Certified Compilation in the TELEPERM XS Nuclear Safety I&C Platform

Alexandre Berard
Framatome SAS Grenoble, 23 Chemin du Vieux Chêne, 38240 Meylan, France
alexandre.berard@framatome.com
Université Grenoble Alpes - CNRS - Grenoble INP - Verimag, France
alexandre.berard2@univ-grenoble-alpes.fr

Richard B. Kreckel
Framatome GmbH, Paul-Gossen-Str. 100, Erlangen, Germany
richard.kreckel@framatome.com

The large safety instrumentation & control (I&C) systems in civil nuclear power plants (NPPs) are mainly safe-shutdown systems (reactor protection) or limitation and control systems. Framatome's established TELEPERM XS (TXS Core) product family is a digital I&C system platform to cover all these applications. We illustrate the role of verification in the different stages of the software production toolchain, focus on the formal compilation process, and discuss the contribution of the CompCert certified compiler to the safety case of the product. Scrutinizing the object code produced by this compiler has exhibited suboptimal run-time performance in a certain simple but recurring generated code pattern. We explain how formal methods allow us to address this issue in the compiler while simultaneously reducing its trusted computing base (TCB), thereby strengthening the safety case rather than merely preserving it.

1 Introduction

I&C systems important to safety in NPPs are subject to strict requirements on their specification, development process, and verification & validation (V&V). For example, most national safety authorities require the application of the IEC codes and standards under the umbrella of IEC 61513 [5]. Requirements for software-based systems are specifically elaborated in the IEC 60880 [2].[1]

While that régime steers clear of requiring specific programming paradigms, it does impose certain constraints for the most safety-critical functions like the absence of recursion or dynamic heap-based memory. This may help explain why function block diagrams (FBDs) like those proposed in [3] continue to be the prevailing programming paradigm for reactor protection, limitation, and control.

Systems based on FBDs offer a range of advantages. Their dataflow-based structure conveniently caters to the process-centric requirements and the perspective of control engineers. It provides inherent safety from a range of programming errors. The availability of a pre-developed and fully qualified function block (FB) library disburdens the control engineers from low-level programming tasks. Composite FBs (CFBs) [4] suggest themselves as a natural application of the principles of component-based software development from specification of I&C functions over their implementation all the way to V&V activities [28]. These advantages are well known and are described in textbooks, e.g. [14].

[1] The IEC 61513 standard is for nuclear installations what IEC 26262 is for automotive sector. Both nuclear standards IEC 61513 and 60880 are currently being updated.

Mircea Marin, Adrian Crăciun (Eds.): FROM 2026
EPTCS 452, 2026, pp. 92–103, doi:10.4204/EPTCS.452.7

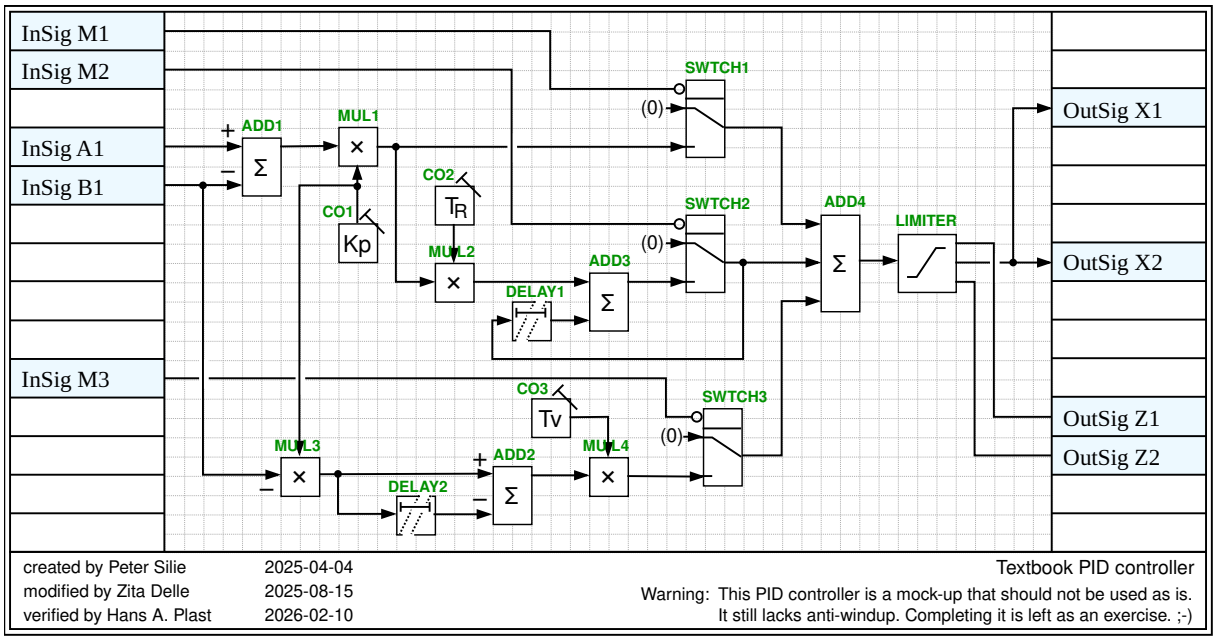

Figure 1: A function block diagram (FBD) implementing a simple standard form PID controller. Two analog input signals A1 and B1 and three binary active-high 'disable' signals M1…M3 are interconnected with 17 function blocks and 24 internal signals to produce the four output signals X1, X2, Z1, and Z2.

