Deductive Verification of System Software in the Verisoft XT Project

Bernhard Beckert, Michał Moskal

The main goal of the Verisoft XT project is the creation of methods and tools which allow for the pervasive formal verification of integrated computer systems, and the prototypical realization of four concrete industrial application tasks.
In this paper, we report on two of Verisoft XT's sub-projects, where formal verification is applied to real-world system software, namely Microsoft's Hypervisor and the embedded operating system PikeOS. We describe the deductive verification technology used in Verisoft XT and the tool chain that implements these methods, including the C verifier called VCC and the SMT solver Z3.

1 Introduction

In recent years, deductive program verification has improved to a degree that makes it feasible for real-world programs. Following this observation, the main goal of the Verisoft XT project is (a) the creation of methods and tools which allow the pervasive formal verification of integrated computer systems, and (b) the prototypical realization of four concrete industrial application tasks. Verisoft XT is funded by the German Federal Ministry of Education and Research (BMBF).

As correctness of the built-in operating system is a crucial requirement for the reliability of safety- and security-critical systems, two of the four Verisoft XT application tasks are operating system verifications: the goal of the *Avionics* sub-project is to prove functional correctness of the microkernel in the hypervisor PikeOS, a commercial operating system for embedded systems [3, 4]; the *Hypervisor* verification sub-project aims at full functional verification of the kernel of Hyper-V, an industrial virtualization platform, currently shipped with Microsoft Windows Server 2008.

Modern operating systems are usually implemented in C, with small, but significant, parts in assembly code. The C code is highly optimized, does not adhere to any simple type discipline, is concurrent, and implements a wide range of custom concurrency control primitives. These characteristics are shared by both our verification targets and place very strict requirements on the verification methodology to be applied.

Both projects decided to use deductive verification as the underlying verification technology. To verify that (a part of) a program performs according to its specification, a logical formula is automatically generated from the source P of the program and its specification S. This formula ϕ, called *verification condition*, is rendered in predicate logic and has the property that, if it is valid, then P is correct w.r.t. S. Finding a proof for the validity of ϕ, which would serve as a witness for the correctness of P, is then a task to be solved by a deduction system. More information on deductive program verification and current developments in this area may be found in the related article [1] in this issue.

Because Verisoft XT aims at industrially viable methods, it is highly desirable for the deduction system to be automatic and for the specification to be reasonably easy to develop.

PikeOS (see `http://www.pikeos.com/`) consists of a microkernel acting as paravirtualizing hypervisor (i.e., it presents the guest systems with an interface similar but not identical to that of the underlying hardware) and a system software component. The PikeOS *kernel* is particularly tailored to the context of embedded systems, featuring real-time functionality and orthogonal partitioning of resources such as processor time, user address space memory, and kernel resources. The PikeOS *system software* component is responsible for system configuration. The allocation of resources can be bound at compile-time, for example to conform to partitioning requirements. At the kernel level, the mechanisms for communication between threads are IPC, events, and shared memory. High-level communication concepts can be mapped onto these kernel-level mechanisms. For a thorough discussion of PikeOS and its evolution, see [11].

The **Hypervisor** verification project aims at full functional verification of the kernel of Hyper-V, an industrial virtualization platform, currently shipped with Microsoft Windows Server 2008. Hyper-V allows for running several operating systems on the same physical machine. It is essentially a small operating system, with memory management, scheduler, and essential device drivers. It consists of about $100\,000$ lines of C code (excluding comments) and about $5\,000$ lines of x64 assembly code. The ultimate goal of the project is a formal proof that Hyper-V simulates the virtualized hardware for each of the guest operating systems. There are however multiple intermediate goals, the first one being verification of memory safety in a concurrent context. Even this first step relies on establishing, e.g., functional correctness of red-black trees and complex concurrency synchronization protocols.

