Automated Integration of Safety Mechanisms into Functional Software for Safety-relevant Systems

Rolf Schmedes[0009−0008−0326−1245], Gregor Nitsche[0000−0002−5232−0976], Ralf Stemmer[0000−0002−8302−7713], Kim Grüttner[0000−0002−4988−3858]

German Aerospace Center, Germany `{rolf.schmedes, gregor.nitsche, ralf.stemmer, kim.gruettner}@dlr.de`

Abstract. In the development of safety-relevant systems, the integration of safety software into functional software is crucial for reliable and safe operation. This paper presents a novel semi-automated process designed to integrate software safety mechanisms into safety-relevant systems efficiently. Leveraging a model-driven engineering approach, the method initially separates functional and safety source code and then subsequently combines them in a semi-automated weaving step, producing functionally safe source code, ready for compilation. This approach incorporates expert safety engineering knowledge during the setup phase, facilitating the integration process. The proposed methodology not only enhances cost-effectiveness and reduces human error but also supports the quick evaluation of various safety configurations. A proof-of-concept implementation, demonstrated with an adaptive cruise control system, illustrates the practical application and effectiveness of this method. Future work will explore the preservation of timing behavior when retrofitting safety mechanisms, potentially extending the applicability of this approach to further use cases.

Keywords: Model-Driven Engineering · Code Generation · Safety Software · Embedded Systems

1 Introduction

Functionally safe software is designed and implemented to operate correctly and reliably, especially in critical or hazardous situations where failures could result in harm, injury, or damage. It aims to minimize the risk of failures and ensures that the software behaves predictably, even in the presence of faults or errors. Functionally safe software is the combination of functional software and safety software.

The development of functionally safe software is in general a complex endeavour. Functional requirements and safety requirements can be contradictory to each other and require a complex, holistic analysis. The same holds true on a software level. Modifications to the functional code can impact safety mechanisms and vice versa. Changes must be carefully analyzed to ensure they don't compromise safety requirements or introduce new risks. A systematic separation between functional and safety software could mitigate the complexity, increase the maintainability and the likelihood of reuse.

In order to develop software for safety-relevant systems, it is essential to adhere to international standards, such as IEC 61508[10] for general industrial applications, ISO 26262[11] for automotive systems, or DO-178C[7] for avionics software. These standards dictate a strictly systematic development process and require the implementation of certain software safety mechanisms to minimize risk. This results in a fixed set of safety mechanisms used in developing safety-relevant systems aiming for certification. Depending on the required risk minimization, more or fewer mechanisms from this set are relevant. However, because safety software and functional software are closely integrated, reuse is uncommon, and safety mechanisms are manually reimplemented for each project. This presents an untapped potential for automation, which could lead to significant cost savings.

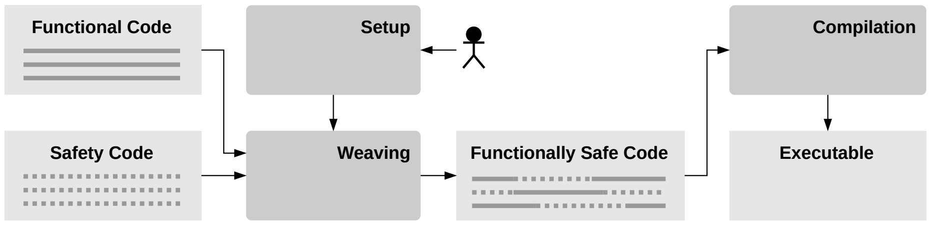

Fig. 1: Fundamental Idea of the Approach

In this paper we present a semi-automated process for integrating software safety mechanisms for safety-relevant systems. The basic idea is depicted in Figure 1. Functional source code and safety source code are initially considered separately. Both aspects of the source code are then combined in a semi-automated weaving step, leading to a source-to-source transformation, which produces functionally safe source code that can then be compiled as usual. Expert knowledge of a safety engineer will be incorporated into the weaving step via a setup option. The proposed approach enables the efficient and therefore more cost-effective integration of safety software. In addition, the systematic and automated procedures allow for easy and quick evaluation of different safety configurations, while reducing the likelihood for human error and improving the overall quality of the code base. Moreover, the presented approach can be used for retrofitting software safety mechanisms to existing systems since it works on already existing source code.