The nuclear IEC standards also impose requirements on toolchains for translating FBDs into code for execution on programmable logic controllers (PLCs) performing safety functions in NPPs. There is broad consensus [25] that in addition to comprehensive validation of the final result the stages of a qualified toolchain for producing nuclear safety application software must (i) produce output that is then *comprehensively verified* using diverse techniques and (ii) provide convincing evidence that they *preserve the semantics* of the input in the produced output. In this vein, chapter 14 of IEC 60880 requires mitigation against errors which could potentially be introduced by translation tools.

This paper is organized as follows: In Section 2 we introduce the two stages of Framatome's TXS Core toolchain [11, 13, 1]: Verified code generation and certified compilation. The second stage is based on the CompCert[2] verified compiler. We will show why the first stage naturally produces certain repetitive patterns of code. In section 3 we will present work which aims at improving the executable code compiled from these patterns. This obviously involves the compiler itself, but also turns out to require a small re-organization of the code generation. Section 4 overviews related work and in section 5 we will put this into the perspective of the general safety case.

2 The TXS Core Fully Verified Toolchain

At the user level, Framatome's TXS Core toolchain[3] begins with a graphical editor: I&C engineers drag graphical FBs from a menu of over 100 different pre-defined and qualified FB types and intercon-

[2]CompCert's official sources: https://github.com/AbsInt/CompCert.

[3]To be precise, we describe the TXS Core 4.5 platform. Earlier versions differ in several details.

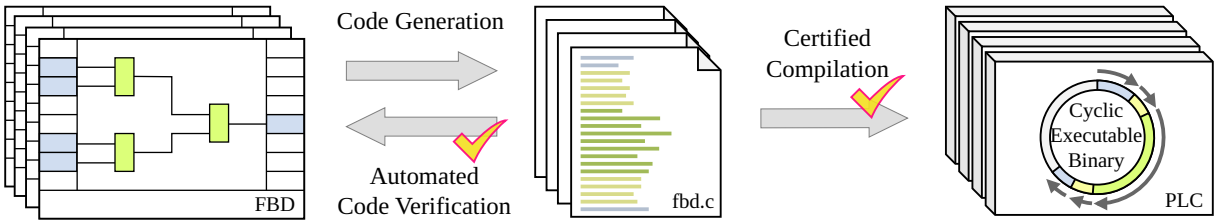

Figure 2: The TXS Core toolchain translates FBDs into certified binaries.

nect them with signals representing the dataflow between the FBs and I/O. This results in FBDs or in CFB types which can then be instantiated on FBDs. I/O comes in several forms: It is either electrical (4...20 mA analog signals, 24 V binary signals, etc.) driven by dedicated I/O modules communicating with the PLC over a backplane bus or it consists of digital signals transmitted in network messages using a deterministic protocol. Fig. 1 shows a FBD which could alternatively be the implementation of a CFB.

The subsequent stages (see Fig. 2) are fully automated. No user intervention whatsoever is required (in fact, not even permitted). Below, we will unfold them.

2.1 Code Generation

Using the persisted graphical FBDs as input, a code generator first produces ISO C code. Largely, it consists of function calls to the standardized, qualified, and pre-compiled FBs and of glue code representing the dataflow between the blocks. This ISO C code is often long but always linearly structured and is itself suitable for a spectrum of analyses.

A note on the use of C: ISO C has proven to be a rewarding intermediate interface. There is an international standard [6]. When restricted to a subset, the language's ambiguities can be fully avoided. There exists an ample toolscape for performing static analyses (Astrée, FramaC, etc). Last, but not least, it is the lingua franca for interfacing with the gamut of commercial simulator platforms so that the entire I&C application can be close-looped with plant models for all purposes from full-scope simulation to operator training [23].

Nevertheless, translation from FBDs to ISO C code requires careful verification to rule out any error in the code generator.

2.2 Automated Code Verification

In order to ensure the correctness of the translation from the FBD specifications to ISO C code, Framatome has developed a fully automated independent verifier operating on the persisted graphical FBDs and the generated code as input. This verifier exploits the fact that an FBD is a directed acyclic graph of edges connecting FB ports (the netlist). Every signal line connecting two FBs is equivalent to the constraint of causality: The source $\mathrm{FB}_{\mathrm{src}}$ must be invoked before the destination $\mathrm{FB}_{\mathrm{dst}}$ within each PLC cycle. In the generated code, causality is equivalent to the constraint that the source $\mathrm{FB}_{\mathrm{src}}$ must be called before the destination $\mathrm{FB}_{\mathrm{dst}}$.

For example, in Fig. 1, FB instance ADD1 must be called before FB instance MUL1, symbolically $\mathsf{ADD1} <_{\mathsf{FB}} \mathsf{MUL1}$. We also find $\mathsf{CO1} <_{\mathsf{FB}} \mathsf{MUL1}$, $\mathsf{MUL1} <_{\mathsf{FB}} \mathsf{SWITCH1}$, and by transitivity $\mathsf{ADD1} <_{\mathsf{FB}}$

SWITCH1, etc. On the other hand, there is no causality relation between blocks ADD1 and CO1. Obviously, the interconnections on the FBD induce a strict weak ordering between all FB instances.