2 The C Verifier VCC

Both sub-projects employ a C verifier called VCC [6]. Using VCC, the specification of the program is supplied using annotations directly in the program text. Given an annotated C program, VCC performs three steps to conduct a correctness proof (if possible). The reason for this breakdown into several steps is a better separation of concerns and easy integration of different tools: (1) The annotated C code is compiled into an intermediate imperative programming language called BoogiePL [8], which includes the specified properties of the C program rendered as assertions. (2) The input for the following translation step consists of two parts: (a) the BoogiePL code that results from compiling the original C source (including annotations) and (b) axiomatic descriptions (in BoogiePL syntax) of certain aspects of the C programming language, called the BoogiePL prelude. The annotated BoogiePL program together with the prelude is then transformed into first-order predicate logic formulas (verification conditions), which state that the program satisfies the annotated specification. (3) These verification conditions are given to the automatic theorem prover (SMT solver) Z3 [7] to check whether they are valid, which then implies that the original C program is correct w.r.t. the annotated specification.

The possible results Z3 may return are: (1) The formulas are valid (Z3 has found a proof). (2) At least one of the formulas is not valid (Z3 has found a counter-example). (3) Z3 runs out of resources (time or space). In Case (1) above, the program verification was successful. In Cases (2) and (3), the verification engineer has to analyse the problem and correct the error. In Case (2), the counter-example, which can be thought of as an execution trace, can be visualized using a VCC-specific tool with debugger-like interface. In Case (3), one may also find that the program indeed satisfies the annotations. Then new annotations (stronger invariants, helpful lemmas, etc.) have to be added. This process is repeated until a proof is found.

Annotation Language. The annotation language of VCC is guarded by C preprocessor macros. When verifying, a flag is set so that these macros evaluate to keywords specific to VCC, which in turn generates the corresponding BoogiePL code out of them. If a normal C compiler is used (without this flag), all annotations evaluate to the empty string, so that the annotations are transparent for the compilation process.

The VCC annotation language consists of side-effect-free C expressions, extended with first-order quantification, lambda expressions, and a number of constructs used to attach specifications to existing C constructs, e.g., to attach function contracts to function declarations, loop invariants to loops and so forth. Additionally one can also place explicit assertions in the running code, e.g., to help in debugging specifications.

Modularity. Verification in VCC is modular, both with respect to threads and functions. Functions are equipped with contracts in form of pre- and post-conditions, giving all necessary conditions to call the function and the guarantees on the state, when the function returns. Callers are then verified with respect to the contracts, not bodies, of the called functions. The program is verified as if it were executed by a single thread but, to handle concurrency, predicates describing knowledge about the state are weakened at possible points of interleavings to simulate the effects of other threads.

Ghost Fields and Ghost Code. Verification of complex, functional properties of programs has been, up to date, mostly done using interactive, higher-order provers. VCC restricts the specification language not to use any higher order or specialized logics, and instead relies on purely first-order specification for two reasons. The first one is automation: development of automatic first-order systems has received much more attention than either specialized logics or higher-order logics. The second reason is the belief that first-order logic is better understood by "ordinary" programmers.

This restriction comes at a price. For example, the graph reachability relation, a crucial concept in specification of recursive data structures, is not expressible in first-order logic. In places where such relations are needed we introduce *ghost fields* in data structures to keep track of the set of reachable objects and use *ghost code* to update these fields whenever the data structure is updated. Ghost code is introduced only to facilitate verification and is syntactically restricted not to alter execution of the physical code. We thus verify the program with ghost code, but the set of reachable physical states is the same with and without ghost code. VCC supports manipulation of a number of ghost-only types, including maps (from pointers and integers into arbitrary types) as well as entire states of execution, which can be captured and used to evaluate expressions in them. Additionally, new user-defined ghost data types can be specified at the level of C, using function symbols and axioms.

Ghost code and ghost state allow for introduction of abstraction layers: for example a red-black tree, from the user's point of view, is represented as a mathematical map, greatly simplifying reasoning. Additionally ghost state is used to capture protocols using flags, ownership transfer, and two-state invariants. A significant advantage of ghost code is that it is well understood by "ordinary" programmers, being much like code introduced for debugging and runtime assertion checking.