The structure of this paper is as follows: First, the relevant context for the problem and the selected software safety mechanisms are described in Section 2. The approach section (Section 3) describes the process and the utilized models. A proof-of-concept implementation of the approach in C++ is then briefly summarized in Section 4 using the example of an adaptive cruise controller. The second to last Section 5 discusses related research work. Finally, the paper concludes by summarizing the results and offering suggestions for future research and improvements.

2 Background

The background chapter is divided into two sections. First, it describes how functionally safe software is usually developed. The second section presents an overview of software safety mechanisms required by common standards.

2.1 State-of-the-Art Safety Engineering

Numerous safety standards exist to reduce the likelihood of safety-relevant system failures. These standards offer guidelines for designing, implementing, and maintaining such systems. The subsequent sections will provide an overview of the IEC 61508 standard.

The fundamental principle of IEC 61508 requires that any safety-related system should either function correctly or fail predictably and safely under all possible stated conditions. The standard outlines a thorough engineering process known as the safety life cycle, consisting of 16 phases to achieve this objective. Beginning with analysis, progressing through principles for realization, and concluding with stages related to system operation.

An essential aspect of this life cycle is a hazard and risk analysis, involving a probabilistic failure approach to categorize the safety implications of a component's failure. It consists of three key stages: hazard identification, analysis, and risk assessment. For the risk assessment, risk is considered as a function of the likelihood of a hazardous event and the severity of its consequences. The assessment can be done either with qualitative or quantitative analysis techniques. This evaluation helps identify risks that need mitigation, enabling the design of appropriate safety software and thereby reduces the likelihood of under- or overuse of software safety mechanisms. The required risk reduction is then translated into a target safety integrity level (SIL). SILs are discrete levels (ranging from SIL 1 to SIL 4) that represent the relative levels of risk-reduction provided by a safety function. The underlying rationale for SILs is hereby as follows: to achieve a higher risk reduction, the safety-related system must have a higher reliability, which requires a correspondingly higher target SIL.

IEC 61508 and other relevant safety standards provide guidance on the selection of software safety mechanisms to achieve a specified SIL. As a result, there exists a recurrent set of software safety mechanisms frequently used in the development of safety-related systems. An overview of those software safety mechanisms is given in the next section.

2.2 Software Safety Mechanisms Commonly Used

To gather an understanding of the mechanisms that a semi-automated approach for integrating software safety mechanisms would have to support, we first reviewed relevant international standards. The results of this research can be seen in the table below[1]. Relevant positions of the standards are referenced. The compatibility of those mechanisms with the presented approach will be discussed in the next paragraphs of this paper.

Table 1: Software Safety Mechanisms in International Standards.

Name/Description	Standards
Error detection codes	IEC 61508–3 (C.3.2)
	ISO 26262–6 (Table 4/5)
	ISO 26262–10 (Table A.5)
Watchdog mechanism	ISO 26262–6 (Table 4)
Range checks for input and output data	ISO 26262–6 (Table 4)
Plausibility check	ISO 26262–6 (Table 4)
Detection of data errors	ISO 26262–6 (Table 4)
External monitoring facility	IEC 61508–3 (Table A.2)
	ISO 26262–6 (Table 4)
Majority voter	IEC 61508–7 (A.1.4)
	ISO 26262–5 (Table D.2)
Control flow monitoring	ISO 26262–6 (Table 4)
Static recovery mechanism	ISO 26262–6 (Table 5)
Self-test by software	IEC 61508–2 (A.3.2)
	ISO 26262–5 (D.2.3.3)
Graceful degradation	IEC 61508–3 (C.3.8)
	ISO 26262–6 (Table 5)
Independent parallel redundancy	ISO 26262–6 (Table 5)

For the proof-of-concept implementation of this work, we implemented three custom software safety mechanisms that cover a larger part of Table 1.

