Using Generic Software Components for Safety-Critical Embedded Systems – An Engineering Framework

Felix Bräunling[*], Robert Hilbrich[†], Simon Wegener[‡], Isabella Stilkerich[§] and Daniel Kästner[‡]

Abstract

Modern software development in the automotive domain would be unthinkable without leveraging reusable software components. Such generic software components have to be configured and tailored for each specific target application. Nowadays, complexity has reached a point where developing generic software components and manually adapting each component for each variant in the product family is error-prone and no longer economically feasible. In this article we propose an engineering framework for automated adaptation of generic software components which focuses on temporal and spatial integrity. The framework is built around a generic methodology and leverages specialized software tools to determine an allocation of software components to the resources of an embedded system and to ensure memory integrity. We use a quadcopter example, executed on the Infineon AURIX™ TC277 processor under the AUTOSAR operating system to illustrate our approach.

1 Introduction

Safety-critical embedded systems represent a special class of computerized control systems. The interplay of their software and hardware parts realizes complex functions, such as engine control or vehicular guidance. Undetected errors in the implementation of a function may jeopardize human lives, hence the additional attribute *safety-critical*. Implementing the necessary software correctly, satisfying all safety requirements *and* maximizing hardware resource utilization in a cost-sensitive and competitive market poses a significant challenge for established software engineering methods and tools.

A recent example for the increasing capabilities of microcontrollers are embedded multicore processors containing multiple and possibly heterogeneous execution units. In order to tap their potential and maximize resource utilization, software components have to be tightly integrated and optimized specifically for each hardware platform. This optimization step is referred to as *adaptation* and often conducted manually.

With more software components sharing common resources of a microcontroller, ensuring their isolation against undesired interferences is essential in order to maintain the reliability and safety of the system. Therefore, the adaptation needs to ensure isolation between tightly integrated software components. This can be achieved by controlling access to both memory and CPU time, thereby supporting error containment.

1.1 Problem Statement

Software engineering for safety-critical embedded systems is currently conducted in a "per-project" fashion. Depending on the setup of the system architecture in different projects, it may be necessary to distribute software components across a network of microcontrollers or to run them on different microcontroller derivatives. Unfortunately, due to the complexity and the lack of proper engineering tools, software components are often developed specifically to match the requirements of a particular project. They are allocated manually to the resources in the system's hardware architecture and adapted manually to make best use of the capabilities of the microcontrollers. Especially for safety-critical systems, ensuring an isolated execution of software components requires additional configuration and analysis steps which are specific to the microcontroller and thus often conducted manually as well.

While the "per-project" approach may be sufficient for a small number of projects, it still requires a lot of manual effort and reduces the *reusability* of software components. With more and more control functions in safety-critical devices being implemented in software, the need for reusability and adaptability for different hardware platforms increases, which renders the "per-project" approach no longer economically sustainable. The question arises, how software components for safety-critical embedded systems can be developed in a *generic* and *reusable* way, so that they can be used in *multiple* projects, but without reducing resource utilization or jeopardizing their isolation properties and memory safety.

[*]Felix Bräunling is with Method Park Engineering Gmbh, Germany felix.braeunling@methodpark.de

[†]Robert Hilbrich is with Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Germany robert.hilbrich@dlr.de

[‡]Simon Wegener and Daniel Kästner are with AbsInt Angewandte Informatik, Germany {swegener, kaestner}@absint.com

[§]Isabella Stilkerich is with Schaeffler Technologies AG, Germany isabella.stilkerich@schaeffler.com

1.2 General Approach

The authors argue that the level of reusability can be significantly improved with a *model-based development* of *generic* software components in combination with an automated *adaptation toolkit* to handle project-specific hardware properties and safety requirements.

In general, software components are concerned with the implementation of specific *features* of the system. By using abstraction layers, they can be developed in a platform agnostic manner and organized as a *library* of generic software components for later use in multiple projects. As a result of adopting a formalized and model-based development approach, these generic functions can be automatically tailored and adapted to the specific requirements of a project. In particular, this approach facilitates automated software deployment in combination with automated platform-specific code generation as well as automated configuration and validation of isolation properties. This "feature-based" software development is not entirely new [3, 5]. However, the authors believe that the state of practice and the capabilities of available tools for this purpose have not yet reached the level of maturity needed for the development of multi-platform safety-critical embedded systems.

1.3 Contribution

In this paper, the authors present the results of the development of an *engineering framework* aiding system architects and software engineers. Software for embedded systems relies on mature development tools in order to cope with complexity and to satisfy all (safety) requirements. Therefore, the framework combines and extends the tools *ASSIST*, *Astrée* and *cAMP*. By providing interoperability between these tools and enhancing them with new functionality, the framework is able to automate the integration and tailoring of applications in safety-critical embedded systems (i.e., *automated adaptation*).

In its current state, the framework provides an early and significantly less error-prone evaluation of timing and memory-partitioning decisions at the system level, the software level and also the implementation level. Combining these tools as a framework allows to automatically adapt and integrate generic software components in order to create project-specific applications running on project-specific microcontrollers. Project-specific topics, such as the properties of a particular microcontroller, as well as spatial and/or temporal isolation requirements are also taken into account. By taking advantage of code generation, the framework supports an automated development process, thus building the missing link between generic libraries of software components and project-specific applications.