We have been able to specify and verify multiple recursive data structures, as found in the Hyper-V code, some complex synchronisation primitives (spin locks, reader-writer locks, rundowns, custom algorithms for message passing) and specify a good deal of data structure invariants. We currently do not face expressiveness problems with the restriction to first-order logic.

Object Invariants and Ownership. One way to capture global properties of a software system is to define invariants for data structures (i.e., `struct`s in the case of C) used in the program. With VCC, such invariants can be given by annotating a `struct` with (arbitrarily many) `invariant` clauses. To enable modular reasoning about properties of complex data structures (e.g., pointer structures or nested `struct`s), and to capture relations between data structures, the concept of *ownership* between structured data is used (VCC's ownership model is an extension of the one used in the Spec# methodology [13]). Every `struct` has exactly one "owner" and can itself own arbitrarily many structures. At the top of the ownership hierarchy, `struct`s can be owned by executing threads. The ownership relation is provided explicitly in annotations by the verification engineer, and it reflects his/her abstract knowledge about the data structure and how it is used.

3 The Deduction System Z3

While there are ongoing efforts to accommodate different theorem provers into the VCC tool chain (including interactive ones, like Isabelle/HOL [5]), the Verisoft XT sub-projects *Hypervisor* and *Avionics* mostly use the SMT solver called Z3 as the underlying deduction engine. SMT stands for Satisfiability Modulo Theories. SMT solvers decide satisfiability of first-order formulas in presence of background theories like integer or bit-vector arithmetic, arrays, etc. Conjunctions of literals from the theories are tested for satisfiability by *decision procedures*.

While most SMT solvers handle only quantifier-free, decidable logics, Z3 is also capable of solving problems with quantified formulas. Universal quantification is handled either with incomplete instantiation heuristics, complete model-based instantiation or with superposition based calculi. VCC uses only the first method, due to performance reasons.

We shall now go through a simple program, how its correctness is encoded as an SMT formula, and how the SMT solver checks its satisfiability.

```
void compute_abs(int *x)
{ if (*x < 0)
    *x = -(*x);
  assert(*x >= 0); }
```

For brevity we skip proof obligations stemming from checking validity of memory accesses, so by correctness of the program we mean that the explicit assertion never fails. VCC generates the following three formulas. The unsatisfiability of the conjunction of them implies the correctness of the program.

$$\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H,p,v),p) = v \qquad (1)$$

$$(\mathrm{rd}(H_0,x) < 0 \ \wedge H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x))) \vee (\neg(\mathrm{rd}(H_0,x) < 0) \wedge H_1 = H_0) \qquad (2)$$

$$\neg(\mathrm{rd}(H_1,x) \geq 0) \qquad (3)$$

The first conjunct (1) is one of the axioms describing the behavior of the heap: if one writes v at heap location p, then reading from the updated heap at p will yield v. There are more axioms describing the heap, as well as a few hundred other axioms encoding the rest of the VCC verification methodology, which are not needed for this example. The second conjunct (2) encodes the semantics of the `if` statement: either the condition was true, and the new heap is constructed by updating the old heap at x, or the condition was not true, and the new heap equals the old heap. Finally, conjunct (3) says that the assertion is violated. A model for the conjunction of the three formulas corresponds to a program execution where the assertion is violated. If such a model does not exists (i.e., the formula is unsatisfiable), then the program is correct.