Contracts. This software safety mechanism is based on the Design by Contract programming paradigm [13]. It can be used to perform checks on the input and output variables of function calls in the form of assumptions and guarantees. If the calling party fulfills the assumption of the contract, the function itself is obligated to meet the guarantee. When the guarantee isn't satisfied, the function itself is to blame. If the assumption is not met, it is the fault of the calling party. This approach of using contracts for runtime monitoring in C++ was already published by us in [14]. Contracts can be utilized to perform range checks of input and output data or plausibility checks as requested by safety standards (see Table 1).

Dual Modular Redundancy. The dual modular redundancy (DMR) mechanism can be applied to function calls. If applied, the function is executed redundantly. The results of both executions are then passed to redundant voter components. If the results match, there is no error. If there is a discrepancy, an appropriate error handling will be initiated. The DMR mechanism can be

[1] Result of the SAFE4I project (01IS17032L)

used for the detection of data errors, having a majority voter or as an external monitoring facility.

Time Measurement and Control Blocks. In addition, a software safety mechanism for analyzing, altering or monitoring the timing behavior of an application has been implemented. It covers the watchdog functionality listed in Table 1. The implementation is based on the work of Bruns et al. [6]. The mechanism is used for analysis purposes by measuring the execution time of a specific program section. In addition, the mechanism can also ensure that a specified execution time is not exceeded or it even allows to enforce a desired execution time by forcing a program section to consume all of its specified time. This can be helpful, e.g., if the environment expects a certain temporal behavior from the application.

3 Approach

In this section we describe our proposed approach for the semi-automated integration of software safety mechanisms. The overall process of the approach is depicted in Figure 2. The actual integration of software safety mechanisms happens as a source-to-source transformation where existing functional source code is systematically extended by calls to software safety mechanism libraries. Possible integration points for mechanisms are found automatically by analyzing the functional source code.

3.1 Prerequisites

In order for this source-to-source transformation to take place, the following preconditions must be met. For one, the functional source code must be available for analysis and rewriting. Furthermore, software safety mechanisms (SSM) must be implemented in a specific library-based manner and be accompanied by a model representation, called `SSM Model`. This representation contains, among other things, necessary information for the actual code changes during integration and allows for formulating requirements of the software safety mechanism to the hardware/software environment. In order to check requirements automatically during the integration process, the target platform has to be modelled accordingly. The modelling process is called `Target2Model` and the resulting model is labeled as `Target Platform Model`. Both can be seen in the upper right part of Figure 2. They are depicted somewhat transparently since there is no direct contribution to this part described in this paper.

3.2 Process

The process starts with automatically analyzing the functional source code to find all possible integration points for software safety mechanisms. The result of this analysis is the `Application Model`. The analysis itself is called `Code2Model` and is numbered as 1 in Figure 2. A detailed description of the model and its automatic generation is provided in Subsection 3.3. Within a graphical user interface, the safety engineer is now able to map available and compatible software safety mechanisms to integration points. The `SSM Model` of the mechanism defines what type of integration points are viable options. In addition, requirements that the mechanisms have of the target platform will be checked automatically against the `Target Platform Model`. Also, if the general functionality of the software safety mechanism allows it, parts of the mechanism can be mapped to hardware/software resources via configuration. The mapping of a `SSM Model` to the `Application`- and the `Target Platform Model` is referred to as `Weaving` while the required manual configuration by a safety engineer is labeled as `Setup` in Figure 2. The result of the mapping is labeled as `Final Model`. It contains all necessary information to rewrite and thereby safeguard the functional source code.

