Providing a string table type".
- **Core** contains the UA protocol independent basic functionality. Its main functions are: "Basic type handling (Guid, DateTime, Buffer, List, String)", "Basic proxystub handling", "Basic memory functions", "Thread and Threadpool management", "Timer functions", "Tracing functions and "Some utility functions (bsearch, qsort,...)".
- **Platform** contains platform specific submodules (Linux, Windows, ...). A platform specific submodule interfaces and abstracts the OS dependent system calls. Its main functions are: "Thread handling", "Mutex/Semaphore functions", "Timer implementations", "Date and Time handling", "Socket handling", "String handling" and "Security functions (interfacing OpenSSL)".

After briefly presenting the BLAST model checker in the next section, we will document its application in analysis of the OPC UA C-Stack source codes.

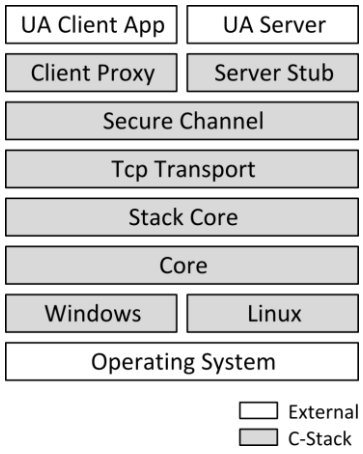

Figure 2. Internal structure of the OPC UA C-Stack

3 Overview of BLAST

As already mentioned, BLAST is a model checker for analyzing programs written in the C programming language. As well as some other code model checkers designed for C, SLAM [1] and SATABS [7], BLAST utilizes *predicate abstraction* [2] and iterative refinement of the abstraction based on spurious counter-examples. In literature, this technique is often referred to as *counter-example guided abstraction refinement* (CEGAR).

In a nutshell, a coarse existential abstraction (over-approximation) is initially created from the program under analysis. This abstraction is then traversed and sought for any reachable error state. If no reachable error state is found, the use of over-approximation grants that also the original program is error-free. On the other hand, if there is a reachable error state, then this may be either a real error of the original program or a spurious error due to the abstraction. In the second case, the abstraction is refined based on the spurious error in order to avoid it in the future. After such a refinement, the new abstraction is traversed and the process iterates.

Unlike other tools based on CEGAR, BLAST features *lazy abstraction* [11]. This means that BLAST creates the abstraction on-the-fly, and in the refinement step, it refines only the necessary portions of the abstraction, while keeping the rest. Thus, only the reachable part of the abstraction is created and the portions that were previously proven to be error-free are not refined again. This is in contrast to the naïve CEGAR implementation which recreates the whole abstraction in every iteration.

Another useful feature of BLAST is *configurable program analysis* (CPA), published by the BLAST's authors in [5]. The CPA concept stems from *abstract interpretation* [9] and was originally introduced to provide a uniform view on model checking and static analysis. The basic idea is to have multiple CPAs for tracking different kinds of information (e.g., predicates and heap shape) about the program under analysis. Each CPA tracks the information in either a path sensitive or insensitive way. By combining the different CPAs, various configurations of the resulting analysis can be achieved. In [6], this idea is extended by the notion of dynamically adjustable precision of the information that is tracked, yielding *configurable program analysis with dynamic precision adjustment* (CPA+).

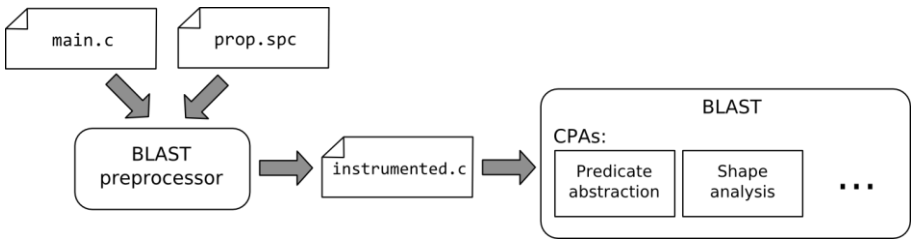

Figure 3. Original BLAST architecture with property specification preprocessing and configurable program analysis