Let us now examine how the SMT solver establishes unsatisfiability of the conjunction. Generally, at first it ignores function symbols and quantified subformulas, and looks for a propositionally satisfying truth assignment to the atoms (where by atoms we mean applications of predicates and quantified subformulas) of the formula. Assume the first assignment the solver considers is to make the quantified formula (1) and $H_1 = H_0$ true and to make $\mathrm{rd}(H_0,x) < 0$ and $\mathrm{rd}(H_1,x) \geq 0$ false. The uninterpreted function decision procedure (DP)[1] infers $\mathrm{rd}(H_0,x) = \mathrm{rd}(H_1,x)$ based on $H_1 = H_0$. The linear arithmetic DP perspective on this is $\neg(a < 0)$ and $\neg(b \geq 0)$ (where a and b are abstractions of $\mathrm{rd}(H_0,x)$ and $\mathrm{rd}(H_1,x)$ respectively, the arithmetic DP is not concerned about their internal structure), and now it receives the literal $a = b$, and says that the resulting set of literals is unsatisfiable. Such situations, where the current literal assignment is unsatisfiable, are called *conflict*. Then, the DPs narrow down the subset of currently assigned literals, that actually participate in the conflict. A disjunction of negations of those literals is called *conflict clause* and is a tautology modulo background theories. In our case the conflict clause is: $\mathrm{rd}(H_0,x) < 0 \vee \neg(H_1 = H_0) \vee \mathrm{rd}(H_1,x) \geq 0$. Because the conflict clause is a tautology, conjoining it to the input formula does not change the satisfiability status of the input formula. After we conjoin it, the search space narrows down: a class of literal assignments, including the current one, is propositionally excluded. In our case, the only remaining possible propositional assignment for our formula is to make the quantified formula (1), $\mathrm{rd}(H_0,x) < 0$, and $H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x))$ true, and to make $\mathrm{rd}(H_1,x) \geq 0$ false. These literals do not conflict at the ground level, and thus the quantified formula is instantiated. The following tautology is conjoined to the original formula:

$$(\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H,p,v),p) = v) \Rightarrow \mathrm{rd}(\mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x)),x) = -\mathrm{rd}(H_0,x)$$

The propositional assignment is then extended to make the literal $\mathrm{rd}(\mathrm{wr}(H_0,x,\ -\mathrm{rd}(H_0,x)),x) = -\mathrm{rd}(H_0,x)$ true, which is the only way to satisfy the implication above. The uninterpreted function DP infers $\neg(-\mathrm{rd}(H_0,x) \geq 0)$, and the linear arithmetic view of $a < 0$ and $\neg(-a \geq 0)$ generates a conflict clause, which blocks the only remaining propositional assignment. The formula is thus deemed unsatisfiable.

This description of the proof search highlights the most important features of SMT solvers. (1) Handling of the propositional structure of the formula is very much like in the modern, extremely efficient, propositional SAT solvers. (2) Multiple DPs build into the SMT solvers need to cooperate, in our example the uninterpreted function DP propagated equality to the linear arithmetic DP. (3) The search space is narrowed by the conflict clauses that the SMT solver learns during the search. (4) The quantified formulas are handled by instantiation and, therefore, we need some heuristic to decide how to instantiate.

The heuristic which we use in Z3 for VCC is based on *E-matching*.[2] Certain subterms of the quantified formula body are designated as *triggers*. The SMT solver searches for substitutions, which make the triggers equal to some subterms of currently assigned ground literals, in hope that an instance sharing subterms with the current ground problem will be relevant for the search. For example, let us assume the trigger for the heap axiom is $\mathrm{rd}(\mathrm{wr}(H,p,v),p)$. At the moment, where we needed the instantiation, the ground literals included $\neg(\mathrm{rd}(H_1,x) \geq 0)$ and $H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x)))$. If we take

$$\sigma = [H := H_0, p := x, v := -\mathrm{rd}(H_0,x)] \ ,$$

then

$$\sigma(\mathrm{rd}(\mathrm{wr}(H,p,v),p)) = \mathrm{rd}(H_1,x) \ ,$$

because $H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x)))$ (we thus consider the currently assumed equality atoms).