2 Conceptual Overview

Figure 1 depicts the simplified workflow of our framework. It is based on the idea of a strictly top-down engineering approach combined with the *correctness by construction* methodology [8] and supports the engineer by automating the synthesis and validation of crucial engineering artifacts.

Based on a model of the *functional architecture*, the systems engineer creates a model of the envisioned *system architecture*. This model contains a selection of generic software components from a library and a project-specific hardware platform. The feasibility of the chosen hardware platform with respect to the technical and safety-related requirements of the particular project can be automatically validated by constructing a deployment for the selected software components (see Section 6). If a valid deployment cannot be found, either the hardware platform or the selection of software components need to be modified. Then, *sound* semantic code analysis is applied to all software code to ensure sufficient isolation and memory protection, both of which are essential to ensure the correctness of the system (see Section 7). Finally, data and code of software components are automatically mapped to the isolation partitions of their microcontroller (see Section 8).

The essential parts of the framework are described in more detail in the next sections. However, this paper is not intended to provide a thorough and detailed description for each of the tools used in the framework. Instead it focuses on their *contribution* for an automated development process based on generic software components.

3 I4Copter

To illustrate our approach, we will use the I4Copter flight controller as an overarching example. The I4Copter is a research project developing a quadcopter as an example for hard real-time systems and control systems [37, 36]. While the quadcopter software was originally meant to be deployed to a single-core Infineon TriCore TC1796, we used the AURIX TC277 for this paper, which is a three-core automotive microcontroller. The control software of the I4Copter consists of six modules: a digital signal processor (DSP) for reading sensor and remote inputs, a two-stage controller, and three observers. All these modules are responsible for controlling the positioning of the quadcopter during take-off, flight and landing. Further details of the I4Copter and the targeted hardware will be given in the following sections to describe their relevance to the presented framework. The example system resembles control systems similar to those used in automotive chassis systems in complexity and size. The AURIX TC277 was selected because it is a widely used microcontroller in the automotive domain. It also provides heterogeneous memories and computation units which allow for a flexible and adaptive deployment of software based on

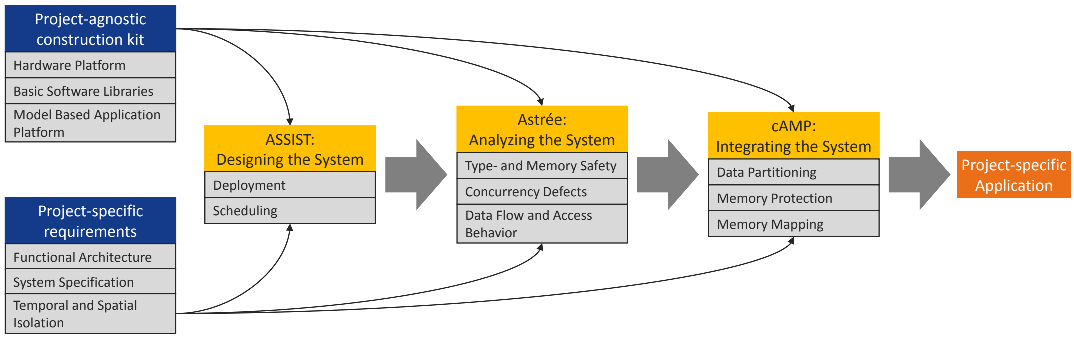

Figure 1: Simplified Workflow

the product's requirements, which makes it a suitable target platform for a *software platform driven development* approach.

4 Platform Development

In order to develop application software components in a reusable and structured way, the state of practice in systems engineering [20] recommends to start with a *functional architecture*. It is constructed as a result of a rigorous requirements engineering process and its relationship to other system views [27].

The functional architecture is tightly bound to the *logical* intentions and dependencies of the software applications. It is a special kind of abstraction and usually, it does not include details of the *technical architecture*.

The separation into a functional and a technical architecture allows for a *separation of concerns* between the problem domain and the variations of possible technical solutions. This approach allows to construct applications and infrastructure software independently, because implementations on both ends can be developed separately as long as they comply with the "contract" of the common interface.

In our case, this can be described by using the *bridge pattern*, which is depicted in Figure 2. The bridge pattern is intended to decouple the abstraction from its implementation [12]. A *functional element* describes an abstract function. A *technical implementation* on the other hand is a tangible implementation providing the functionality described by the functional element. The functional element does not need to know about the tangible implementation, thus the technical implementation can be replaced by other solutions as long as all project-specific requirements are met. Still, the functional and technical view are not fully independent of each other, because changes in either view may have significant effects on the other.

One example for a functional element in the I4Copter software is the filter used to reduce noise from sensor inputs. A technical implementation to provide the functionality could be an alpha-beta-filter, a running average filter or a Kalman filter. Their software modules must provide a common interface to make the filter implementations exchangeable. The choice of the filtering algorithm can be controlled by changing which module is linked during compile time into the application code.

In each project, it is the task of the systems engineer to select the suitable building blocks for each functional element, i.e., generic software components from a library as well as hardware components (microcontrollers), in order to create the *system architecture* model. Of course, the model needs to fulfill the requirements set forth in the *functional architecture*.