A special case is the integration loop between FB instances ADD3, SWITCH2, and DELAY1 in Fig. 1: On first glance, this creates a strongly connected component which violates the ordering. Ordering is re-established by breaking the DELAY-type block into two parts where the second part is *not* causally related to the first part within the same cycle (but in the next cycle).

This strict weak ordering of FBs suggests an efficient node-by-node and edge-by-edge method for the translation validation of the generated ISO C code. A parser reads the generated code and performs the following program:

- Construct the list of all FB instances, attributed by their types and, if applicable, all other artifacts like the FBs' parameters, etc.
  - Verify that each element in this list has an exact counterpart in the specified FBDs and that no other FB instances than the ones in this list are specified.
- Construct the set of all I/O drivers and network messages, together with their counterparts (I/O module, communication partner) and the lists of signals they contain.
  - Verify that each element of this set has an exact counterpart in the specification and no other I/O drivers or network messages are specified.
- Construct the 'netlist' of all interconnections on the FBDs. The interconnections are attributed by signal types, possible signal negations, their sources and destinations.
  - Verify for each signal in this netlist that it is connected with source and destination(s) as specified in the specification FBD, that their attributes match, and that no other connections exist in the FBD.
  - For each signal in the netlist, verify that the ordering relation $<_{\mathsf{FB}}$ of its source and destination(s) in the generated code complies with the one in the FBD.

Note the softness of the last item: One can only verify causality for all FB instances. This is a necessary and sufficient condition for the translation's semantic preservation. It would not be possible to compare the sequentialized list of FBs in the generated code with a list of FBs in the specification because the latter list is not unique. After all, the FBD contains many FBs which are incomparable w.r.t. $<_{\mathsf{FB}}$ (like ADD1 and CO1 in Fig. 1).

Successfully performing the above program proves that the semantics of the FBD is indeed preserved in the generated application software: the structures are the same and the generated code respects the FBD's implicit causal relations.

An important overarching non-functional safety requirement on the automatic verifier is that its implementation must be fully diverse from the code generator and not share any source code with it. Without this independence, it would be possible for the two tools to share a hidden fault leading to the non-detection of an error in the generated code.

Only when the verifier has successfully terminated, the compiler will be invoked on the generated ISO C code.

2.3 Certified Compilation

In a second stage, the generated ISO C code is compiled to ArmV7 object code by the CompCert certified compiler [20]. The object files can then be linked with any of several ELF linkers. CompCert performs a series of optimizing transformations based on internal intermediate representations.

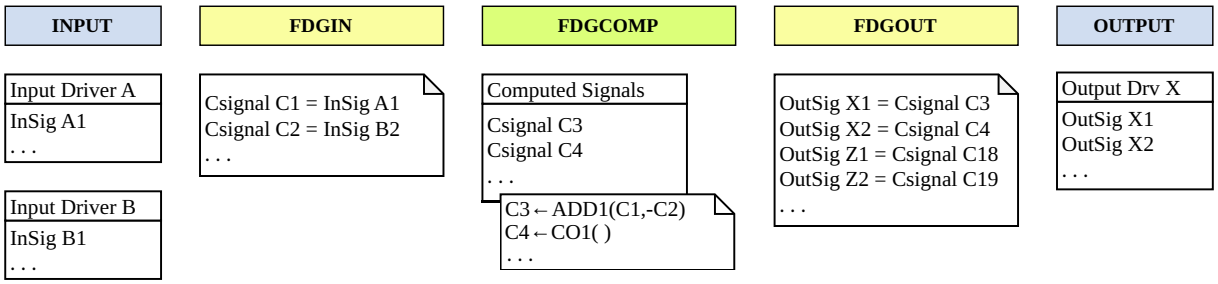

Figure 3: Data moves for computing the FBD from Fig. 1 in the TXS Core cycle. The FDGIN (FDGOUT) function routes input (output) signals from one structure to another one.

These transformations are fully formally proven to preserve the semantics. This proof is formalized and automatically checked by the Rocq Prover (formerly Coq) each time CompCert is built from sources.

Although formally rigorous, CompCert's proof of correctness is limited by its Trusted Computing Base (TCB), which is extensively studied in [22] and mentioned in sections 3.1 and 3.3. Within TXS Core, the use of CompCert extensions which are not proven correct is strictly avoided.

Beyond that, studies subjecting compilers to synthetically generated C code have independently confirmed that rate of bugs in CompCert is orders of magnitude below that of usual compilers [27].

3 CompCert Improvements

The generated ISO C code compiled by CompCert benefits from its safe proof of semantic preservation, but it suffers from its lack of some state-of-the-art code optimizing algorithms that many other compilers have. In fact, the introduction of such algorithms must be followed with a proof of their semantic preservation, which require significant work, and may require changing some aspects of CompCert's theory models. On top of CompCert's already existing algorithms such as dead code elimination and constant propagation, Chamois-CompCert[4], a frequently updated fork of CompCert's official releases, features several more aggressive code optimizations and uses a framework that facilitates their writing.