Building on this work of others, we have proposed an improvement of property specification in CEGAR based tools and implemented an extension in BLAST. Tools

like SLAM and BLAST allow for specification of temporal safety properties in a specialized formalism SLIC [3] and BLAST specification language [4], respectively. As depicted on Figure 3, the property specified in such formalism is then used to instrument the original source code of the program under analysis. Artificial variables, their updates, and assertions are added to the source code. The instrumented source code is then analyzed for reachability of error states. This way, the property is encoded into the source code and manifests itself as additional predicates that have to be tracked during the state space traversal, requiring additional theorem proving overhead. Moreover, the instrumentation step has to be repeated after any modification of the original code.

In [16], we proposed an enhancement based on tracking the state of the property explicitly side by side with the program's abstraction, thus, overcoming the need for prior source code instrumentation. In the prototype implementation of the BLAST extension, the property is encoded and tracked in a separate Behavior CPA. The idea is depicted in Figure 4.

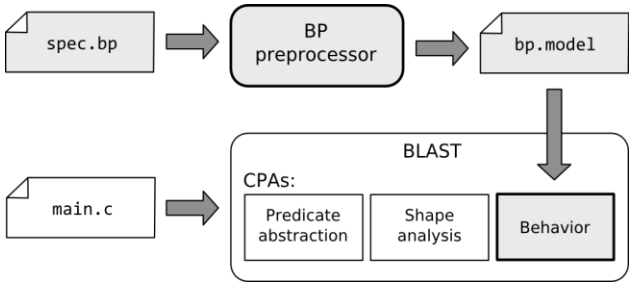

Figure 4. Extended BLAST architecture when property specification is tracked in a specialized CPA (Behavior CPA)

In the following, we document application of the BLAST model checker enhanced with the abovementioned property specification extension to the C implementation of OPC UA Stack. We have chosen BLAST for this purpose, because, it uses very advanced techniques in comparison to the other CEGAR based tools and it is generally regarded as a state-of-the-art tool in this category. Another reason was our previous positive experience with the tool.

4 Experiment

An important aspect of the OPC UA C-Stack source code we have analyzed is that it is basically a library. In contrast to standalone software, when analyzing a library, there is always the problem of a missing environment. In other words, one cannot simply feed the source code into a model checker tool and hope for meaningful results, because the source code of a library constitutes only a partial model, e.g., there is no main function and the behavior of user code is missing. Moreover, a library is in general used in different environments and analyzing it in a specific one fails to provide guarantees for the others.

There are basically two ways to mitigate this issue. First, one can use the *assume-guarantee principle* [12] and try to formulate the library's assumption about its environment by creating a very general testing harness, i.e., the *most general environment*. Such an environment should cover both typical usage patterns and border cases. A consecutive analysis of the library in the most general environment would then provide guarantees for any environment subsumed by the most general one (i.e., any environment satisfying the library's assumption).

The second way to cope with the missing environment is to use defensive programming and make no assumptions on the order and the context in which the library's API functions are used, while requiring the library to stay error free and internally consistent at all times. Although the second option gives stronger guarantees and provides for analysis of the API functions in separation, it is also prone to reporting many false positives (due to ignoring potential assumptions), unless the library's source code is written defensively itself.

Although we have originally planned to try both ways, we were unable to complete the first one due to several BLAST limitations. We have, however, achieved satisfactory results in applying the second way with a relatively low ratio of false positives. This is mainly due to the fact that the OPC UA C-Stack code is written very defensively with little implicit assumptions. In the following, this second attempt is described in detail, the discussion of the BLAST's limitations is postponed to Section 4.4.

4.1 Methodology

In our experiments, we have identified three properties (described in Section 4.2) that should be satisfied during calls to the OPC UA C-Stack API functions. Each property was specified using a simple regular language [16]. For each property and each relevant OPC UA C-Stack API function, we have executed the BLAST model checker with the Behavior CPA extension. Although specific functions were used as entry points, BLAST traverses all the states reachable from the specific function in an inter-procedural manner (i.e., also functions transitionally reachable from the API functions are considered).