5 System Architecture

At the core of our approach is a system architecture model. There are several notations available to model a system architecture, for example SysML, UML, or AMALTHEA [15], which allows to express safety- and timing-related requirements as well-defined first-class model elements.

Generally speaking, the system architecture model comprises of a description of the *software* components with their distinct resource requirements, a description of the available *hardware* resources, such as microcontrollers or memory, and also *constraints*, in particular safety requirements, that restrict the resource usage and allocation of software components.

Examples for safety requirements can be the need to detect sporadic hardware defects in the computation of a software component, heterogeneous hardware execution environments or the use of additional hardware for error correcting/detecting codes. These requirements translate to constraints on the system architecture and thus are limiting the solution space of the architecture.

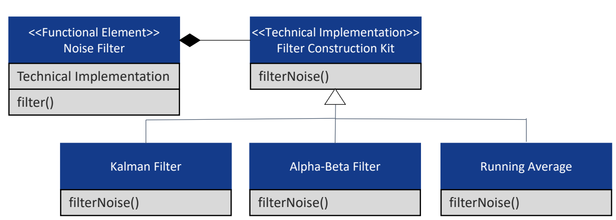

Figure 2: The relation between the functional elements and the technical implementation exhibits the bridge pattern.

6 Deployment

With the system architecture model being available, the mapping between software components and hardware resources is constructed in the next step. The construction needs to consider the capacities of all available resources, e.g., the size of flash memory. It must also ensure that the additional constraints (temporal and spatial isolation) are satisfied. This is achieved by constructing a feasible mapping and a static periodic schedule for all software components. Due to the complexity of the solution space and the importance of the correctness of the solution, this process is conducted in an automated fashion with the tool ASSIST [13]. The challenge of finding a correct mapping and a feasible static schedule is addressed by transformation of this problem into an equivalent *constraint-satisfaction problem* and the subsequent application of *constraint programming* [10, 4, 31]. Similar approaches for safety-critical systems have been published [14, 32].

Constraint programming refers to a set of techniques in operations research, discrete optimization, and artificial intelligence. These techniques assist in finding solutions for problems based on variables, which are affected by constraints (constraint-satisfaction problems). Each variable has a finite integer domain and every constraint defines valid or invalid solutions for a subset of these variables. Solutions for this problem class can be obtained by applying a combination of search techniques—including backtracking—and constraint propagation techniques for value elimination. ASSIST automates this process and hides the intricacies of a formal specification from the user by offering a user-friendly domain specific language to describe the mapping and scheduling problem.

```
Hardware {
  /* ... */
  Processor Processor1 {
    Manufacturer = "Infineon";
    Type = "TC277";
    Provides 32768 of exclusive feature "LMU RAM";
    Provides 4194304 of exclusive feature "PMU Program Flash";
    Core Core0 {
        Capacity = 100;
        Architecture = "TriCore 1.6 P";
        Provides shared feature "Performance";
        Provides shared feature "FPU";
        Provides 16384 of exclusive feature "I-Cache";
        Provides 8192 of exclusive feature "D-Cache";
    }
    Core Core1 {
        /* identical to Core0 */
    }
    Core Core2 {
        Capacity = 60;
        Architecture = "TriCore 1.6 E";
        Provides shared feature "Efficiency";
        Provides shared feature "FPU";
        Provides shared feature "Lockstep";
        Provides 8192 of exclusive feature "I-Cache";
        Provides 128 of exclusive feature "DMI Readbuffer";
    }
  }
}
```

Figure 3: Hardware Specification in ASSIST

In contrast to other approaches [29, 30], which are based on the composition of components with rich interfaces, ASSIST works on a higher abstraction level and treats mapping and scheduling as separate steps in order to reduce the size of the problem in each step. For this purpose, the modeling of component properties in

```
Software {
  Application OS_Application_0 {
    Task T1_Controllers { CoreUtilization = 2; }
  }
  Application OS_Application_1 {
     Task T3_AttitudeObserver { CoreUtilization = 20; }
  }
  Application OS_Application_2 {
     Task T2_EngineController {
            CoreUtilization = 2;
            Requires shared Core feature "Lockstep";
     }
  }
  Application OS_Application_3 {
     Task T4_HeightObserver { CoreUtilization = 20; }
  }
  Application OS_Application_4 {
     Task T6_DSP { CoreUtilization = 3; }
  }
  Application OS_Application_5 {
     Task T5_AltitudeObserver { CoreUtilization = 20; }
  }
}
```

Figure 4: Applications and Tasks in ASSIST

ASSIST is less detailed. Software tasks are considered to be "black boxes" with annotated resource requirements. ASSIST also aims to simplify the complexity of the scheduling problem, by constructing a scheduling for a single hyperperiod containing periodic executions of all tasks. Internally, ASSIST uses the CHOCO SOLVER [28], which has been successfully applied in a wide variety of scheduling problems.

Figure 3 contains the specification of the hardware properties of the Infineon AURIX TC277 microcontroller. ASSIST allows to specify *features*, such as an FPU, as well as *capacities*, for example flash memory, to constrain the deployment process.