The following sections will show how both CompCert and Chamois-CompCert fail to optimize TXS Core's generated ISO C code and explains how our recent Chamois-CompCert improvements solve this issue.

3.1 Suboptimal Patterns Identified in Generated Code

The code generated from FBDs contains long repetitive patterns of assignments, essentially present in places where we route I/O signals towards and away from compute structures. Those can be seen in Fig. 3, especially in the FDGIN and FDGOUT functions. Also, each signal is translated into a small C structure, causing the compiler to translate the said assignments into heavy structure copies.

This is where we observe a weakness: in CompCert semantics, a structure copy is translated into a compiler built-in function dedicated to copying data, which has valid semantics, but needs to be expanded. And this built-in function, abbreviated here with Memcpy (not the C libary `memcpy` function), is immutably propagated throughout all the intermediate representations of code, all the way before the

[4] Chamois-CompCert's official sources: https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert

back-end part of the compiler dedicated to emitting assembly code. Such built-in functions are not sensitive to any code optimization, which explains why these patterns that extensively use Memcpy are poorly compiled. Moreover, their expansion to assembly code is directly performed by OCAML code with no verification; this code belongs to the TCB of COMPCERT [22]. Causing any change to this unproven code is not recommended, especially if we want to apply complex and unproven optimizations over it. CHAMOIS-COMPCERT is helpless here as well: currently, none of its optimizations improves the generation of built-in functions.

3.2 Overview of CHAMOIS-COMPCERT's Formal Verification Framework

CHAMOIS-COMPCERT possess an Intermediate language Representation (IR), called Block Transfer Language (BTL), plugged to COMPCERT's Register Transfer Language (RTL) IR. BTL is provided with a symbolic execution engine, here dedicated to verify middle-end optimizations. This engine contributes to CHAMOIS-COMPCERT exclusive block verification framework, whose task is to defensively validate code optimizing oracles [12], written in OCAML. Sometimes, the symbolic checker needs some information computed by the oracles to be able to validate these code transformations: this is part of CHAMOIS-COMPCERT Formally Verified Framework, proven to be a rigorous block verification mechanism [8]. This allows the rewriting of operations computed by invariants and memory operations with pointer nonaliasing, allowing e.g. the simplification of operations across several blocks, or the permutation of load and store instructions. Similarly, some back-end optimizations are verified with another symbolic execution engine, but over an abstracted representation of basic blocks of generic assembly operations, obtained by proven translation from the configured target architecture. The latter allows the writing of a postpass instruction scheduling algorithm, where instructions can also be mutated (*peephole optimization*), which we ported to the ArmV7 backend. For our case, validating Memcpy transformations required us to make use of all the cited features.

3.3 Improvements Made to Compiler Built-in Memory Copies

RTL and BTL place Memcpy in an algebraic type dedicated to compiler built-in instructions. Memcpy semantics are simple: it consists in (i) loading contiguous bytes from a source memory location, then (ii) storing those bytes to a destination location, given the amount of data to copy and memory alignment.

Given the form of suboptimal patterns observed in section 3.1, we came up with ideas that would improve COMPCERT's Memcpy. We first found that the offset of each field of a structure is statically known, as they are placed in the same memory block. This made us think about a way to *merge* memory copies where both the source and destination offsets are adjacent. Then, another idea arose: since the alignment of the global memory block is also statically known, the internal structural copies could benefit of a *realignment* analysis, that we use to determine a better width of copy instruction to use. A picture of these ideas is shown in Fig. 4. The addition and usage of realignment analysis conducted us to change several aspects of COMPCERT's inner memory model [10]. The following paragraphs will briefly explain how we incorporated these new ideas within CHAMOIS-COMPCERT.

3.3.1 Merging memory copies with a verified oracle

This part aims at manipulating several Memcpy instructions together, merging those which have adjacent copy addresses. We opt to do this part at the BTL stage, benefiting from its symbolic execution engine capable of validating memory transformations obtained from an untrusted oracle. Using this principle, we