Note also, that some code changes had to be introduced to make the analysis by BLAST possible. Main source of these changes was use of bit operations in error handling code. Although, in principle, BLAST can analyze source code featuring bit operations, it cannot reason about them properly when reachability of an error state depends directly on the result of a bit operation. Note that OPC UA C-Stack was not developed with model checking in mind. Therefore use of some bit operations in the error handling code was superfluous and was rewritten by other means to facilitate analysis. Another change was rewriting of some C macros into functions, so that they survived code preprocessing and BLAST could reason about them. This was the case with mutex locking/unlocking functions, which were originally implemented as macros. After code preprocessing, it was hard to formulate properties concerning proper ordering of mutex locking and unlocking.

It is worth to mention that majority of the necessary changes have been made in header files. In our settings, this resulted in two versions of header files, the original

ones and the "model checking friendly" headers. However, given the rather small amount of changes (tens of lines), this duality would not be strictly necessary if the development was started with model checking and its limitations in mind.

4.2 Properties

Initially, a domain expert identified about eight informal properties to be analyzed. From these, based on experience with model checking tools, we have picked the following three amenable to analysis using BLAST.

Locking policy. The OPC UA C-Stack is multithreaded by design. Where necessary, access to shared data structures is controlled by locking and unlocking of mutexes. A natural question here concerns correctness of such a locking policy. For example, it is important to see, that any locked lock is always unlocked before returning from an API function call, no matter how exceptional situation might occur.

MessageContext management. Whenever the OPC UA C-Stack implementation manipulates a network message, it uses a message context to hold all necessary data. Before using the message context, the `OpcUa_MessageContext_Initialize` function has to be called first to initialize it. The message context has to be cleared for other use by invoking the `OpcUa_MessageContext_Clear` function afterwards.

Encoder management. The OPC UA C-Stack supports secure connection by encoding messages using SSL. This task is carried out by an encoder. Similarly to the previous property, the encoder has to be opened by invoking `OpcUa_Encoder_Open` before attaching it to a stream. When the encoding is done, and the encoder has to be closed by calling the `OpcUa_Encoder_Close` function in order to free the associated resources and to prevent leaks.

For example, one omitted property is to check that all pointer parameters of an API function are checked for `null` before use. This task is more suitable for static analysis than for model checking. Another one is to check that all allocated memory is eventually freed, which we consider beyond the power of the current model checkers.

4.3 Experiment Results

All the tests were run on a Linux 2.6.27 system with Intel Pentium 4 CPU at 3.00GHz featuring 2GB of memory[1]. Tables 1-3 summarize running times and number of defects found in the individual tested files for each of the three above listed properties. The error traces reported by BLAST (*reported errors*) were manually inspected to identify real defects (*real errors*). In few cases, the tool was unable to perform the analysis (*tool failures*). This was mainly due to recursive functions,

[1] Although the system supports hyper-threading, BLAST and its process subtree was executed on a single virtual core. This is due to a synchronization issue present in BLAST at the time of writing, which manifested itself on multiprocessors resulting in random deadlocks.

which are not supported by the tools combination we have used. Of course, such cases have to be considered as potentially erroneous.

A typical spurious error is a situation, where the locking policy is rightfully violated by design. For example, the `TcpSecureChannel_GetSecuritySet` locks a mutex which is unlocked by the `TcpSecureChannel_ReleaseSecuritySet` function. In the functions, a missing unlock and a missing lock are reported, respectively. Second type of a spurious error is a situation, where multiple mutexes are manipulated in a single function and they are mismatched by the tool. This is due to the fact that the properties are specified as a correct ordering of function calls, ignoring the function parameters (which identify the distinct mutexes in the code).