The specification of the software architecture is presented in Figure 4. It consists of six applications together with their tasks. For the sake of simplicity and readability of the example, each task only requires a certain amount of the processor time (called `CoreUtilization` in the specification). The core utilization is determined by the task's worst-case execution time (WCET) divided by the task's period. Safe upper bounds on the WCET can be calculated, e.g., by aiT WCET Analyzer [18] for timing-predictable processors such as the AURIX TC277. On non-timing-predictable multi-core processors, WCET estimates can be provided by hybrid WCET analyzers such as TimeWeaver [19]. The task `T2_EngineController` shows how particular features of a processing core can be required by a task.

Figure 5 shows the dependencies between the different tasks which are present for a single period of the cyclic tasks of the I4Copter system. The tasks `T4_HeightObserver` and `T5_AltitudeObserver` could be run in parallel to exploit the resources of the TC277 multicore processor.

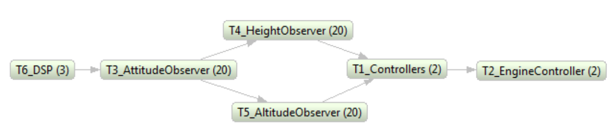

Figure 5: Task Dependency Graph

In order to allow a parallel execution of these two tasks, they should be deployed to separate cores. How-

```
TaskGraph {
    T6_DSP              -> T3_AttitudeObserver;
    T3_AttitudeObserver -> T4_HeightObserver, T5_AltitudeObserver;
    T4_HeightObserver   -> T1_Controllers;
    T5_AltitudeObserver -> T1_Controllers;
    T1_Controllers      -> T2_EngineController;
}
```

Figure 6: Task Graph Specification in ASSIST

```
Restrictions {
    T6_DSP, T3_AttitudeObserver dislocal up to Core;
    T6_DSP, T4_HeightObserver dislocal up to Core;
}
```

Figure 7: Mapping Constraint Specification in ASSIST

ever, this does not constitute a hard constraint for the deployment synthesis, because a deployment of these tasks to the same core may still be feasible, but less desirable. Therefore, ASSIST needs to treat the information about the parallel execution of `T4_HeightObserver` and `T5_AltitudeObserver` as a *hint* for achieving an optimized solution. For this purpose, a task graph can be specified in ASSIST, which allows the tool to automatically determine which tasks could be run in parallel in order to derive the optimization hints for the deployment. The specification of the task graph for the example of this paper is contained in Figure 6.

In contrast to the "soft" optimization hints, there are also "hard" deployment constraints, such as safety-related constraints, which affect the feasibility and validity of the results. These constraints must be satisfied in order to obtain a valid deployment. For the example use case, two additional safety requirements are assumed to ensure reliability despite potentially harsh environmental conditions.

1. The tasks `T6_DSP` and `T3_AttitudeObserver` must not share the same core.

2. The tasks `T6_DSP` and `T4_HeightOberserver` must not share the same core.

Figure 7 shows how these requirements can be expressed as "hard" mapping constraints in ASSIST.

Based on the deployment specification described in the listings above, ASSIST was able to determine *all* valid deployment solutions. There are 208 different solutions, which were computed in about 250 ms on a regular desktop computer. In order to find the "best" solutions among the set of valid solutions, ASSIST allows to apply a set of *metrics* to each deployment. Those metrics allow to compute a *score* for each solution, which reflects the fulfillment of the optimization goals reflected in the metrics. Solutions with the highest scores are therefore assumed to be the "best" solutions.

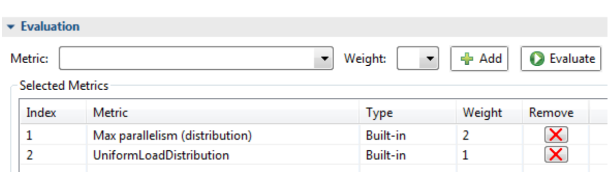

Figure 8: Evaluation of Solutions

For the example use case, two optimization goals for the deployment of the application tasks where pursued. Most important is the achievement of *parallel execution of tasks* by the deployment. Therefore, solutions should be ranked higher, if parallel tasks in the task graph are indeed mapped to different cores. Furthermore, mapping solutions with a *uniform core load* are preferred over solutions with a heterogeneous load. Figure 8 shows the selection of these metrics in ASSIST. Setting the *Weight* of the *Max parallelism* metric to the value of two allows to express the importance of the first goal in comparison to the second optimization goal. The automated evaluation with the aforementioned metrics identified two solutions with the highest score, from which we selected one (see Figure 10).

As a last step, an *AUTOSAR-OS configuration file* is generated by ASSIST, which constitutes a central engineering artifact for the following steps in our engineering framework.

7 Static Analysis of OS Configuration and Source Code

This section addresses the prerequisites for an automated low-level deployment of code and data in order to provide the memory handling and memory protection.

7.1 AUTOSAR-OS System Model

AUTOSAR is a partnership between different automotive manufacturers to design a standard for an embedded automotive operating system. Currently there are two versions of the standard, the AUTOSAR Classic for static systems and AUTOSAR Adaptive for dynamic systems. In this paper we only refer to the AUTOSAR Classic family of operating systems. The AUTOSAR system model is depicted in Figure 9.