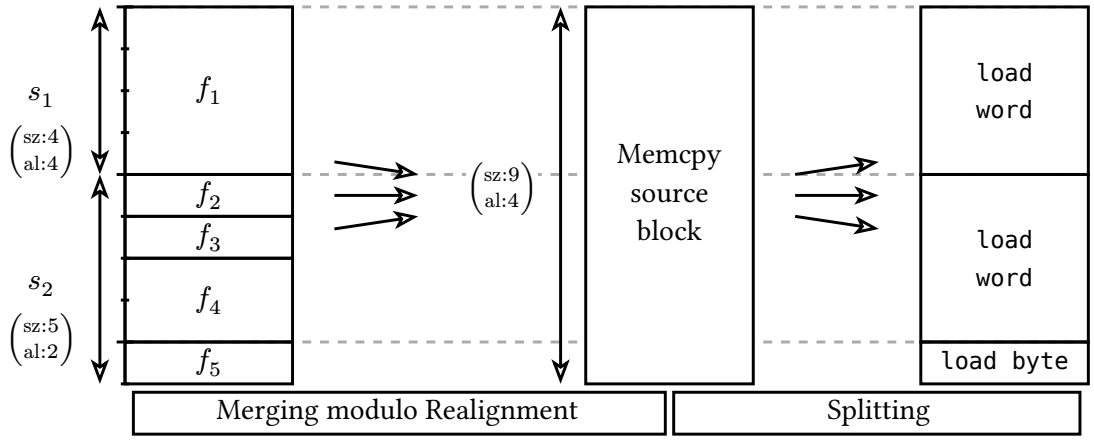

Figure 4: Abstracted representation of our Memcpy manipulations, by viewing only at source addresses of structures in-memory (layout is assumed to be similar for destination addresses). The two structures, s_1 and s_2, are merged and realigned (left part), and so, can be fragmented into a series of two word-sized and one byte-sized loads (right part). The size (sz) and alignment (al) is also specified for each structure, as well as their fields (e.g. f_2 to f_5 for s_2) of varying sizes.

can write a merger oracle that can reorder and simplify instructions – for instance, minimizing the number of virtual registers used in Memcpy addressing arguments, thereby reducing the number of blocks that may differ from one to another. The symbolic checker also supports validating the reordering of memory operations, including Memcpy, through alias analysis [10], which further increases opportunities for merging.

The merger oracle consists in greedily finding valid Memcpy candidates for merging. For this, we reuse the dependency graph from CHAMOIS-COMPCERT instruction scheduling; we prioritize other instructions and keep the encountered Memcpy instructions in a separate set. Once no further instructions can be scheduled, we consider the remaining set of Memcpy instructions and determine whether any of them can be merged. Two Memcpy can be merged only if their memory blocks are contiguous, for both destination and source addresses. Also, we consider the case when merging two copies cause an overlap between the source and the destination blocks. This case made us change the semantics of memory copies, where a writing mode is now specified: *forward* or *backward*.[5] Thus, a copy merge is valid when both copies have the same writing mode, which must be correct with regard to the overlapping case. Although the theoretical foundations for validating memory copy merges via symbolic execution have been fully established, the current oracle does not handle overlapping cases; this is left for future work.

The symbolic checker also needs to be adapted. Originally, it was possible in CHAMOIS-COMPCERT to replace two single word store instructions with a double word one during peephole selection [24]. We extend the same principle for the validation of merged Memcpy, despite the need for further information. Through the oracle, we annotate each merged Memcpy with a list of all the original candidates. This information is used at the rewriting stage in order to validate merges with regard to the source code, relating for instance the address ranges and rewriting modes of each copy from the source with the copy list. The generated annotation must also exhibit the same behavior than the Memcpy it is attached to. Note that the annotation list does not necessarily need to preserve the source order, since the symbolic checker can deal with the reordering of memory operations. For proof simplicity, we still have to provide

[5] This can be seen as analogous to the `memcpy` and `memmove` functions in the C standard library.

the copies in increasing order of address offsets in *forward* copy mode, and vice versa for the *backward* mode.

3.3.2 Realignment analysis

Increasing the width of selected copy instructions used for memory copies, as shown in Fig. 4 would obviously decrease the total number of load and store instructions after expansion. This was done by retaining the global alignment of the memory block as β, which could be a power of $2 \in \{1,2,4,8\}$ in bytes. For global variables, it corresponds to the maximum alignment among all fields in the memory block.[6] The computation happens during constant propagation where more information about Memcpy alignment is propagated: formerly only retaining its size and alignment, we now also registers the couple (o,α), where α is the maximal alignment that can be set, and o being the necessary offset to reach that maximal alignment. This pair is computed with respect to β and the global source and destination offsets of a Memcpy address within the block, under various case-by-case scenarios.

As an example, from Fig. 4, realignment is performed on structure s_2; it is not shown on the diagram, but the realignment analysis propagates $\alpha = 4$ from the previous block (s_1) where $al = 4$. This step is necessary for merging blocks s_1 and s_2 into the bigger block with $al = 4$. Offsets are also necessary, e.g. for a copy of fields from f_3 to f_5, assuming a pointer cast to a structure of valid shape, we can still derivate $\alpha = 2$ given $o = 1$, i.e. we can reach a best alignement of 2 after an offset of 1.

This realignment upgrade allows us to fragment Memcpy into bigger parts at expansion stage. Also, it integrates quite well with the fusion of Memcpy, giving the opportunity of using wider copy instructions across several structures whose assignments were merged.

3.3.3 Formal expansion of memory copies with fragmentation

We wrote a fully proven pass in `BTL` that does a formal expansion of Memcpy instructions. Symbolic execution engines are not involved here: we wrote a general *local* expansion functor that defines an *expanse* function, verified with a *local* simulation proof. More precisely, this function replaces each `BTL` block with a sub-Control Flow Graph (CFG) of `BTL` blocks that locally simulates the source block. And the function was instantiated for the expansion of `BTL` blocks containing Memcpy. The sub-CFG structure allows the expansion of loops, which is useful for the expansion of large Memcpy. The writing and the validation of the *expanse* function was simplified by the use of a *state+error* monad, which propagates two information: the *fresh* registers and labels introduced by the expansion. The *local* simulation proof ensures that the *expanse* function returns a valid transformation, where the state of execution (register set, memory) must be preserved, excluding the *fresh* values newly introduced within the expansion monad.

