Exploiting Template-Metaprogramming for Highly Adaptable Device Drivers – a Case Study on CANARY an AVR CAN-Driver

Christoph Steup, Michael Schulze, Jörg Kaiser

[1]Department for Distributed Systems
Universität Magdeburg

Universitätsplatz 2, 39106 Magdeburg, Germany

Christoph.Steup@st.ovgu.de, {mschulze, kaiser}@ivs.cs.uni-magdeburg.de

Abstract. *Providing applications with a perfectly tailored device driver is essential to avoid the waste of resources. This is even necessary for the broad field of embedded systems development. However, the development of device drivers is in general a difficult task, and supporting a portable, configurable as well as adaptable device driver is even harder. To achieve such a device driver architecture, we propose declarative configuration specifications, exploit template-metaprogramming, and introduce the new concept of RegisterMaps. We evaluate the device driver architecture, showing that the device driver's resource usage scales with different configurations. We compare our device driver architecture against a device driver implementation of a hardware vendor, proving the competitiveness of our solution.*

1. Introduction

The development of device drivers is a difficult task. Allowing the use of a driver in different products, on different platforms, and with different application demands, requests development mechanisms that handle the challenge of these variability. In the embedded systems field, an important requirement – resource consumption – has to be treated additionally. Thus, a minimal RAM/ROM footprint device driver is desirable that gives applications only the functionality they need but not more. Providing a device driver that is portable, configurable as well as adaptable and uses resources very efficiently makes the development even harder, and in development for embedded devices, all these aspects are of major interest.

Portability, configurability and adaptability are properties device drivers should have, however to understand the differences between them, it needs further explanation. In general, a portable device driver allows for using on different platforms. From the application point of view, a need is a stable driver interface. Against such an interface applications can be implemented. If the interface stays unchanged even in case of different hardware platforms, application migration without code changes in the best case will be possible, leading to decreasing costs when new platforms have to be used. Hence, the development of such application is simplified, since no platform dependencies are propagated from the driver level up to the application level.

The configuration of a device driver is the process of setting parameters for tailoring the functionality to application demands, thus the device driver behaves as intended.

For example, if a device driver supports a polling and an interrupt feature, and an application decides to use polling only, the interrupt functionality will be switched off. This transition or change in the functionality is part of the adaptation process of the device driver. Thus, an adaptable device driver has the ability to serve different demands on the one hand arising from applications as configuration requests and on the other hand due to features given by the used hardware platform. The latter adaptation is done automatically without an explicit trigger by an application or user.

Developing a device driver having all preferred characteristics is as mentioned a difficult task. However, we are not the first one that try to realize adaptable software at all. Different development techniques like conditional compilation, object-orientation, feature-orientation, aspect-orientation, or component-orientation approaches try to tackle the problem, but all have their own strength and weaknesses. Because device drivers are usually written in C or C++ we focus on techniques that are only available for these languages.

From the developer's point of view, conditional compilation with the help of a preprocessor tool, mainly the C-preprocessor *cpp*, is easy to learn. Statements like *#define*, *#ifdef*, *#else*, etc. are used for framing areas that should be included or excluded in dependence of the given condition. However, due to the annotation of driver code with preprocessor directives the whole code is cluttered and obfuscated, leading to hard maintainability. Furthermore, the annotations are not type safe, because it is usual text processing only, allowing for every transformation, even those creating incorrect sources. This is error-prone, because the developer has a different view on the sources as the compiler. In the literature, the use of the preprocessor is often criticized. For example, [Spencer and Collyer 1992] demand "*#ifdef* considered harmful" and [Lohmann et al. 2006] speak from the "*#ifdef*-hell".

An alternative development technique is object-orientation, which is also supported by C++, due to its nature as multi-paradigm language. Implementing the preferable features with the object-oriented approach can be done in two directions. Firstly, one can implement the device driver as a class library, that defers all possible configuration steps to the runtime, having a dynamic, parameterizable driver, but leading to an immense overhead concerning resource usage. Secondly, the opposite is programming all the functionality by fine-modular, structured sub-classing. While this achieves minimal code overhead, it is a maintenance nightmare due to the exponential class explosion with every new upcoming feature [Gamma et al. 1995].