The system is structured around *OS-Applications* that are executed on one specific computing unit of the underlying hardware. OS-Applications manages instructions and data in memory as an execution environment for one or more *tasks*. The tasks themselves are schedulable execution units, each with its own data and stack segment. By using hardware-based memory access protection, such as a memory protection unit (MPU), the kernel memory can be spatially isolated from OS-Applications. The same mechanisms also allows for spatial isolation of OS-Applications from other OS-Applications, as well as of tasks inside of OS-Applications. This is indicated by the thick black lines in Figure 9

The static structure of an AUTOSAR system, consisting of the mapping of cores, OS-Applications, tasks and memory protection regions as well as schedules and timer events, is described in a configuration file. The so called ARXML files are written in an AUTOSAR-standardized XML format. Based on this file operating systems implementing the AUTOSAR standard can generate operating system code reflecting the configuration described in the ARXML file. Such an ARXML

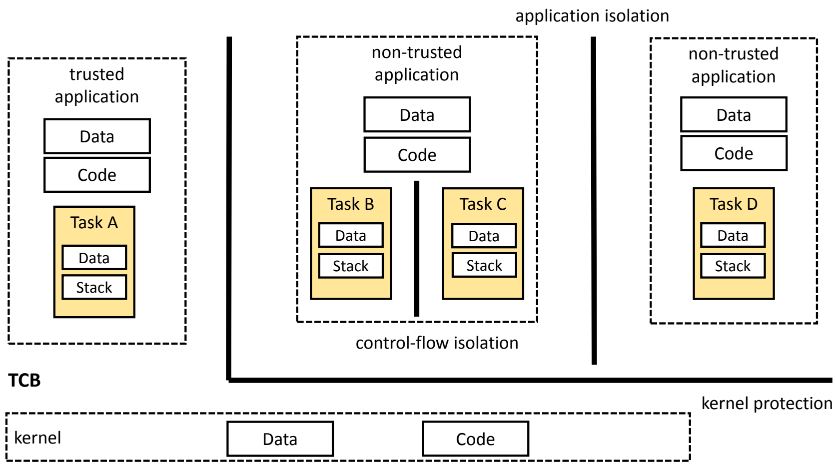

Figure 9: AUTOSAR Isolation Schemes

file is the output of ASSIST, which in turn can then be used by Astrée for a semantic analysis of the operating system and the application code.

7.2 Ensuring Memory and Type Safety

Despite the successful deployment of all software components, memory and type safety need to be addressed in order to ensure spatial isolation between all tightly integrated software components. Memory safety in the scope of the C programming language is defined as the absence of memory accesses that trigger undefined behavior, as well as the absence of data races where shared variables are accessed by concurrent threads without proper synchronization. Although the C programming language itself does not ensure memory safety, a sound static analysis of the C source code is able to guarantee memory safety at the programming language level.

Table 1: Mapping of Requirements of Type- and Memory Safety [2] to Astrée Alarm Types [1].

Requirement	Alarm Type
Operations only applied for instances of correct type	Invalid pointer comparison
	Subtraction of invalid pointers
	Attempt to write to a constant
	Dereference of mis-aligned pointer
	Overflow of Integers or Float
	Invalid shift argument
	Use of uninitialized variables
	Division or modulo by zero
	Undefined integer modulo
	Invalid function calls
	Unsynchronized access to shared data
Access only existing objects	Dereference of null or invalid pointer
	Pointer to invalid or null function
	Use of dangling pointer
	Arithmetics on invalid pointers
	Possible overflow upon dereference
Access only inside object boundaries	Incorrect field dereference
	Out-of-bound array access
	Dereference of mis-aligned pointer
	Possible overflow upon dereference

In our framework we use the Astrée analyzer [17, 24]. Its main purpose is to report program defects caused by unspecified and undefined behaviors according to the C99 standard. The reported code defects include integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (buffer overflows, null pointer dereferencing, dangling pointers, etc.), data races, lock/unlock problems, and deadlocks. To deal with concurrency defects, Astrée implements a low-level concurrent semantics [23] which provides a scalable sound abstraction covering all possible thread interleavings. The analyzer takes task priorities into account and, for multicore systems, the mapping of tasks to applications and cores. Table 1 shows the relation between memory and type safety and the defects found by Astrée. Astrée is widely used in safety-critical systems, and provides the necessary tool qualification support, including Qualification Support Kits and Qualification Software Life Cycle Data reports.

The AUTOSAR-OS configuration file produced by ASSIST during the high-level deployment is parsed by Astrée to automatically generate a matching analysis configuration that models the asynchronous execution of the various tasks and ISRs. Astrée returns a list of potential code defects. An analysis resulting in zero alarms guarantees the absence of memory safety violations in the C source code. Since the C semantics assumes unlimited stack space, the source-level analysis needs to be complemented by a sound static stack-usage analysis at the binary level to prove the absence of stack overflows [16]. Moreover, using a formally verified compiler ensures that no memory safety defects are introduced by miscompilation [21]. Together, these approaches are able to prevent software-induced memory corruption, hence establishing memory safety.

Besides reporting runtime errors and concurrency defects, Astrée also produces detailed data and control flow reports. Soundness provides a guarantee that neither control flow paths nor read or write accesses are missed, even in case of data or function pointer accesses, or task interference. Global data and control flow analysis gives a summary of variable accesses and function invocations throughout program execution.