Expansing a Memcpy instruction consists in fragmenting the copy block in *atomic* parts that can directly be translated to hardware instructions. By definition, an *atomic* Memcpy has the same size as its memory alignment. The fragmentation is not so trivial: with the formerly computed realignment information, we are able to incrementally select wider sizes for the leading *atomic* block. Thus, the resulting sequence of fragmented Memcpy can be decomposed in three parts: (i) a *prologue*, containing all the necessary *atomic* blocks needed prior to reaching the maximum realignment. It has the same size as the offset o computed during realignment analysis. (ii) A *copy part*, containing a repetition of atomic blocks having the size of the best alignment α, and (iii) an *epilogue*, being the remainder of Memcpy block left to be fragmented. An example of *epilogue* is visible in Fig. 4, which corresponds to the last load of a single byte. For large copies, the *copy part* may be done with a loop, since the same copy

[6] This is the alignment of global variables that CompCert provides to the linker.

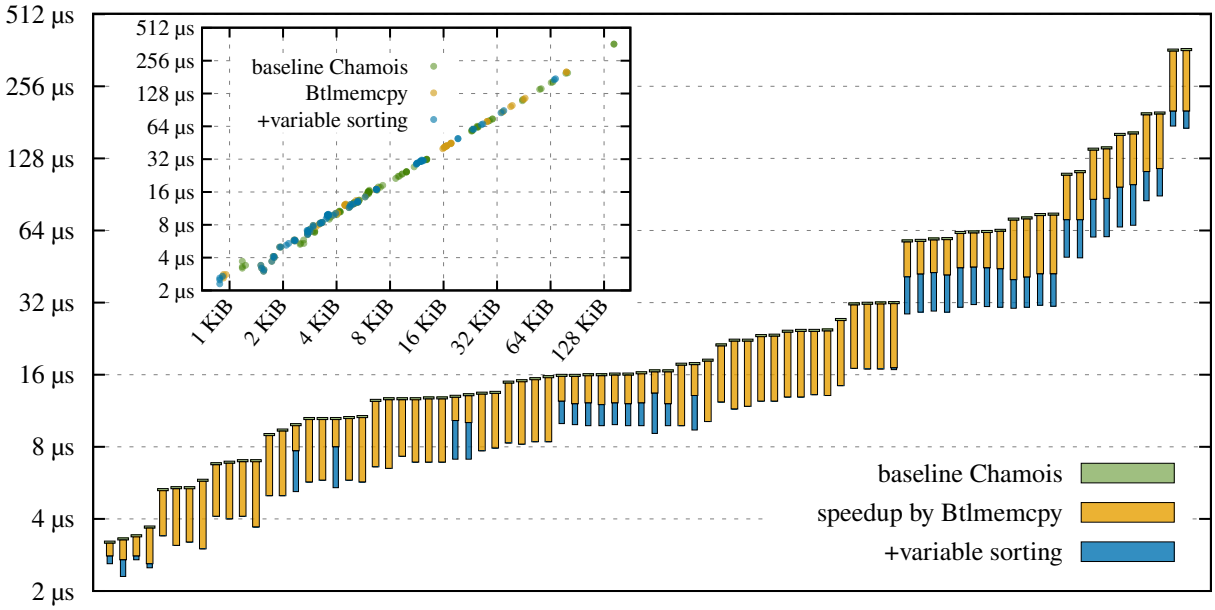

Figure 5: On 82 measured TXS CPUs (arranged horizontally in increasing runtime), the CHAMOIS-COMPCERT improvements (Btlmemcpy) speeds up the FDGIN functions by a factor of 1.64 on average (beige bars). Variable sorting speeds up by another 1.16 (blue bars). The total average speedup is 1.90. The inset compares runtimes with memory footprint of generated and compiled code – it shows that the two are trivially related.

instruction is repeated. The loop body has been designed to only have 4 assembly instructions performing the copy: (i) a post-incremented load, (ii) a post-incremented store,[7] (iii) a pointer comparison upon copy termination, and (iv) the loop's conditional branch.

3.3.4 Reduction of COMPCERT's Trusted Computing Base

By normalizing all Memcpy instructions into *atomic* copies, as explained in Section 3.3.3, we eliminate the need for the complex and unsafe translation of the general-case Memcpy that previously resided in COMPCERT's TCB. This translation is replaced by a significantly simpler one: following the same principle as assembly printing, it consists of a one-to-one translation for each case of *atomic* Memcpy instruction, safely derived from the fully proven expansion described above. Note that we cannot remove this translation from the TCB entirely – for instance, by lowering memory copies to individual loads and stores – due to a semantic notion attached to Memcpy pointer fragments that cannot be bound to a valid byte type [10].

3.4 Evaluation Within the TXS Core Toolchain

To assess the effectiveness in a realistic scenario, we set up an experiment using an actual TXS Core safety I&C system specification on real TXS Core hardware. The results in Fig. 5 show that our im-

[7] Those instructions, which are too architecture-specific to live in operations of `RTL`, are selected with CHAMOIS-COMPCERT peephole algorithm in later passes.

provements to CHAMOIS-COMPCERT significantly speed up the patterns from generated code observed in section 3.1.