Device driver development with the component-oriented approach like with UML [Object Management Group 2001] generates a resource overhead, which is at least in the same magnitude as the dynamic, parameterizable device driver of the object-oriented approach. A disadvantage in this context is the loose coupling of components that lead to always have function calls. Furthermore, the compiler is not able to optimize code above component boundaries. Thus, reasoned by the black-box concept of components fine-granular adaptations are not designated, and applications are encumbered with unneeded functionality.

Components as well as object-orientation try to go the way of "separation of concern". Techniques like aspect-orientation (AOP) and feature-orientation (FOP) aim in

the same direction, but consider other mechanisms. Usually a programming language is extended. On the one side, AOP [Kiczales et al. 1997, Tarr et al. 1999] tackles the problem of cross-cutting concerns with aspects as modularisation concept, encapsulating the cross-cutting issue. Aspects are often related with (global) system policies like synchronisations, tracing, and so on. However, in the context of developing a device driver, cross-cutting issues arise seldomly. To allow aspects getting involved in C++ developments, a language extension like AspectC++ [Spinczyk et al. 2002] is needed, which works as source-to-source preprocessor having type-safety in place.

Feature-oriented programming is a technique that tackles the class explosion problem of the object-oriented approach by adding new keywords to the programming language, allowing for describing the relation of base classes and their extensions without directly inheriting. This enables defining multiple extensions to a given base. The configuration is done with the help of an additional grammar and a FOP compiler that transforms the sources according to the configuration. FOP, first introduced by [Prehofer 1997], aims at supporting a way to have variability in the design, and allowing the system tailoring to applications needs in a separate configuration step [Batory et al. 2004].

All the mentioned mechanisms have either drawbacks like error-proneness, heavy resource-consumption or need additional tools that are often in an early development state due to their research character. Another way to reach variability and adaptability is using techniques like generic programming [Czarnecki and Eisenecker 2000], and one form is template-metaprogramming. Template-metaprogramming is a turing-complete functional language that is processed by the compiler during the template instantiation phase. It allows for code generation, constants calculation, type selection, etc. and enables producing only the needed functionality having optimal fitting code at the end. However, if something goes wrong during the compilation, the compiler is highly verbose and generates a lot of messages that can be difficult to interpret. This is a drawback for the development in general, but there are mechanisms to address this problem: on one side by using special programming constructs that enables customized error message generation and on the other side by better compiler support.

For our goal, obtaining a highly adaptable device driver, we exploit the template-metaprogramming concept. We enhance this concept, by providing declarative configuration descriptions that hold the application requirements on the driver, and using the descriptions as input to the metaprograms to tailor the driver's functionality. For driver adaptation on the hardware side, we propose a new concept – RegisterMaps.

The rest of the paper is structured as follows. We start with the explanation of the concept in Section 2, where we describe our basic device driver architecture first. Next, we consider the declarative configuration in Section 2.2, the new RegisterMaps concept in Section 2.3 and the used mechanisms of the template-metalanguage in Section 2.4. Section 3 discusses the resulting device driver and compares it with a device driver implementation of a hardware vendor. In Section 4 we conclude the paper and give a short outlook on future research questions.

2. Concept

2.1. Basic Architecture

The architecture of our highly adaptable device driver is depicted in Figure 1. There are three main aspects, which will be addressed in the fallowing. The definition of configuration parameters are the first aspect. This definition is declaratively done by the application developer, and in Section 2.2 we describe how the declarative specification is realized. The *Configuration* component visible in the Figure 1 uses this application-dependent parameter specification for adapting the driver.

Secondly being portable, we use our RegisterMaps (short *RegMap*) concept, which abstracts the access to the hardware. RegisterMaps acts as mediator between the hardware and the hardware abstraction layer, hiding the underlying bit structure and providing an access by name. In Section 2.3 we discuss the RegisterMaps concept in detail.

