The Use of Java in the Context of AUTOSAR 4.0

Expectations and Possibilities

Christian Wawersich
cwh@methodpark.de
MethodPark Software AG, Germany

Isabella Thomm Michael Stilkerich
{ithomm,stilkerich}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

Modern cars contain a large number of diverse microcontrollers for a wide range of tasks, which imposes high efforts in the integration process of hardware and software. There is a paradigm shift from a federated architecture to an integrated architecture with commonly used resources to reduce complexity, costs, weight and energy.

AUTOSAR [3] is a system platform that allows the integration of software components (SW-C) and basic software modules provided by different manufacturers. The system platform can be tailored in a wide range to efficiently use the resources of the individual electronic control unit.

Software modules - mostly written in C or even Assembler - are rarely isolated from each other and have global access to the memory, wherefore an error can easily spread among different software modules. Memory protection mechanisms therefore gain more importance on the AUTOSAR platform.

Hardware-based memory protection - an optional feature in AUTOSAR systems – requires a memory protection unit (MPU), which is not present on low-end microcontrollers. By using type safe languages such as Java, software-based memory protection can be applied on these devices. The performance, footprint and resource consumption of a module written in Java are comparable to the C implementation, if novel and adapted tools for the application area are used.

This paper presents the KESO compiler tool for efficiently developing Java applications using software-based spatial isolation. Moreover, we show that a seemless integration with legacy C applications on an AUTOSAR system is feasible by means of the KESO component API generator[1].

1. INTRODUCTION

AUTOSAR (Automotive Open System Architecture), which we describe further in section 2, standardizes the software architecture for system functionality and drivers in automotive software. It is motivated by the growing complexity of software functionality provided in cars and facilitates the integration of multiple applications on fewer, more powerful microcontrollers.

Standardized, tested software modules as specified by AUTOSAR are shown to contain less software bugs. However, only a part of a complete AUTOSAR compliant software system is covered by the standard. Applications, ECU specific functionality and drivers for microcontroller external devices have to be developed independently for each project.

In a network of dedicated microcontrollers, the deployed software is physically isolated from each other. This isolation is missing among applications running on the same microcontroller, which enables malfunctioning applications to corrupt the memory of other applications, spreading the error and possibly resulting in a failure of all applications running on the same hardware.

Software development in the C programming language and Assembler tends to be error-prone, yet these languages dominate the development of embedded software. This, along with the ever-increasing software complexity, even aggravates the problem and increases the importance of isolation.

The automotive industry has realized these problems and introduced standards such as AUTOSAR [3] and Misra C [6], which restricts the C language to avoid programming errors, but also causes more inconvenient code. AUTOSAR OS [2], for example, addresses the necessity of memory protection by the optional use of the MPU of a microcontroller unit (MCU), which imposes some problems:

- Isolation can only be achieved if an MPU is present.
- Correct memory partitioning necessary for hardware-based protection is a non-trivial task.
- The lower performance caused by the use of an MPU is not acceptable for some developers, which often leads to the deactivation of the protection feature.

To address these issues, we developed KESO [10], a multi-JVM for deeply embedded systems. It offers software-based memory protection by the use the type-safe language Java, preventing the use of arbitrary values as memory references. Besides isolation, the use of a modern language such as Java avoids many programming errors and improves productivity and maintainability of the resulting software module. We will show, that the overhead caused by the use of Java by

[1]This work is sponsered by ESI

Embedded World 2011 March 1-3, 2011 Nuremberg, Germany
.

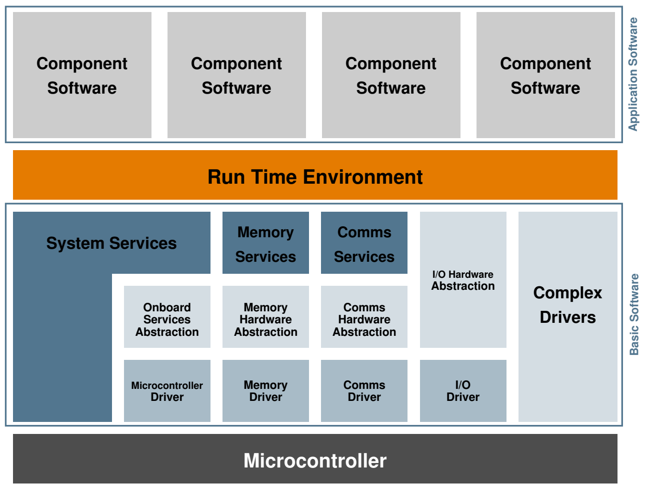

Figure 1: The AUTOSAR platform

means of KESO instead of the language C is viable for the benefits gained.