Table 1: Analysis results of the locking policy property

filename	*time [s]*	*reported errors*	*tool failures*	*real errors*
opcua_proxystub.c	2.3	0	0	0
opcua_threadpool.c	2.0	1	0	1
opcua_trace.c	1.2	0	0	0
opcua_thread.c	1.7	0	0	0
opcua_endpoint_ex.c	1.1	0	0	0
opcua_endpoint.c	10.1	0	0	0
opcua_asynccallstate.c	1.1	0	0	0
opcua_channel.c	22.8	4	0	3
opcua_tcpsecurechannel.c	6.3	5	0	2
opcua_securelistener.c	48.2	1	0	0
opcua_secureconnection.c	3:10.4	9	0	1
opcua_binarydecoder.c	1:24.3	0	6	0
opcua_binaryencoder.c	1:54.5	1	8	1
opcua_tcplistener.c	11.1	1	0	0
opcua_tcpconnection.c	7.8	0	0	0

Table 2: Analysis results of the MessageContext management property

filename	*time [s]*	*reported errors*	*tool failures*	*real errors*
opcua_endpoint.c	7.7	0	0	0
opcua_channel.c	14.6	2	0	2
opcua_securelistener.c	16.1	0	0	0
opcua_secureconnection.c	2:15.8	4	0	2

Table 3: Analysis results of the Encoder management property

filename	*time [s]*	*reported errors*	*tool failures*	*real errors*
opcua_endpoint.c	7.5	0	0	0
opcua_channel.c	7.1	0	0	0
opcua_securelistener.c	16.1	0	0	0
opcua_secureconnection.c	1:41.0	1	0	1

Let us also describe a typical representative of the real defects that were found. In the OPC UA C-Stack, the API function bodies are typically separated into a business part, which performs the desired activity, and an error handling part, which takes care

of exceptional situations and performs the necessary cleanup. Two macros are used to facilitate error handling. The `OpcUa_GotoErrorIfBad(uStatus)` macro jumps into the error handling code if the preceding activity have failed. In the same situation, the `OpcUa_ReturnErrorIfBad(uStatus)` macro immediately returns from the function without performing the cleanup. Misuse of the latter one while holding a lock violates the locking policy property and may easily lead to a deadlock.

4.4 Discussion

Above, we have described our findings and it should be clear that, BLAST can be used in the industrial development process to discover real errors that were missed by previous conventional testing. On the other hand, it is also necessary to see what guarantees are really provided by this type of analysis. Or put differently, we should be aware of what we might have missed.

Naturally, BLAST would verify only those properties that were a priory identified by a human user. As a trivial consequence, one should never mistakenly interpret satisfaction of all predefined properties as overall program correctness. There is always a risk that the set of properties is not exhaustive for a particular task.

Even when an exhaustive set of properties is chosen, some of them might be hard or impossible to specify and verify. Let us consider the locking policy property again. A user might want to verify that a specific lock is always locked before a particular data structure is accessed. Unfortunately, BLAST offers no means for this kind of property to be specified. It would require a specification language with deeper understanding to semantics of data structures than BLAST is able of. Also related to the locking policy is a question of deadlock freedom. Although, the verification approach employed by BLAST, might be used for multithreaded programs in principle, the tool itself has no support for multithreading. As a result, we were able to verify correct locking policy of a single thread (e.g., no pending locks) and even identify some real violations, but there is still a chance that a deadlock may occur (e.g., due to mutexes acquired in different order by distinct threads).

Another issue we have encountered is limitation regarding pointers. First, the OPC UA C-Stack uses function-pointers (unsupported by BLAST) quite heavily. In some situations, this might be overcome by explicit function calls. In others, it cannot. Also quite often, functions accept out-parameters as pointers to values to be modified. Under such circumstances, BLAST often fails to reason about properties directly depending on the modified value. This is the main reason why we were unable to employ an environment that would call the OPC UA API functions in a specific order (as described above in Sect. 4). The API functions depend on out-parameters passed as pointers and modified inside other functions. BLAST failed to analyze this scenario and we have found no workaround, thus, having to abandon the approach entirely.