We then went one step further and modified the TXS Core code generator to help the compiler. As explained in section 3.3, we can merge adjacent memory copies. However, that highly depends on the layout given in generated structures. Especially, the TXS Core code generator was not initially designed to sort the computed signals such that their (arbitrary) order matches the (fixed) order in the input structures. Just sorting these variables produces more assignments where both the source and the destination are consecutive. This additional improvement of 16% on average is only effective together with the improvement of the compiler.

4 Related Work

A verification method called 'diverse back translation' was proposed for safety-critical code in [16]. The authors manually reconstructed the specification from the implementation and compared it with the original specification. They clearly recognized the tedium involved in the task, even for tiny programs. Much later, it was pointed out in [15] that the process could be greatly accelerated while also obsoleting the need for domain experts performing the task if the source is a graphical FBD specification. The RETRANS tool [21] semi-automates diverse back translation of generated TXS Core C code. It includes in its analysis the comparison of redundant structures – this goes beyond our verifier. The GET-R1 engineering tool for the Russian TPTS-SB nuclear safety I&C platform is similar, but in the absence of a verified compiler it tackles back translation from bytecode back to the FBD specification [7]. Both RETRANS and GET-R1 rely on sequentialization hints from the persisted specification. This is incomplete because sequentialization is not a priori part of the specification – the code generation tool establishes it and often writes it into the specification as annotations. As explained in section 2.2, checking the $<_{\mathsf{FB}}$ relation for all edges in the netlist on the generated code fills this gap. D.-A. Lee et al. have applied translation from FBDs *including* the FBs to Verilog which then made it possible to check the equivalence of this Verilog program with the target C program with HW-CBMC, before the C program was compiled for the PLC [19]. The verifier integrated in the TXS Core toolchain may be the first fully automated, complete, and direct diverse back translation tool used in the nuclear industry. A promising approach to formal compilation of synchronous dataflow languages is the Vélus project [9], a Rocq-written compiler that translates Lustre programs directly to COMPCERT's *Clight*, bypassing C generation and subsequent code verification. However, it does not yet appear integrated into a proper tooling environment suitable for compiling graphical FBDs, as required by nuclear industry standards.

The COMPCERT certified compiler has been used before for safety functions in the nuclear industry [17]. There, additional qualification steps are performed. Notably, the fully linked executable is checked using the Valex tool against a serialization of COMPCERT's internal assembly representation to verify that the assembler and linker did not introduce an error. In the same vein, COMPCERT-ELF is an extension that embeds this verification at the core of the verified compilation chain, extending the verification from the assembly file up to the ELF object file [26].

Implementers have long been optimizing library functions for copying memory blocks by taking advantage of wide registers and handling alignment. Unverified compilers like GCC and CLANG are also able to optimize two adjacent 4-byte copies by one 8-byte copy. But performing memory copy merging as discussed in section 3 seems to be novel [10]. The structures and assignments generated so naturally for embedded PLC software seem to be have escaped popular benchmarks.

5 Conclusion and Outlook

In the nuclear industry, formally verifying that the translation from FBDs to the executable code preserves the semantics does not and will not obsolete comprehensive I&C system validation. Still, a certified toolchain is crucial since it justifies the trust placed in model checking [18] and simulations. After all, these disciplines are usually exercised directly on the FBD specifications.

In this work we were able to address a performance weakness of the CompCert compiler leveraged by the TXS Core toolchain for translating generated C code. This led to code which is not only much smaller and faster, but also reduces the compiler's TCB and this directly improves the safety case. This is very important: While increasing the trust in a nuclear safety platform's toolchain is unlikely to obsolete any V&V activity in test bay, it can significantly support the introduction of local functional changes to FBDs at a later stage.

Some practical and administrative activities are still needed to introduce our improvements of CompCert into the TXS Core toolchain.

Acknowledgements. The authors would like to thank the directors of Framatome's I&C business unit for permission to publish this paper. R. Kreckel also thanks Prof. W. Halang for insightful discussions about diverse back translation and proofs of correctness in general.

References

[1] (2018): *Annex I: Diverse Actuation System for EPR at Olkiluoto-3*. In: *Criteria for Diverse Actuation Systems for Nuclear Power Plants, IAEA TECDOC Series No. 1848* v.1848, IAEA, Vienna.

[2] IEC 60880:2006 (2006): *Instrumentation and control systems important to safety, Software aspects for computer-based systems performing category A functions*.

[3] IEC 61131-3:2025 (2025): *Programmable controllers, Part 3: Programming languages*.

[4] IEC 61499:2007 (2007): *Function Blocks for Embedded and Distributed Control Systems Design*.

[5] IEC 61513:2011 (2011): *Nuclear power plants, Instrumentation and control systems important to safety, General requirements for systems*.

[6] ISO/IEC 9899:2024 (2024): *Information technology, Programming languages, C*.

[7] Mikhail Belonosov & Vladimir Kishkin (2018): *Verification on application program generation and loading for safety systems of nuclear power plants based on the reverse engineering method*. *Nuclear Energy and Technology* 4(4), pp. 223–228, doi:10.3897/nucet.4.31868.