2. AUTOSAR

In this section, we briefly describe some important aspects of AUTOSAR. For a detailed explanation, please refer to [3]. The AUTOSAR development partnership, which was founded in 2003, is an association of car, ECU and MCU manufacturers as well as tool suppliers and basic software providers. AUTOSAR specifies the software architecture and basic software modules, that a static embedded system conforming to AUTOSAR consists of. The implementation of this standard is performed by the basic software providers and AUTOSAR-compliant modules can easily be exchanged.

One key understanding of the AUTOSAR consortium was that basic system functionality – that is not directly visible to car customers, such as hardware drivers – does not have to be implemented by all parties in a different way. This approach only leads to compatibility issues and various software bugs.

The competition regarding the expertise on software development can still happen in the application sector, which is directly visible to the customer. The AUTOSAR approach is a logic consequence of more powerful MCUs and development processes that already have taken place on the PC market, where system software such as Linux or Windows has been used for a long time.

Figure 1 illustrates the AUTOSAR platform, that coarsely is divided into:

- The application layer, that consists of so-called software components (SW-C).
- The basic software (BSW), that offers system functionality such as scheduling and communication mechanisms implemented by the OS and COM modules as well as memory and device access.
- The Runtime Environment (RTE), that provides uniform communication of SW-Cs and BSW on the same ECU as well as communication with other ECUs in the network. The kind of the actual communication – CAN or LIN, for example – is transparent for the application programmer. In this way, an SW-C can easily be moved to a different ECU without the need of adaption of the implementation. The only thing that changes, is the configuration information of software modules. The RTE is a completely generated piece of code. If any configuration changes, the RTE has to be created again by an RTE generator tool.

BSW modules in AUTOSAR offer a huge range of possibilities to use a certain module. An example is the NVRAM module in the memory services, that is responsible to handle access to non-volatile memory, which can be an EEPROM or Flash, that is directly present on an MCU or can also be an external device on the ECU. Due to configuration of the NVRAM manager and the respective BSW memory modules, the setup needed for a specific need can be chosen. Functionality, that is not needed can be omitted during the generation process, which leads to an acceptable footprint of the binary image. As can be seen, a lot of work has moved from the implementation to the configuration domain. Configuration information for AUTOSAR modules are placed in XML description files. To ease the handling with the numerous description files, code generators and the correct ordering, a workflow for the integration process has also been suggested by the AUTOSAR Methodology.

However, there are some areas, that are only partially specified by the AUTOSAR standard, since they are specific to a certain project and ECU:

- The SW-C layer, which contains the application logic
- The ECU Abstraction, which provides a software interface to the electrical values of any specific ECU in order to decouple higher-level software from all underlying hardware dependencies
- Complex device drivers for direct hardware access

SW-Cs have standardized interfaces and communicate with other modules via ports. They contain the application logic that can be found in *runnable entities*. Runnable entities are scheduled by the RTE according to the configuration for *events* specified in SW-C description file. The description file, that contains the formal design for an application, facilitates the decoupling of application development and system integration.

Complex device drivers allow to circumvent the AUTOSAR layered architecture to directly access the hardware. This module is particularly interesting for time critical applications and also facilitates the migration from a usual automotive software system to an AUTOSAR compliant system. The use of complex device drivers should be limited, since there is the risk of falling back to obsolete development practice eliminating the advantages of the AUTOSAR platform.

Modules such as SW-Cs or other project-specific parts are a good choice for an implementation with KESO (section 3), since they often have to be implemented for a special purpose, which allows for a smooth migration as described in section 4.

3. KESO: A MULTI-JVM FOR DEEPLY EMBEDDED SYSTEMS

KESO allows several Java applications to safely coexist on a single microcontroller by providing a Java Virtual Machine

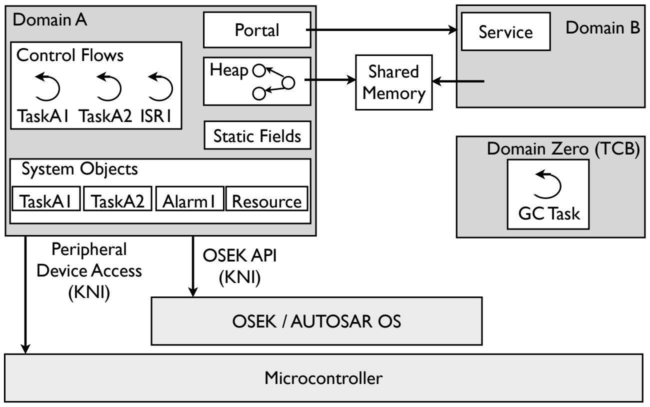

Figure 2: The KESO architecture

(JVM) instance for each application. The KESO architecture is depicted in Figure 2. In this section, we will give a short overview on the target domain and the different architectural components that a KESO system is composed of. A more detailed description of KESO can be found in previous publications [10].