To adapt the driver to an application-dependent declarative configuration, we use template-metaprogramming. During the compilation of the driver, the compile-time interface of the driver is invoked by the template instantiation process, leading to a tailored device driver. In Section 2.4 we show which concepts of the template-metaprogramming approach we use and how we combine it with the RegisterMaps concept. The driver's resulting API is highly specific to the driver and in our case specific to a CAN driver. However to understand our proposed approach of an adaptable device driver, discussing the API is not needed.

In the middle of the Figure 1, the *interrupt handler* component is depicted. It is drawn dashed, because interrupts can be completely configured out of the driver, thus neither RAM nor code memory is needed, letting the application the option of having a polling- or an interrupt-driven device driver.

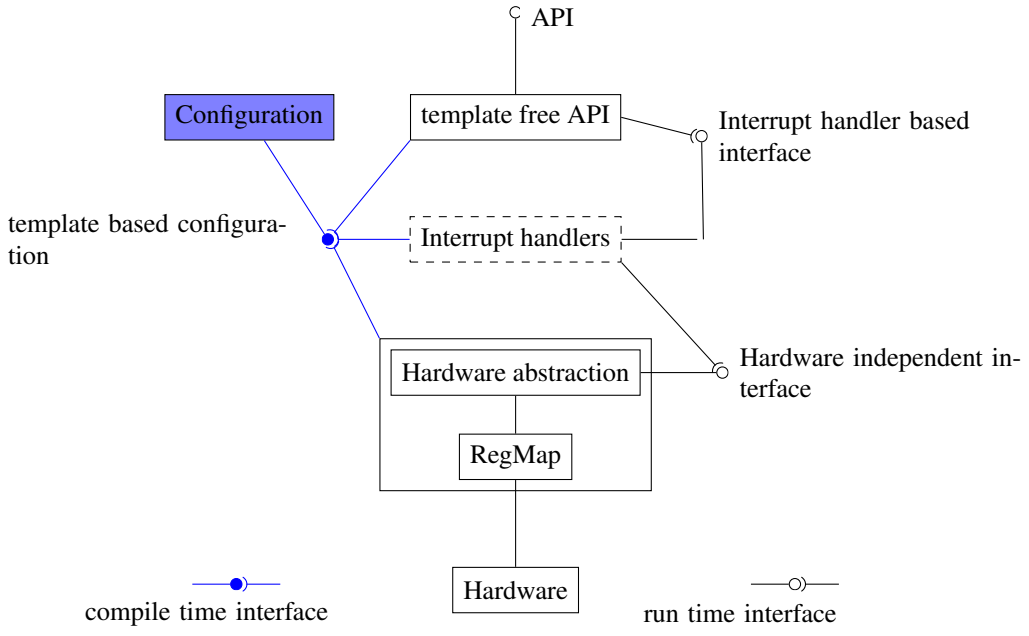

Figure 1. Components of an adaptive driver and their interfaces.

2.2. Declaration of configuration parameters

An adaptive device driver needs input, on which parameters it should adapt. The parameters are given by the user through a configuration structure. This configuration structure is the invocation parameter for the template-metaprogram at the compile-time interface, and it is used to adapt the functionality towards the user's requirements. However, there must also be certain default values in the case, that the user did not specify anything.

To have a flexible, yet resource efficient configuration, we use a C++-structure, which contains enumerations and typedefs. Both are compile-time constants, which will have no additional costs if not used.

```
1  struct CANConfig : defaultCANConfig{
2      typedef BaudRateConfig<
3                  F_CPU,
4                  SPEED_1M,
5                  SUBBITS_16
6              > baudRate;
7
8      enum Parameters{
9          version        = CAN_20B,
10         useReceiveInt = false
11     };
12 };
13
14 Canary<CANConfig>::type can_driver;
```

Listing 1. Declarative specification of configuration parameters for the CAN device driver

Getting a perfect user requirements fitting device driver, the user has to create such a C++-structure. The structure should be inherited from a default configuration to ensure that all configurable parameters are set at least to a default value. An example configuration is depicted in Listing 1.