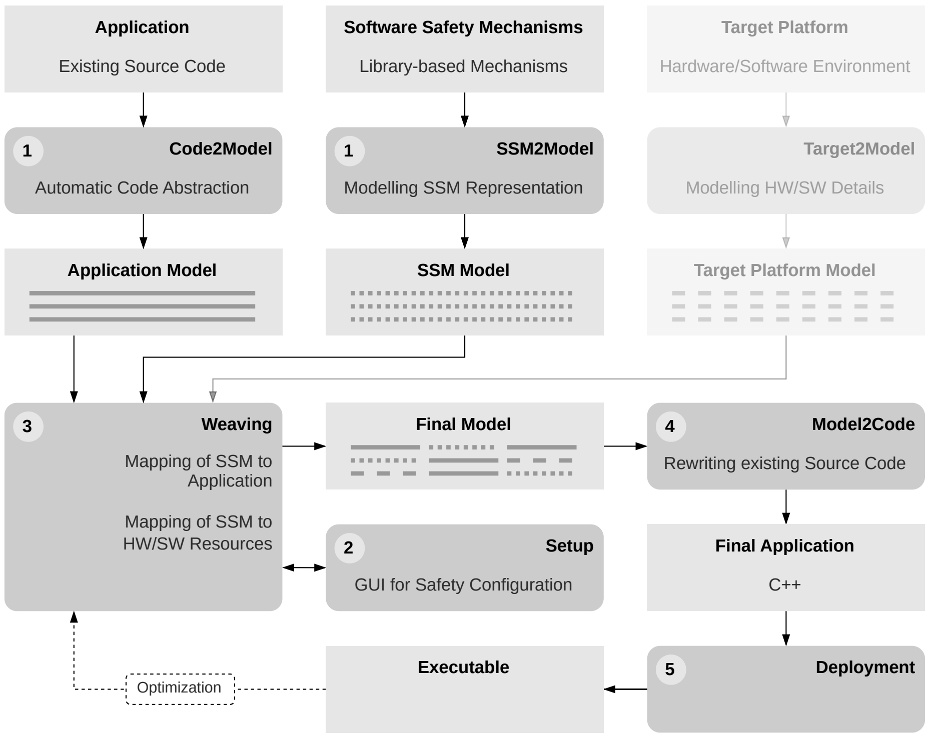

Fig. 2: Overview of the Approach

The source code rewriting is called `Model2Code` in the overview figure. Based on the previously generated model, calls to software safety mechanism libraries are written to integration points in the functional source code. Depending on the implementation of the mechanism, possible dependencies in the linking step may need to be resolved. Otherwise, the deployment will happen as usual.

3.3 Application Model

The `Application Model` serves to hold all information about the functional source code required by the integration process. We determined the necessary information on the basis of the previously identified software safety mechanisms found in standards, listed in Table 1. Furthermore, the `Application Model` is used for the visual abstraction of the functional source code, which the safety engineer uses to determine where software safety mechanisms should be integrated. For this reason, the control flow is part of the model since it supports the decision-making, for example, as to what parts of the application should potentially be executed redundantly or where a watchdog mechanism should sensibly enforce timing constraints.

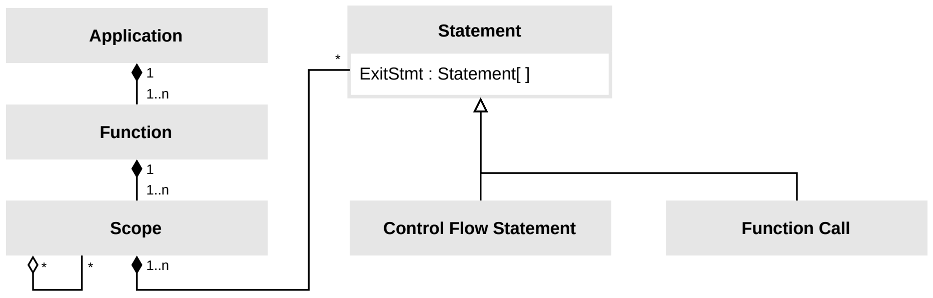

Fig. 3: Application Meta Model

Figure 3 shows the meta model of the `Application Model`. An abstraction of the functional source code happens function-wise. In functions there are scopes, which in turn can contain further scopes. Inside those scopes can be function calls or control flow changing statements. Both have a common parent class, that holds an adjacency list to store the actual control flow between statements. As of now, two ways of integration points are supported. Mechanisms can be applied to scopes and/or to function calls.

Automatic Code Abstraction. The automatic abstraction of the functional source code happens in the `Code2Model` step of Figure 2. At first, the abstract syntax tree (AST) of the code is generated. Integration points are then extracted from the AST with the help of AST matchers. In order to gather the control flow between the integration points, a source-level, intra-procedural control flow graph is generated. Combined, these aspects form the `Application Model`.

4 Demonstration

Proof-of-Concept Implementation. We carried out a proof-of-concept implementation to evaluate the concepts developed within this work. Tools for abstracting and rewriting the functional source code were created on the basis of Clang LibTooling [5]. The AST Matching was realized with [2], the source-level control flow graph was created with [3]. Also, we created a user interface as a Visual Studio Code extension. The example `Application Models` in the images below are actually screenshots from this extension. The source code rewriting was realized with [4].