As a last comment, BLAST would greatly benefit from support during the entire development process. There are typically some changes necessary to the code base in order to make it model checking friendly (e.g., providing dummy versions of some 3rd party libraries, etc.). With no tool support, managing the two versions is an extra burden for the developers. Interpreting the checking results with no graphical support

is also very tedious. Unfortunately, there are quite a lot of such tiny bits which, in real development, weight against use of this type of formal analysis.

5 Related Work

As already described in Section 3, BLAST is only one of the C model checkers based on CEGAR. Related tools include SLAM [1] and SATABS [7]. A bit different model checking tool, CBMC [15], is based on bounded model checking. This means that it does not exhaustively traverse the whole state space of a program, but rather limits the search in depth (e.g., by a number of executed code blocks or by a number of context switches).

Another direction of related work is application of the model checking tools on case studies. Typically, authors present their tool on examples more or less chosen to manifest its strong points. Quite often, the tools are run on a code with a previously known bug. The source code under analysis is typically manually simplified to make the analysis feasible. Unfortunately, it is hard to see to whether the simplification is driven by some kind of generally applicable guidelines or be the goal of finding the specific bug.

In [14], Muhlberg et al. used BLAST to analyze portions of Linux kernel code for previously reported bugs. Although the bugs were known in advance, substantial manual code changes were introduced in order to make them detectable. The authors conclude that BLAST has several limitations mainly concerning documentation, pointer support and general usability, on which we agree. In contrast to this experiment, our input was source code and informal list of properties rather than a list of previously known defects. In this respect, we have a more positive experience with identification of previously unknown bugs. However, it would be too optimistic to claim that our verification effort provides strong guarantees regarding the program correctness (as discussed in Section 4.4).

Other techniques than model checking (e.g., static and runtime analysis) are often used for error detection. Nice summaries can be found in [14, 10]. Although these techniques are typically easier to use and they scale much better than model checking, the correctness guarantees are weaker.

6 Conclusion

We have presented an experiment comprising application of the state-of-the-art C code model checker BLAST to an industrial case study. We regard the experiment as successful. We have discovered a number of real issues of the OPC UA C-Stack with a reasonable number of false positives. On the other hand, we have met several limitations that make adoption of these tools in the industrial development process difficult. Some of them follow from the fact that the tools are still research prototypes and might be overcome by additional tool support facilitating day-to-day use. Other ones (e.g., better pointer and multithreading support) are still a hot topic of the current research.

References

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. *SIGPLAN Not.*, 36(5):203–213, 2001.
3. T. Ball and S. K. Rajamani. Slic: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.
4. D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS: Static Analysis*, Lecture Notes in Computer Science 3148, pages 2–18. Springer, 2004.
5. D. Beyer, T. A. Henzinger, and G. Theoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 504–518, Springer, 2007.
6. D. Beyer, T. A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, September 15-19)*. IEEE Computer Society Press, 2008.
7. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of Lecture Notes in Computer Science, pages 570–574. Springer Verlag, 2005.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
10. D. Engler, M. Musuvathi. Static analysis versus software model checking for bug finding. In Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS 2937, pp. 405–427, 2004.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.
12. D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh. Assume-guarantee Verification of Source Code with Design-Level Assumptions, In *Proceedings of the 26th International Conference on Software Engineering*, IEEE, 2004.
13. W. Mahnke, S.-H. Leitner, M. Damm. *OPC Unified Architecture*. Springer, February 2009.
14. J. T. Muhlberg, G. Luttgen. BLASTing Linux Code. In Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 06), LNCS 4346, pp. 211–226, 2007.
15. I. Rabinovitz, O. Grumberg. Bounded Model Checking of Concurrent Programs. In Proceedings of Computer Aided Verification (CAV'05), LNCS 3576, 2005.
16. O. Sery. Enhanced Property Specification and Verification in BLAST. Accepted for publication in *Proceedings of Fundamental Approaches to Software Engineering (FASE'09)*, Lecture Notes in Computer Science, UK, 2009.
17. OPC UA Foundation, www.opcfoundation.org