The triggers can be either explicitly supplied to the SMT solver, or the solver can select them using a simple heuristic. The explicit triggering has proven to be a powerful, if somewhat arcane, tool for building custom SMT theories, like the one describing a particular verification methodology [9, 10, 2, 12]. The

[1]Everything that is not pure equality and propositional connectives is treated as theory in SMT. This includes the uninterpreted function theory, which could be axiomatized with $\forall x_1,...,x_n,y_1,...,y_n.x_1 = y_1 \wedge ... \wedge x_n = y_n \Rightarrow f(x_1,...,x_n) = f(y_1,...,y_n)$ for every function symbol f with arity n.

[2]The E in "E-matching"stands for equality.

triggering annotation can be viewed as a logic programming language used to implement a theory to be executed by the SMT solver. Of course one could also implement the theory inside an SMT solver, which would likely be much more efficient, but the implementation would be much harder. Given how fast such a theory evolves during development of a verification tool, it seems counterproductive in most cases.

The problems stemming from deductive software verification are quite different from the ground SMT problems that mostly result from hardware verification. For example, the number of conflicts per time unit that the solver finds tends to a thousand times smaller for software. This is not enough to calibrate usual heuristics for ordering propositional assignments. Also the implementation of various indices for E-matching is crucial for performance. Z3 is very good with ground SMT problems, and it is definitely the leading solver for quantified queries. This is partially a result of close collaboration between the authors of Z3 and researchers using it for software verification.

4 Conclusion

The *Hypervisor* sub-project involves up to 20 people working, mostly on specification of the Hyper-V, for three years, making it one of the largest formal verification efforts ever attempted. While the *Avionics* sub-project is smaller, it still involves the full functional verification of thousands of lines of complex C code.

Based on experience and results from the first half of the project, we can conclude that verifying concurrent system software written in C and assembly code is difficult and at the edge of the state of the art of software verification. But given sufficient resources, it can be done and, considering the importance of correct system software, is useful and feasible on an industrial scale. The Verisoft XT project emphasises the well-known fact that software verification is one of the most important applications of automated deduction. The project's success relies heavily on recent advances in deduction technology.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Martin Giese, and Philipp Rümmer. Practical aspects of automated deduction for program verification. *KI*, 2009. *In this issue*.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.

[3] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification. In *Proc. Embedded World Conference*, 2009.

[4] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *Proc. SAFECOMP 2009*, LNCS. Springer, 2009. To appear.

[5] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie: An interactive prover-backend for the Verifiying C Compiler. *Journal of Automated Reasoning*, 2009. To appear.

[6] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proc. TPHOLs 2009*, LNCS 5674, pages 23–42. Springer, 2009. Invited paper.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS 2008*, LNCS 4963, pages 337–340. Springer, 2008.

[8] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[9] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, Palo Alto, 1998.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI 2002*, SIGPLAN Notices 37, pages 234–245. ACM, 2002.

[11] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *Proc. 1st International Workshop on Microkernels for Embedded Systems (MIKES)*, 2007. Available at `http://ertos.nicta.com.au/publications/papers/Kuz_Petters_07.pdf`.

[12] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Proc. POPL 2008*, pages 171–182. ACM, 2008.

[13] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proc. ECOOP 2008*, LNCS 3086. Springer, 2004.

Contact

Prof. Dr. Bernhard Beckert
Karlsruhe Institute of Technology
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
Phone: +49 721 608-4025
Email: beckert@kit.de

Michał Moskal
European Microsoft Innovation Center
Ritterstrasse 23, 52072 Aachen, Germany
Email: michal.moskal@microsoft.com

Bild

Bernhard Beckert is a professor of Application-oriented Formal Verification at the Karlsruhe Insitute of Technology. His research interests include automated deduction, non-classical logics, and formal methods in software engineering.

Bild

Michał Moskal is a researcher at the European Microsoft Innovation Center, Aachen, Germany, closely collaborating with the Research in Software Engineering Group at Microsoft Research, Redmond, USA. He is also a PhD student at the Computer Science Institute of the University of Wrocław, Poland. His research currently focuses on software verification with SMT and the VCC verifier in particular.