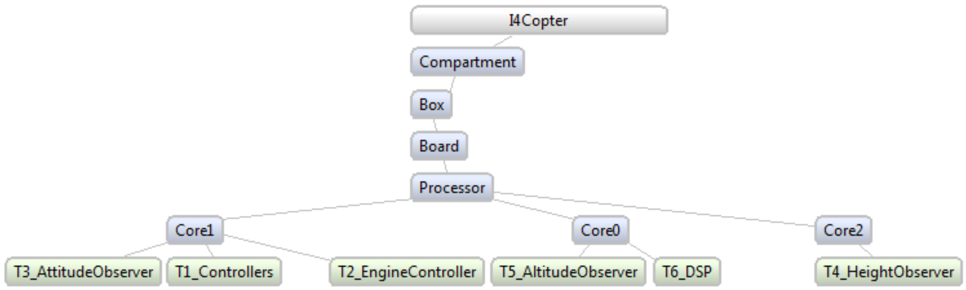

Figure 10: Deployment solution found by ASSIST

Table 2: Excerpt from Astrée Data Flow Report

Variable	Function	Access	Process	Data Races	Shared	Class
In_altCtr_AccZ_g_altObs_noiseVariance	TASK_T1_Controllers	write	T1_Controllers	no	no	process local
In_altCtr_AccZ_g_altObs_processVariance	TASK_T1_Controllers	write	T1_Controllers	no	no	process local
Out_accX_g_dsp_noiseVariance	TASK_T3_AttitudeObserver	read	T3_AttitudeObserver	yes	yes	global
Out_accX_g_dsp_noiseVariance	TASK_T6_DSP	write	T6_DSP	yes	yes	global
Out_accX_g_dsp_processVariance	TASK_T3_AttitudeObserver	read	T3_AttitudeObserver	yes	yes	global
Out_accX_g_dsp_processVariance	TASK_T6_DSP	write	T6_DSP	yes	yes	global
Out_torqueX_NM_attCtr	TASK_T2_EngineController	read	T2_EngineController	yes	yes	core local
Out_torqueX_NM_attCtr	STEP_AttitudeController	write	T1_Controllers	yes	yes	core local
Out_torqueX_NM_engCtr	TASK_T3_AttitudeObserver	read	T3_AttitudeObserver	yes	yes	core local
...	...	...	...	...	...	...

The reports also contain each effectively shared variable, the list of tasks accessing it, the application and the core to which the task has been assigned, and the types of the accesses (read, write, read/write). Indirect variable accesses via pointers as well as function pointer call targets are fully taken into account. Filtering allows determining the control and data flow per software component, thus supporting the analysis of data and control coupling as required by DO-178C. An excerpt from the data flow report for the example system is shown in Table 2. Note that Astrée detects data races for each of the shared variables in our example system, which is expected, because the code does not contain any synchronization mechanisms.

8 Memory Mapping

After the definition of the system, the deployment of all software components, and the analysis for memory safety, the next step constitutes the low-level mapping of all instructions and their data to the physical memories of the specific microcontroller. This step is also called *binding*.

8.1 Heterogeneous Memory

This task is especially important for multicore processors with heterogeneous memories, such as the AURIX TC277, in order to achieve good runtime behavior as well as spatial isolation for freedom from interference. The AURIX TC277 offers six core-coupled SRAMs for data (DSPR) and instructions (PSPR), a non-volatile flash memory (PMU) and a bus-accessed SRAM (LMU). It is possible to access any of the memories from any of the cores using the microcontroller's bus. However, read and write operations to the core-coupled SRAM take only one CPU cycle if they originate from the coupled core, whereas memory accesses via the bus take more time. Moreover, accesses via the bus have less deterministic access times because of interference due to concurrent bus accesses. The flash memory is also equipped with error correction capabilities to ensure data integrity.

8.2 Data and Function Classes

At the same time, data and functions exhibit traits such as origin of access, type of access, frequency of access, and logical traits such as being constant or used for in-system calibration. The combination of these properties favors the binding of each data item or function to a different memory of the microcontroller. Data items and functions that exhibit similar traits can be grouped to *variable and function classes*, each with a set of preferred memories dictated by a binding policy. In the I4Copter example two orthogonal types of classification exists: *task*-wise and *core*-wise. *Task-local* data and functions are only accessed by a single task whereas *task-global* data is accessed by two or more tasks. Analogous to this, *core-local* data and functions are only accessed from a single core, while *core-global* data and functions are shared between different cores. *Constant* data items are grouped into different classes, as their read-only property makes them suitable for binding them to the flash memory and relying on core-coupled cache memories to reduce access times.

8.3 Binding Policy

Binding policies describe to which memory a particular data item or functions should be mapped, depending on both the properties exhibited by the available memories as well as the variable and function classes.

An example for such a policy is the mapping of task-local, core-local data to the core-coupled SRAM to achieve the fastest access time. Another policy is that task-global, core-global data is mapped to the core-coupled memory of the core from which most accesses are originating. As the capacity of these memories is finite, it needs to be decided which data and functions are bound first. This can be done by optimizing policies, ranging from a very simple one based on the total access frequency, using different weights for read and write operations, to complex optimization algorithms using either constraint solving problems or integer linear programming.