Adaptive Cruise Control Example. The proposed method is demonstrated using source code excerpts from a simplified adaptive cruise control (ACC) system. In particular, the function for updating the speed is considered. For the demonstration, this chapter first describes a functional source code snippet of the ACC and then shows the visual representation of the corresponding `Application Model`. The actual changes in the C++ source code caused by the integration of software safety mechanisms are described later.

The left side of Figure 4 shows the `Update` function. First, the control deviation for the PID controller is calculated, which is then passed to the `PID_calculate` function. The result of this function is the speed adjustment. A conditional statement checks whether the new speed value would fall below a threshold value. If so, the function `DisableACC` would deactivate the system. Otherwise, the `SetSpeed` function passes on the new speed value to the actuating system.

```
// Update speed based on distance using PID
void Update(double target, double curr_speed)
{
  double speed_err = target - curr_speed;
  double speed_adj = PID_calculate(speed_err);

  curr_speed += speed_adj;

  // Ensure speed is within safe limits
  if (curr_speed < 30.0)
  {
    DisableACC();
  }
  else{
    SetSpeed(curr_speed);
  }
}
```

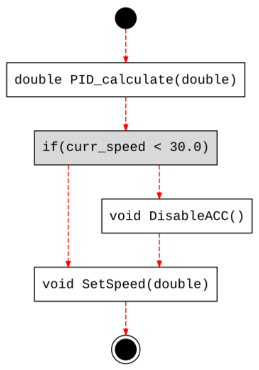

Fig. 4: Functional Source Code and the Corresponding Visual Representation of the `Application Model`

The right side of Figure 4 shows the visual representation of the `Application Model`. The white nodes are integrations points. The grey nodes represent control flow changing statements. The extensions user could select an integration point, whereupon the software safety mechanisms catalog is presented. If an edge is selected, all edges of the directly associated scope are highlighted, then the user can select compatible mechanisms.

The example safety setup will be as follows: The `PID_calculate` function will be safeguarded with a DMR mechanism, while a Time Measurement and Control Block will ensure that the maximum execution time of `DisableACC` is not exceeded. Finally, a Contract safety mechanism will be utilized to perform plausibility checks on the `SetSpeed` function.

The result of the automatic integration can be seen in Listing 1.1 and Figure 5. Applying mechanisms to non member function calls is very straightforward. To do this, the original function call is simply replaced with a wrapper function. The wrapper function is defined in an additional header file (`rg_config.hpp`) which has to be included. It executes the software safety mechanism and the safeguarded function. This way, the perceived changes to the functional code will be kept to a minimum. The wrapper function has the original name, extended by a random suffix, as seen in line 6, 13 and 16. The source code generation for the wrapper function, the rewriting of the original function call and the include of the additional header file all happen automatically.

Applying mechanisms to member function calls is also possible, though it is somewhat more complex. It requires passing the member function call as a function object to the wrapper function using a lambda expression. After that, the procedure is the same.

```
 0 #include "rg_config.hpp"
 1 
 2 // Update speed based on distance using PID
 3 void Update(double target, double curr_speed)
 4 {
 5   double speed_err = target - curr_speed;
 6   double speed_adj = PID_calculate_RG_1(speed_err);
 7 
 8   curr_speed += speed_adj;
 9 
10   // Ensure speed is within safe limits
11   if (current_speed < 30.0)
12   {
13     DisableACC_RG_2();
14   }
15   else{
16     SetSpeed_RG_3(curr_speed);
17   }
18 }
```