3.1 Target Domain

KESO targets statically configured applications to be run on even deeply embedded systems. The primary development platform was an OSEK/VDX system running on an Infineon Tricore TC1796, which is a common microcontroller in the automotive sector. But also other platforms were used and the smallest microcontroller so far, was the ATmega8535[2].

3.2 Ahead-of-Time Compilation

The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. Normally, this bytecode is interpreted or just-in-time compiled by the JVM at runtime. Both techniques introduce a significant overhead to the amount of code.

For this reason, we opted for generating native code ahead of time, which facilitates to achieve a very slim runtime environment and performance comparable to that of applications written in languages such as C. Instead of directly compiling the Java bytecode to native code, our compiler *JINO* emits ISO-C90 code, which has some advantages over directly generating native code:

- No need for a JINO backend for each supported target platform. A standard C compiler is available for almost any of the target platforms.
- The available C compilers allow to create highly optimized code at the function level. We can therefore concentrate on high-level optimizations in JINO and leave the low-level optimizations to the C compiler.
- Existing tools (e.g., OSEK/VDX build enviroment, WCET analysis) operating on C code remain applicable.

The generated C code does not only contain the compiled class files, but also the KESO runtime data structures. Moreover, additional code is inserted to retain the properties of a JVM, such as `null` reference checks and array bounds checks, and the code of other services of the KESO runtime environment, such as garbage collection and inter-domain communication.

The generated KESO runtime is tailored towards the application's requirements, so the infrastructure code and data required for features such as floating point arithmetic or the support for multiple domains will only be added to the runtime if used by the application.

3.3 Protection Domains

The fundamental structural component in a KESO system is the protection *domain*, which defines a realm of protection and enables different applications to peacefully coexist on a microcontroller with communication limited to a set of well-defined and safe communication channels. From the perspective of the application, each domain appears to be a JVM of its own, which is why this architecture is also referred to as a Multi-JVM.

Domains are containers of control flows (i.e., tasks/threads and interrupt service routines (ISR)) and system objects (i.e., instances of operating system abstractions such as resources/locks or timers/alarms). Actions on these system objects are also limited to control flows within the same domain, except for system objects that are explicitly made available to other domains. The special domain Zero is part of the trusted computing base (TCB) and contains privileged control flows of the runtime environment such as the garbage collector.

Spatial isolation ensures that control flows are only able to access memory of data regions belonging to the 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, KESO ensures that a reference value is never present in more than a single domain and all inter-domain communication mechanisms must ensure that no reference values can be propagated to another domain.

3.4 Inter-Domain Communication

KESO provides several mechanisms that enable domains to interact with their environment. The *KESO Native Interface* (KNI) enables unsafe operations such as configuring peripheral hardware or interfacing with native libraries and the operating system API. The KNI provides a mechanism to extend the ahead-of-time compiler (JINO) with so-called weavelets. A weavelet takes over the C code generation for registered join points (e.g. methods calls, method bodies, field accesses) and replaces the generated code with application specific code. In this way, extentions can be woven into the generated code without additional overhead.

Another inter-domain communication mechanism provided by KESO is a kind of *shared memory*. Untyped memory areas can be allocated, dynamically or statically via a memory service. Within the domain, the allocated memory area is represented by a *memory object* and the data can be accessed via set/get-methods. It is possible to allocate the same physical memory area in different domains or the memory object can be passed to a different domain. To access the memory area via set/get-methods is like an array access,

[2]The ATmega8535 is a 8-bit microcontroller with 8 KiB of Flash ROM and 544 bytes internal SRAM.

where the items are adressed relative to the start adress of the memory area. A boundary check ensures that all accessed items are placed within the allocated memory area.

A memory object can be mapped to a *memory type* for saving the costs for address calculation and boundary checking at runtime. The activity of mapping a memory object to a memory type is similar to activity of casting untyped memory to a struct in C.