After inheriting, the user can override the default values and adjust the parameters according to the desired behavior. To do so, he just creates an enumeration that contains the configuration parameters and the values he wants. In the case of the example the enumeration is called parameters. The example overrides the used CAN version to be CAN 2.0B and disables the receive interrupt. Also the baud rate is set to be 1 Mbit through the typedef of an additional helper structure BaudRateConfig. To apply this declarative parameter specification to the driver, the structure is given to the compile-time interface *Canary* as template parameter, resulting in an adapted device driver type *Canary<CANConfig>::type* in Line 14. Next the tailored device driver can be used through the created *can_driver* object.

The realisation of functionality inside the driver that correspond to the passed configuration parameters, depends on the used mechanism of the template-metalanguage, and we describe the used mechanisms in Section 2.4. With the help of the metaprogramming, additional features are possible like compile-time checks of configuration parameters. This enables us to notify the user of misconfiguration, e. g. if he tries to use functionality on the *can_driver* object, which is disabled in the current configuration.

As stated in the beginning of this paragraph the configuration is only one set of parameters, to which the driver has to adapt. The other set consists of the specification of the current hardware. Since these can be very tedious to handle, we propose a new concept to abstract the lowest level of hardware access in the next paragraph.

2.3. RegisterMaps – RegMaps

The goal of a RegMap is to provide an hardware independent, bit-wise access to memory mapped I/O-registers. This allows to reference single bits within a register by name. It also provides an abstraction mechanism that grants you hardware independence on I/O-register level. Thus means, that the position of registers in RAM and the order of bits inside a register are abstracted through the RegMap.

A RegMap first used in the avr-halib [Schulze et al. 2008] is realized as C++-structure with bit-field definitions, describing the abstracted registers and the contained bits. Bit fields are provided in the current C and C++ standards since years. Bit fields provide a way to access single bits of a register transparently without the need for bit mask and bit shift operations. Since the compiler does the burden of accessing the bits correctly, the usage of a RegMap reduces the amount of possible programming errors.

As an example of a RegMap we show Listing 2. This RegMap is used to abstract the access to the different interrupt enable flags of the used CAN hardware. An unused bit in a register cannot be left out of the RegMap, because it is still important for the order and position of the other bits in the considered underlying register. Declaring an unused bit, is done by creating an unnamed bit. Accessing such unnamed bit through the RegMap is impossible, which prevents programming errors. In Line four an example of an unnamed bit is shown. Unnamed bits are usually padding bits or bits that are reserved for future use e. g. in later processor revisions.

```
1 struct CanIntRegMap {
2     uint8_t timerOverrunInt   : 1;
3     uint8_t generalErrorsInt  : 1;
4     uint8_t                   : 1;
5     uint8_t timerOverrunInt   : 1;
6     uint8_t transmitInt       : 1;
7     uint8_t receiveInt        : 1;
8 };
```

Listing 2. An example RegMap, abstracting the interrupts of the CAN hardware.

Because a RegMap is designed to be an overlay for memory mapped I/O-registers, and therefore, it must be placed exactly at the position where the register resides inside the RAM. However, the compiler does not know about the special meaning of a RegMap's content, and if we would use the general construction process of C++ objects, it will be constructed somewhere in RAM. To tackle this problem, we use a special mechanism the *placement new operator* to overlay the RegMap exactly at the position of the respective I/O-registers. In our case, we use a macro *UseRegMap* to hide the syntactical burden of this mechanism.

Listing 3 shows the use of the defined RegMap. The example shows how the user specified configuration parameter *useReceiveInt* is used to set the receive interrupt flag

```
1  template<typename user_spec>
2  class Canary : ... {
3      public:
4          Canary(){
5              ...
6              UseRegMap(rm, user_spec::CanIntRegMap);
7              rm.receiveInt=user_spec::useReceiveInt;
8              SyncRegMap(rm);
9              ...
10         }
11         ...
12 };
```

Listing 3. Using the RegMap from Listing 2 to access the interrupt enable flag of the CAN hardware.