In the I4Copter example presented in this paper, data items are sorted by the total amount of reads and writes to these data items. The access frequencies are determined by taking the amount of read and writes reported by Astrée multiplied by the amount of task activations during one hyperperiod (9 ms) of the system. Data items with a higher count of total accesses are mapped first.

8.4 Automated Memory Mapping

Gathering the information about data and function traits, combining it into classes and then creating a mapping to hardware memories is not feasible if done manually in systems of arbitrary complexity. The cAMP tool addresses this problem by automatically generating a mapping of instructions and data to memories. For this, all available information from the previous development steps is used: deployment information from ASSIST by taking into account the deployment solution described in the generated ARXML file. Information about data and functions traits can be extracted from Astrée by taking into account the data flow report (see Table 2). Finally, also the requirements from the system specification such as task times, safety requirements and constraints are taken into account.

The information is aggregated within cAMP and used to generate a binding of data and functions to memories, as well as assigning MPU-enforced memory protection regions to each used section of memory. The binding is performed by following the binding policies for the selected hardware. The result of this process is a linker script that describes the memory mapping and exposes the sections needed to configure the system's MPU.

To reduce the amount of effort necessary to use the linker script, i.e., the assignment of individual variables and functions to the various memory sections, cAMP is able to interface with code generation tools to automatically include the necessary annotations in the generated code. In its current form, cAMP is able to interface with TargetLink, a widely used C code generator for MatLab/Simulink.

The use of cAMP allows the automatic tailoring of applications to specific hardware and memory layouts by creating a reproducible binding process. Using code generation and platform-based development adds further benefits, such as automatic code adaption and reusable binding policies. The automatic binding of data and code results in a significant reduction of software development time, while being less error-prone than manual binding.

9 Related Work

The basic idea for our framework is inspired by KESO and program families presented by Parnas [26]. Parnas was one of the first to give thought to program families and software-product lines (SPL). He described the problem in the context of operating systems. Building on his ideas, later approaches examined the variability challenge in large software systems. For instance, Sincero et al. [34] investigated the Linux kernel and treated it as an SPL through configuration analyses. Liebig et al. [22] looked at configurable software composed in C and explored ways to deal with the complexity induced through preprocessor directives. There are also solutions (e.g., [11, 33]), which use generative programming [9] to create program variants from configurability models. Another project that leverages the idea of program families is an operating-system construction kit called PURE [6]. The authors define a base set of reusable OS-infrastructure components (e.g., threads, scheduling, concurrency, interruptions or memory service) needed to build infrastructure services. For instance, light-weight threads that can be used to compose address spaces and processes. The authors employ a configuration- and code-generation-based framework to create operating-system variants. Different from these prior works, we allow to manipulate spatial and temporal isolation properties in an early design phase based on architecture- and code-analyzing techniques of reusable application parts that can also be developed using model-based techniques.

The KESO Java Virtual Machine [35] provides an isolation concept that is similar to the process concept found in general-purpose operating systems. KESO features a compiler, which is able to produce a virtual machine environment specialized for a particular application. Therefore, KESO adopts ideas from Parnas' approach. In the following, we describe the characteristics of KESO that are relevant for our work. *Spatial isolation* ensures that control flows are only able to access memory of data regions belonging to the protection zone (called *domain*) in the context of which the control flow is being executed. Therefore, each piece of data can be logically assigned to exactly one domain. In Java, type safety ensures that programs can only access memory regions to which they were given an explicit reference; the type of the reference also determines in which way a program can access the memory region pointed to by the reference. To achieve spatial isolation, the KESO compiler enforces that a reference value is never present in more than a single domain. Different from KESO, our applications are not developed in Java but using a model-based technique in which the generated C code is analyzed using abstract interpretation in order to create memory-safe C code. We extended Astrée to use information on the AUTOSAR OS threading model to— amongst other things— be able to perform a flow- and context-sensitive analysis based on OS-Applications to build logical isolation zones that are enforced by memory protection hardware.

The article [25] addresses problems similar to those addressed by our work, but assumes a different communication model. In contrast to AUTOSAR, which uses shared memory for communication between task, their communication model is based on Kahn Process Networks, and the used operating system is based on a non-shared memory model. For spatial isolation, they rely solely on the use of the MPU to ensure dynamic fault containment. We, in contrast, use sound static analysis of the integrated source code to foster safety by construction and utilize the MPU as a safety net. Moreover, the high-level deployment is not part of their workflow but assumed as an input.

10 Conclusion

The authors present the results of a joint research and development project towards an engineering framework for using generic software components in safety-critical embedded systems. The framework combines several tools, so that generic software components can be *adapted* to a particular microcontroller and *analyzed* for memory integrity. Common starting point is a model of the system architecture comprising of a system specification and a functional architecture. Additional information about the target hardware (microcontroller family) and a library of generic software components is considered to be available as well. These specifications are passed through different engineering tools and code analyzers via common exchange formats.

In particular, the tool suite ASSIST allows to generate software allocations for the targeted hardware. By including timing constraints, it also allows to construct a static schedule, thus ensuring the feasibility of the system design. This results in an operating system configuration describing the spatial and temporal behavior of all software components.