Listing 1.1: Functional code after `Weaving`

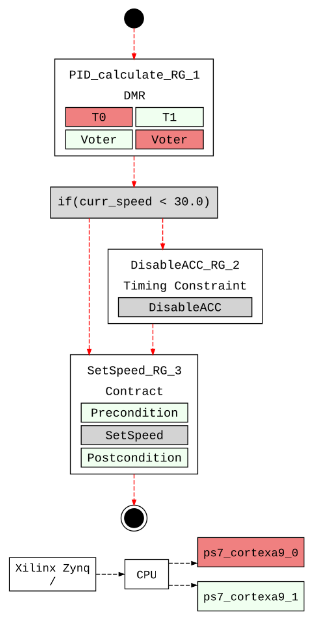

Fig. 5: Application and hardware model with applied mechanisms

Listing 1.2 shows the wrapper functions. If a software safety mechanism is used, the necessary library includes (line 0 to 2) and function definitions (line 4 to 6) are added to the header file. Also, the boiler plate code for the respective mechanisms are automatically generated and added. In the case of the DMR mechanism (`PID_calculate_RG_1`), source code lines 8 to 18 were generated. Lines 13 to 16 define the mapping to the hardware. Default values are initially entered here, which must be configured accordingly by a safety engineer. Figure 5 shows how the tooling visually indicates the configured mapping through color coding.

`DisableACC_RG_2` shows the application of the Time Measurement and Control Block. In line 24 the constructor of `MaxExecutionTime` is called and the `met` object is created. This starts a monitoring thread that ensures a maximum execution time. The wrapper function for the Contract mechanism had to be omitted due to the limited page length.

```
 0 #include <dmr.hpp>
 1 #include <timing_analysis.hpp>
 2 #include <scontract.hpp>
 3
 4 double PID_calculate(double error);
 5 void DisableACC();
 6 void SetSpeed(double speed);
 7
 8 // RG 1
 9 // ------------------------------------------------------------
10 inline double PID_calculate_RG_1(double error)
11 {
12   auto d = DTMR(f);
13   d.MapT0To(CPU::ps7_cortexa9_0);
14   d.MapT1To(CPU::ps7_cortexa9_1);
15   d.MapVoter0To(CPU::ps7_cortexa9_1);
16   d.MapVoter1To(CPU::ps7_cortexa9_0);
17   return d.Execute(error);
18 }
19
20 // RG 2
21 // ------------------------------------------------------------
22 void DisableACC_RG_2()
23 {
24   auto met = MaxExecutionTime(2, std::chrono::milliseconds(10));
25   DisableACC();
26 }
```

Listing 1.2: Function wrapper for the DMR mechansim in `rg_config.hpp`

5 Related Work

This section first examines related work in general before focusing on aspect-oriented programming and respective approaches.

One related approach that generally aims to separate the concerns safety and functionality is the Universal Safety Format (USF) [8] [9]. USF supports a model-driven development approach to automatically integrate software safety mechanisms into functional software. USF provides a domain-agnostic metamodel to describe the functional software as well as a transformation language, the USF Transformation Language (UTL), which is capable of incorporating mechanisms at model level. To bridge the gap between the domain-agnostic USF/UTL and the obviously domain-specific target system, appropriate tooling is necessary. The approach presented in this paper should be seen as complementary and compatible with USF since the tooling described in this paper could be used to apply the domain-agnostic methods of USF to a domain-specific purpose. The main focus of USF lies on the metamodel and the respective model transformations and not on the automated abstraction of existing source code or on the code rewriting/code generation process. Therefore, these two approaches complement each other.

Aspect-oriented programming (AOP) [12] is a programming paradigm that follows similar objectives to the approach presented here. It allows developers to modularize cross-cutting concerns, such as logging or error handling, which would otherwise span multiple modules. AOP separates these concerns from the main codebase and promises cleaner source code. Although AOP has several advantages, it has not yet established itself in hardware-oriented programming in C/C++, despite efforts to do so [1]. An exhaustive consideration of the applicability of AOP to safety-relevant systems is beyond the scope of this paper. Still, we will briefly discuss the relation to the ideas of this work. The approach presented in this paper is highly related to AOP. However, the complexity is reduced compared to AOP due to the limited integration points for mechanisms, though this also weakens the general ability to integrate source code into existing code bases. For the development of safety-relevant systems, the presented approach offers a better balance between complexity and effectiveness. The key difference from AOP is the combined, holistic view of functional source code, software safety mechanisms, and the targeted hardware/software environment.