Portals provide an RPC-like, control-flow-oriented mechanism to communicate between domains. A domain can promote the interface of a service class to other domains via a global name service. The promoted interface is represented by a proxy object in the client domain and the methods can be accessed via a normal method invocation. Portals can also be used to communicate between domains on different microcontrollers as presented in an earlier publication [12] to create distributed applications. To ensure the spatial isolation of a domain, all parameters are passed by value with the only exception of memory objects where only the memory object is copied but not the represented memory area.

4. JAVA SOFTWARE COMPONENTS

In this section, we describe how KESO can be used to implement AUTOSAR SW-Cs in Java and let them communicate with AUTOSAR BSW modules and other SW-Cs written in C.

4.1 Approach

KESO was designed to create C code out of Java bytecode, which can be integrated into an OSEK/VDX environment. Since AUTOSAR OS is the successor of OSEK/VDX with supplementary features, the generated C code can easily be integrated into an AUTOSAR system.

But AUTOSAR has also a lot of additional functionality in contrast to OSEK/VDX. The RTE, for example, provides a powerful abstraction layer for communication among local and non-local SW-Cs and the BSW. The usage of the RTE is complicated in comparison to the portal mechanism provided by KESO, but it offers more functionality (e.g. asychronous communication) and it is mandatory for a smooth integration of SW-Cs into an AUTOSAR system. Therefore, we decided to make the AUTOSAR RTE accessable to the KESO domain.

4.2 RTE Access

4.2.1 Java RTE interface

The communication ports of a AUTOSAR SW-C are described in an XML configuration file. The configuration is used by an RTE tool to create the needed implementation for the communication channels. We use the same configuration file to create a Java class representing the RTE and provide the C functions as static Java methods. The application programmer can therefore use the methods as one would use the original C functions.

In addition, a KNI weavelet (see chapter 3.4) is appropriately configured to replace the method invocation at the caller side with the correspondig RTE call. While the function call is woven into the generated C code, there will be no additional overhead for ports with primitive parameters only.

4.2.2 RTE parameter handling

Primitive parameters like integers can be passed to the RTE call without conversion as long as there are compatible types in Java. In Java, all integer types are signed, unsigned types do not exist. The application developer has to handle these cases. Primitive parameters are passed by value and therefore they will not break the isolation property of the KESO domain.

Complex parameters (pointers, arrays or structs) are more complicated. We handle these cases by the use of *memory objects* and *memory types*. Appropriate classes for the *memory types* and glue code to direct the pointers are generated where needed.

4.3 Integrated Generation Process

In this section, we will show how KESO can be embedded in the standard AUTOSAR generation process. Figure 3 shows the combined approach. For a detailed description of the generation process, please refer to the AUTOSAR specification. The light-grey colored boxes depict the standard generation as specified by AUTOSAR, while the dark-grey boxes represent the components of the Java toolchain. The generation according to AUTOSAR requires a range of formal descriptions and configuration files in the XML format for the:

- SW-Cs (e.g. interfaces, runnable code)
- ECUs (hardware characteristics)
- System constraints (e.g. communication matrix), whereby a system is a compound of ECUs

The files are used to create the system configuration description, that contains the descriptions for each ECU in a network. A lot of BSW modules also use configuration files and their generators for creating BSW modules tailored for a certain setup. Java BSW modules and available generation tools can also be embedded in this tool chain.

The SW-C descriptions are used for the component API generators. They generate the interface functions, that are needed for communication with BSW services, other SW-Cs or ECUs in the network. For KESO, we have the *KESO component API generator*, that takes standard SW-C descriptions and creates the respective Java API for Java SW-C, which facilitates a smooth integration. After the API generation, application development can be performed independently of the system integration process.

A Java compiler is used to create bytecode from all Java source components. The generated bytecode and third-party bytecode code are processed by JINO. The emitted C code is then compiled and linked with all other C files to a binary image, that can be loaded onto an ECU.

4.4 Evaluation

For a detailed evalation of KESO in comparison to standard C applications, please refer to [10], where the flight attitude control algorithm of the I4Copter quadrocopter [11] was ported to Java. The main control unit on the quadrocopter is an Infineon Tricore TC1796 [3] device, that is also often employed in the automotive domain.