[8] Sylvain Boulmé (2021): *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. Habilitation à diriger des recherches, Université Grenoble-Alpes. Available at https://hal.science/tel-03356701.

[9] Timothy Bourke, Basile Pesin & Marc Pouzet (2023): *Verified Compilation of Synchronous Dataflow with State Machines*. *ACM Transactions on Embedded Computing Systems* 22(5s), pp. 137:1–137:26, doi:10.1145/3608102. ESWEEK special issue including presentations at the 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[10] Alexandre Bérard, Benjamin Bonneau, Sylvain Boulmé & David Monniaux (2026): *Formally Verified Compilation for Aligned Memory Copies*. Available at https://hal.science/hal-05493413.

[11] Siemens Power Corporation (2000): *TELEPERM XS: A Digital Reactor Protection System*. Technical Report.

[12] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux & Alexandre Bérard (2023): *Formally Verifying Optimizations with Block Simulations*. *Proceedings of the ACM on Programming Languages* 7(OOPSLA2), pp. 224:59–224:88, doi:10.1145/3622799.

[13] A. Graf (2004): *Appendix C: Experience in Refitting in the Field of Safety Instrumentation and Control*. In: *Managing Modernization of Nuclear Power Plant Instrumentation and Control Systems*, *IAEA-TECDOC* 1389, INTERNATIONAL ATOMIC ENERGY AGENCY, Vienna.

[14] Wolfgang A. Halang & Rudolf M. Konakovsky (2013): *Sicherheitsgerichtete Echtzeitsysteme*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-642-37298-8.

[15] Wolfgang A. Halang, Bernd Krämer & Leszek Trybus (1995): *Exploiting a Graphical Programming Paradigm to Facilitate Rigorous Verification of Embedded Software*. *The Computer Journal* 38(4), pp. 301–309, doi:10.1093/comjnl/38.4.301.

[16] H Krebs & U Haspel (1984): *Ein Verfahren zur Software-Verifikation*. *rtp. Regelungstechnische Praxis* 26(2), pp. 73–78.

[17] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy & Sandrine Blazy (2018): *CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler*. In: *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, 3AF, SEE, SIE, Toulouse, France, pp. 1–9. Available at https://inria.hal.science/hal-01643290.

[18] Jussi Lahtinen (2016): *Model Checking Large Nuclear Power Plant Safety System Designs*. Aalto University. ISSN: 1799-4942 (Aalto, electronic).

[19] Dong-Ah Lee, Junbeom Yoo & Jang-Soo Lee (2013): *A systematic verification of behavioral consistency between FBD design and ANSI-C implementation using HW-CBMC*. *Reliability Engineering & System Safety* 120, pp. 139–149, doi:10.1016/j.ress.2013.06.006.

[20] Xavier Leroy (2009): *A Formally Verified Compiler Back-end*. *Journal of Automated Reasoning* 43(4), pp. 363–446, doi:10.1007/s10817-009-9155-4.

[21] H Miedl (1998): *RETRANS-A tool to verify the functional equivalence of automatically generated source code with its specification*. In: *Specialists meeting on design and assessment of instrumentation and control systems in NPP coping with rapid technological change*, pp. 137–147. Issue: IAEA-IWG-NPPCI–98/3.

[22] David Monniaux & Sylvain Boulmé (2022): *The Trusted Computing Base of the CompCert Verified Compiler*. In Ilya Sergey, editor: *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 204–233, doi:10.1007/978-3-030-99336-8_8.

[23] S. Richter & J.-U Wittig (2003): *Verification and validation process for safety I&C systems*. *Nuclear Plant Journal* 21, pp. 36–38+40.

[24] Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse & Nicolas Nardino (2022): *Formally Verified Superblock Scheduling*. In Andrei Popescu & Steve Zdancewic, editors: *CPP 2022: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ACM Digital Library, Philadelphia, United States, pp. 40–54, doi:10.1145/3497775.3503679.

[25] B. E. L. V (Belgium), BfS (DE), CSN (Spain), ISTec (DE), NII (GB), SSM (Sweden) & STUK (Finland) (2010): *Licensing of safety critical software for nuclear reactors. Common position of seven European nuclear regulators and authorised technical support organisations*. Technical Report SSM–2010-01, BEL V (Belgium).

[26] Yuting Wang, Xiangzhe Xu, Pierre Wilke & Zhong Shao (2020): *CompCertELF: verified separate compilation of C programs into ELF object files*. *Proceedings of the ACM on Programming Languages* 4(OOPSLA), pp. 1–28, doi:10.1145/3428265.

[27] Xuejun Yang, Yang Chen, Eric Eide & John Regehr (2011): *Finding and understanding bugs in C compilers*. *ACM SIGPLAN Notices* 46(6), pp. 283–294, doi:10.1145/1993316.1993532.

[28] Wei Zhang, Wolfgang A. Halang & Christian Dietrich (2005): *Specification and Verification of Applications Based on Function Blocks*. In Colin Atkinson, Christian Bunse, Hans-Gerhard Gross & Christian Peper, editors: *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, Springer, Berlin, Heidelberg, pp. 8–34, doi:10.1007/11591962_2.