In combination with the generated application software, the C sources are examined with Astrée. By removing memory and type defects found during the analysis, memory integrity can be ensured at the C code level. Furthermore, information about data and function access behavior is collected during the analysis step. In a final step, this information is used by cAMP to perform the low-level mapping of data and functions to physical memories and protection regions.

The final result comprises of annotated application code and a linker script describing the memory mapping, which is guaranteed to satisfy all project-specific safety requirements and to utilize the capabilities of the microcontroller. The final results as well as the intermediate work products can be found on GitHub [7]. Using this framework allows to adapt generic application software for safety-critical systems in an efficient and automated manner, so that software reusability can be achieved without reducing resource utilization or jeopardizing system safety.

ACKNOWLEDGMENT

This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research with the funding ID 01IS16025 and by the Deutsche Forschungsgemeinschaft (DFG) within the AORTA project with funding ID SCHR 603/9-1. The responsibility for the content remains with the authors.

References

[1] AbsInt Angewandte Informatik GmbH. *Safety Manual for aiT, Astrée, RuleChecker, StackAnalyzer*, May 2018.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: 2006 W'shop on Memory System Performance and Correctness*, pages 1–10, New York, NY, USA, 2006. ACM.

[3] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[4] K. R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE TOSE*, 39(12):1611–1640, Dec 2013.

[6] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, Washington, DC, USA, May 1999. IEEE.

[7] F. Bräunling, R. Hilbrich, S. Wegener, I. Stilkerich, and D. Kästner. Generic Software Tailoring Example. https://github.com/Gronner/GenericSoftwareTailoringExample.

[8] R. Chapman. Correctness by construction: a manifesto for high integrity software. In *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[9] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming*, volume 1766 of *LNCS*, pages 25–39. Springer, 2000.

[10] R. Dechter. *Constraint Processing*. Elsevier Science & Technology, 2003.

[11] C. Elsner, G. Botterweck, D. Lohmann, and W. Schröder-Preikschat. Variability in time – product line variability and evolution revisited. In *4th Int.*

W'shop on Variability Modelling of Software-intensive Systems (VAMOS '10), number 37 in ICB Research Reports, pages 131–138, Jan. 2010.

[12] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[13] R. Hilbrich and M. Behrisch. Experiences gained from modeling and solving large mapping problems during system design. In *IEEE Systems Conference 2017*, pages 620–627, Februar 2017.

[14] R. Hilbrich and H.-J. Goltz. Model-based generation of static schedules for safety critical multi-core systems in the avionics domain. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[15] R. Höttger, H. Mackamul, A. Sailer, J.-P. Steghöfer, and J. Tessmer. APP4MC: Application platform project for multi- and many-core systems. *it - Information Technology*, 59(5), jan 2017.

[16] D. Kästner and C. Ferdinand. Proving the Absence of Stack Overflows. In *SAFECOMP '14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security*, volume 8666 of *LNCS*, pages 202–213. Springer, September 2014.

[17] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.

[18] D. Kästner, M. Pister, G. Gebhard, and C. Ferdinand. Reliability of WCET Analysis. *Embedded Real Time Software and Systems Congress ERTS* [2], 2014.

[19] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In S. Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASIcs)*, pages 1:1–1:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] A. Kossiakoff, W. Sweet, S. Seymour, and S. Biemer. *Systems Engineering Principles and Practice*. Wiley Series in Systems Engineering and Management. Wiley, 2011.

[21] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, Jan. 2016. SEE.

[22] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *32nd Int. Conf. on Software Engineering (ICSE '10)*, New York, NY, USA, 2010. ACM.

[23] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.

[24] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. *Embedded Real Time Software and Systems Congress ERTS*[2], 2016.

[25] B. Pagano, C. Pasteur, and G. Siegel. A Model Based Safety Critical Flow for the AURIX Multi-core Platform. In *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan. 2018.

[26] D. L. Parnas. On the design and development of program families. *IEEE TOSE*, SE-2(1):1–9, Mar. 1976.

[27] K. Pohl, H. Hönninger, R. Achatz, and M. Broy, editors. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.

[28] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[29] O. Rogovchenko and J. Malenfant. Composition and compositionality in a component model for autonomous robots. In *Software Composition*, pages 34–49. Springer Berlin Heidelberg, 2010.

[30] O. Rogovchenko and J. Malenfant. Handling hardware heterogeneity through rich interfaces in a component model for autonomous robotics. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 312–323. Springer Berlin Heidelberg, 2010.

[31] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. ELSEVIER SCIENCE & TECHNOLOGY, 2006.

[32] W. Schwitzer, R. Schneider, D. Reinhardt, and G. Hofstetter. Tackling the Complexity of Timing-Relevant Deployment Decisions in Multicore-Based Embedded Automotive Software Systems. *SAE Int. J. Passeng. Cars - Electron. Electr. Syst.*, 6(2):478–488, Apr. 2013.

[33] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[34] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux kernel a software product line? In F. van der Linden and B. Lundell, editors, *International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, 2007.

[35] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012.

[36] P. Ulbrich. The I4Copter project — Research platform for embedded and safety-critical system software. `https://www4.cs.fau.de/Research/I4Copter/`, visited 2012-07-20.

[37] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *26th ACM Symp. on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM.