Formally verified source code optimization for safety critical software

Wendlasida Ouédraogo, Danko Ilik, and Lutz Straßburger

Introduction and context A system is considered to be safety-critical[1] if its failure could have dramatic effects on the environment, human life, or major property damage. Due to their dangerousness, those systems are subject to standards, that ensure that providers deliver trustworthy systems. Siemens Mobility France, the World's foremost supplier of computer systems (hardware and software) that serve for automatic control and operation of metro trains and in particular driver-less trains, develops SIL4[2] certified systems – the highest Software Integrity Level required by the European standard EN 50128[3]. Those systems are proven to have a probability of random failure lower than 10^{-8} per hour.

This achievement results from a development process allowing to detect when a given program does not run conformably to its specification. This process starts with a translation of a given program specification into abstract descriptions of that program called B machines[4]. Those B-machines are then successively transformed into concrete descriptions of the program – B implementations – whose syntax is very close to imperative programming languages. The tools used to write those B elements ensure the coherence between the B machines and the B implementations and require developers to write formal proofs if needed. Those B implementations are then translated into a subset of Ada – Ada PSC– that implements the Vital Coded Processor technique [5], a technique based on modular arithmetic and probabilities to detect inconsistencies between software source code and its execution behaviour. In this technique, variables are associated with N-bit values (variable signatures) that predictably change depending on the program instructions. At compile time, a software called Signature Predetermination Tool (SPT) computes the expected signatures for vital variables. Those signatures are then embedded in the program executable. Execution is performed by two processors: a vital co-processor that is in charge of signature updates (triggered at each instruction) and a standard processor that is in charge of the other tasks (variable value computation, flow execution...).

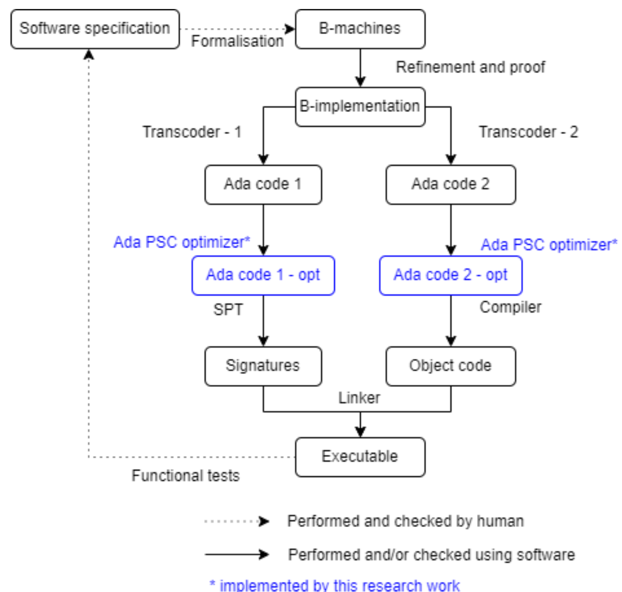

Figure 1: Compilation process

From the point of view of developers, signature management is transparent when analysing an Ada PSC program source code. Each operation (arithmetic, test condition, assignment ...) is performed using library procedures that ensures the correct update of signatures. Consequently, from a point of view of standard compilers, Ada PSC instructions are just external function calls and so, they cannot perform usual optimizations such as *dead code elimination* or *expression simplification*.

The translation from B implementation to Ada PSC is performed by two translators according to the N version programming methodology[6]. The output of one of those translators is processed by a certified Ada compiler while the output of the other compiler is processed by the SPT. The outputs of both, SPT and Ada compiler, are then linked with the operation library to build an executable.

Using that development process, the precomputed signatures and the signatures and their run-time value will differ if any error in the translator, the compiler, the linker, or at run time (with a probability of failure lower than 2^{-n}, where n is the number of bits used to encode variable signatures).

A formally verified optimizing compiler for Ada PSC We focused our research on the optimization of Ada PSC source code because optimization at B level would require B proof repair. However, the B method does not manipulate proof objects, but merely calls black-box automated proof procedures via tactics — and optimization at object code level could be incompatible with the Vital Coded Processor technique, leading to errors in signature verification. Our Ada PSC to Ada PSC compiler is structured as follows:

- A formally verified front-end: We built a formally verified front-end in Coq using Coqlex[7], a software that generates formally verified lexers, and menhir[8], a software that generates formally verified parsers. Those two pieces of software allow to transform a given source code text into an abstract syntax tree (AST).
- Formally verified optimizations: We implemented a big step semantics (using Coq inductive properties) for Ada PSC that interprets the operation library, giving a real advantage over standard Ada optimizing compilers that interprets Ada PSC instructions as a sequence of external function call. Then we implemented AST `->` AST functions (optimizers) that are in charge of performing optimizations by AST transformations. Some of those optimizers are based on Meyer's dynamic annotation rules[9, 10] that we implemented and proved correct in Coq. Then we defined semantic equivalence property (using Coq inductive properties) and proved that all the optimizations we implemented preserve the behaviour of the input source code.
- A verified back-end: We implemented our compiler back-end as an AST `->` Ada PSC code function. This function is the reciprocal function of the front-end. We used that property to implement translation validation[11] and thus, ensure that every Ada PSC text generated by our compiler back-end is correct.

We successfully run our compiler on a real automatic train control program consisting of thousands of source code. That compiler implements well know optimizations such as *constant propagation*, *expression simplification* (14% of expressions have been simplified), *dead code elimination*, *loop unroll* (less than 1%). It also implements Ada PSC related optimisations that helped to reduce operation library call (12% of branch related operations), the number of signatures to compute by 1%, the size of the STP output by 10% and thus, the number interaction between the processors.

Acknowledgements During this work, Wendlasida Ouédraogo was supported by a CIFRE thesis project between Siemens Mobility France and Inria Saclay.

References

[1] J.C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, 2002.

[2] H Jansen and H Schäbe. Computer architectures and safety integrity level apportionment. *WIT Transactions on The Built Environment*, 74, 2004.

[3] T Myklebust, T Stålhane, and N Lyngby. Application of an agile development process for en50128/railway con-formant software. 2015.

[4] J-R Abrial, Matthew KO Lee, DS Neilson, PN Scharbach, and Ib Holm Sørensen. The b-method. In *VDM'91 Formal Software Development Methods: 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, October 21–25, 1991 Proceedings*, pages 398–405. Springer, 2005.

[5] Daniel Dollé. Vital software: Formal method and coded processor. In *ERTS'06*, Toulouse, France, 2006.

[6] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[7] Wendlasida Ouedraogo, Lutz Straßburger, and Gabriel Scherer. Coqlex: Generating formally verified lexers. *The Art, Science, and Engineering of Programming*, 8(1), 2023.

[8] François Pottier and Yann Régis-Gianas. The menhir parser generator. http://gallium.inria.fr/fpottier/menhir, 2016.

[9] Uwe Meyer. Techniques for partial evaluation of imperative languages. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 94–105, 1991.

[10] Uwe Meyer. Correctness of on-line partial evaluation for a pascal-like language. *Science of computer programming*, 34(1):55–73, 1999.

[11] M Siegel, A Pnueli, and E Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.