One approach that tries to address the matter of cross-cutting concerns in embedded systems with the help of AOP can be found in [15]. Wehrmeister et al. present a model-driven engineering approach that combines the unified modeling language (UML) and AOP to improve encapsulation of concerns and speed up the development process. The authors' concept starts with a high-level system specification in UML, which is extended by additional diagrams to allow special modeling for AOP. In addition, this work uses a predefined set of aspects, which includes reusable model and source code elements for extra-functional requirements. During the modeling phase, these aspects can be applied to parts of the functional model. Afterward, a script-based generation tool will generate platform-specific source code. Model integration of aspects is not a part of this approach. Moreover, the development process of this approach begins with a modeling phase. The approach presented in this paper can be applied to existing code bases.

6 Conclusion and Future Work

In this paper, we presented a model-based approach that utilizes a strict separation between functional and safety software to automatically integrate software safety mechanisms into functional source code. A proof-of-concept implementation showcased the automatic analysis of functional source code for finding possible integration points and demonstrated the source code rewriting.

It should be noted that the procedure currently supports two types of integration points — both the automatic functional code analysis and the mechanisms implementations are tailored to this. This is, therefore, a limitation of the approach presented. However, this can be mitigated as follows: We consider the mechanisms required by the standards to be generally compatible. Section 2.2 already pointed out that the mechanisms of this paper already cover a significant part of Table 1. On the other hand, certain mechanisms will be too application-specific to benefit from a library-based approach. This applies, for instance, to the self-test, the static recovery, or the graceful degradation mechanism. Further experiments shall be carried out on this matter.

In addition, alternatives to the abstraction of functional software are to be studied in the future. A query-like approach, similar to that known from AOP, may be a beneficial extension to the approach. We expect that the presented approach will prove to be particularly useful in two use cases: the preservation of timing behavior of an application when retrofitting software safety mechanisms and the creation of software tests for existing applications. Both will be investigated in further studies. Moreover, a future evaluation will examine the functional integrity after weaving, the effectiveness of the software safety mechanisms, their resource usage, and the overall applicability of the approach.

Acknowledgments. This work was partially funded by the German Ministry of Education and Research (BMBF) (grant number 01IS17032L).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aspectc++, https://www.aspectc.org/
2. Ast-matcher, https://clang.llvm.org/docs/LibASTMatchersReference.html
3. Clang cfg, https://clang.llvm.org/doxygen/classclang_1_1CFG.html
4. clang::rewriter, https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html
5. Libtooling, https://clang.llvm.org/docs/LibTooling.html
6. Bruns, F., Yarza, I., Ittershagen, P., Grüttner, K.: Time measurement and control blocks for bare-metal c++ applications. ACM Transactions on Embedded Computing Systems **20** (6 2021). https://doi.org/10.1145/3434401
7. DO-178C: Software considerations in airborne systems and equipment certification. Standard, RTCA, Inc. and EUROCAE (2012)
8. Haxel, F., Viehl, A., Benkel, M., Beyreuther, B., Birken, K., Schmedes, R., Gruttner, K., Mueller-Gritschneder, D.: Universal safety format: Automated safety software generation. International Conference on Model-Driven Engineering and Software Development (2022). https://doi.org/10.5220/0010784200003119
9. Haxel, F., et al.: The universal safety format in action: Tool integration and practical application. SN Computer Science **4** (2023). https://doi.org/10.1007/S42979-022-01532-Z
10. IEC61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (e/e/pe, or e/e/pes). Standard, The International Electrotechnical Commission, Geneva, CH (2010)
11. ISO26262: Road vehicles - functional safety. Standard, International Organization for Standardization, Geneva, CH (2018)
12. Kiczales, G., et al.: Aspect-oriented programming. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **1241**, 220–242 (1997). https://doi.org/10.1007/BFB0053381
13. Meyer, B.: Applying "design by contract". Computer **25**, 40–51 (1992). https://doi.org/10.1109/2.161279
14. Schmedes, R., Ittershagen, P., Grüttner, K.: Towards distributed runtime monitoring with c++ contracts (2019). https://doi.org/10.1145/3312614.3312645
15. Wehrmeister, M.A., et al.: Aspect-oriented model-driven engineering for embedded systems applied to automation systems. IEEE Transactions on Industrial Informatics **9**, 2373–2386 (2013). https://doi.org/10.1109/TII.2013.2240308