[3]150 MHz CPU clock, 75 MHz system clock, 1 MiB MRAM

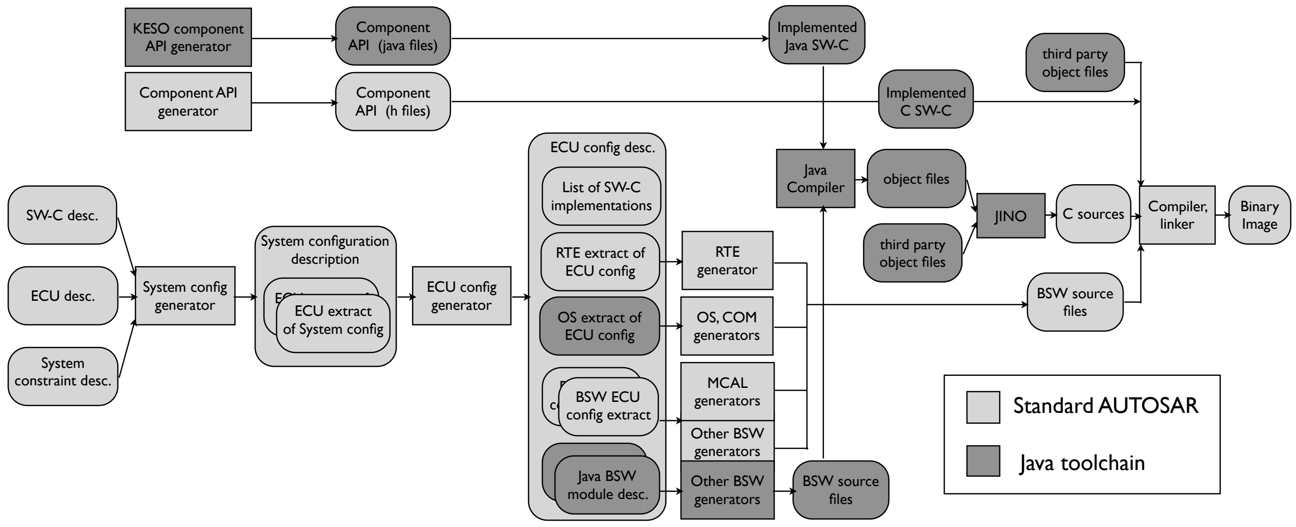

Figure 3: Combined KESO and AUTOSAR Generation Process

5. CONCLUSION

Both KESO and AUTOSAR provide an abstraction to implement software components, which can be distributed more easily between different microcontrollers. In AUTOSAR, the communication ports between software components are provided by the RTE. For a smooth integration of KESO Java applicatons into an AUTOSAR environment, we created a tool to access RTE ports. The needed RTE interface is generated from the XML-configuration defined in the AUTOSAR specification. The Multi-JVM KESO was designed to execute Java bytecode in an OSEK/VDX environment. It assumes a static system configuration, offers software-based spatial isolation and brings the advantages of a modern programming language to the embedded domain, which makes KESO different to other embedded JVMs [1, 5, 8, 4, 7, 9]. KESO reduces the runtime and memory requirements to a reasonable amount. The results of the evaluation are consistent with those of an earlier evaluated prototype application [13]. In both applications, the overhead in code size is less than 10%. Given that KESO could also provide more efficient implementations of the library functions (or simply use the C library's function via the KNI), the overhead in execution time is less than 10% as well.

6. REFERENCES

[1] AJACS: Applying Java to automotive control systems concluding paper V2.0, July 2002. `http://www.ajacs.org/`.

[2] AUTOSAR. Requirements on operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] AUTOSAR homepage. `http://www.autosar.org/`, visited 2009-03-26.

[4] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX Association.

[5] T. Harbaum. NanoVM - Java for the AVR, July 2006. `http://www.harbaum.org/till/nanovm`.

[6] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. Oct. 2004.

[7] F. Pizlo, L. Ziarek, and J. Vitek. Real time java on resource-constrained platforms with fiji vm. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 110–119, New York, NY, USA, 2009. ACM.

[8] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java^{TM} on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *Proceedings of the 2nd USENIX International Conf. on Virtual Execution Environments (VEE '06)*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[9] J. H. Solorzano. TinyVM - Java VM for Lego Mindstorms RCX. http://tinyvm.sourceforge.net/, 2000.

[10] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. KESO: An open-source Multi-JVM for deeply embedded systems. In *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 109–119, New York, NY, USA, 2010. ACM.

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

[12] C. Wawersich, M. Stilkerich, R. Ellner, and W. Schröder-Preikschat. A distributed middleware for automotive applications. In *Proceedings of the 1st Workshop on Models and Analysis for Automotive Systems*, volume 1, pages 25–28, 2006.

[13] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, pages 85–96, Boston, 2007. Springer-Verlag.