supported by the RegMap. At this point, no direct hardware access is needed. However in general, programming I/O-devices without much care leads to undefined behavior of devices. Since there is difference between the access to I/O-registers and memory. Because in case of I/O registers toggling bits multiple times in a sequence of commands is a valid programming sequence of a device. In contrast doing so on usual memory, only the last bit operation is important. Unfortunately, the compiler does not know about the difference between I/O-registers and memory. If we do not support the compiler with the right information, it will optimize aggressively by merging bit operations to create optimal code, but destroying the programming sequence of devices. This is in general a problem, and needs to be treated. We tackle the problem using a *SyncRegMap* that inserts an optimization boundary. Every operation concerning memory read or write must be finished up to this boundary until the next operation can be done by the compiler. This ensures correct behavior even with heavy optimization enabled. The *SyncRegMap* command is an alternative to make the whole RegMap volatile, which would have great negative influence on code memory size and performance.

To achieve hardware independence, for every used hardware a RegMap is needed. This is especially easy within similar families of micro-controllers, where many controllers have similar registers.

Having the abstract low-level access to the hardware available, the question how we select a specific RegMap arises. This problem is strongly coupled with the possibility to provide additional configuration parameters to the RegMap, which we provide by template parameters. The selection problem is solved using template-metaprogramming features of the C++-language, which we will look into in the next paragraph.

2.4. Template-metaprogramming

The concept of templates was already documented in 1990 in the “Annotated C++ Reference Manual” [Ellis and Stroustrup 1990]. Originally it was a technique to create parameterizable data structures. The parameterization is done by specifying types that are arguments for the data structure template instantiation. This allows the creation of generic data structures, because there is no need to write a data structure for every possible type. One popular example of this mechanism is the C++ Standard Template Library (STL)

vector class. Since object orientation is a way to reuse code, templates extended this concept to provide even more code reuse.

With the growth of the C++-language and the capabilities of templates, new ways to use templates were developed. Erwin Unruh showed the possibility to let the compiler calculate prime numbers as first [Unruh 1994]. However, Todd Veldhuizen was the one who recognized the potential behind this idea. He established a new programming approach known as template-metaprogramming [Veldhuizen 1995]. As said in Section 1 the template-metalanguage is a functional, turing-complete, side-effect-free language, that is processed during the compilation of an application. We mainly use certain aspects of the template-metalanguage, which are partial evaluation, template specialization and meta control structures. These aspects will be further described in the following.

The first used mechanism is partial evaluation. This gives us the ability to compile and include only the functions in the driver, which will be needed. For code, that is written in C or that is not using template-metaprogramming, the compiler will translate all defined function and put them in the resulting object file. In source code, that is parameterized by template-metaprogramming, however this is different. For this code, the compiler will only consider those functions that are at least called once. This saves code memory, because unused functions will not be included in the final driver.

The second used technique is template specialization. It is used to create special implementations of a class template for certain types like the CAN-ID class template in Listing 4. During the template instantiation process the compiler selects a specialized implementation that fits the given class template argument. We exploit this for obtaining the right CAN-ID class template specialization in Listing 4 in Line 34. The selection behavior is controlled by the *version* parameter of the *CANConfig* user configuration of Listing 1. The *version* parameter in the example configuration `CAN_20B` means the usage of CAN messages with extended IDs having 29 bits instead of 11 bits. A change of this parameter has an effect on the whole driver, because it leads to different used data structures. These structures differ in size, since the CAN-IDs need either 2 bytes for CAN 2.0A or 4 bytes in case of CAN 2.0B. Through the user-configured *version* parameter, the needed data structure will be automatically selected during compilation. The parameter is passed internally to all components of the driver to ensure consistent configuration. At this point only the driver is tailored to use the same CAN-ID in all components. However, the register content of the hardware also differs with the used CAN version. To cope with that we declared the RegMap to be a class template. This enables us to use the template specialization mechanism again to adapt to different I/O-register content dependent on the CAN *version* parameter. By using both times the same mechanism, the compiler ensures that transferring data between the driver data structures and the CAN hardware is correct by design.

Template specialization can also be exploited to create meta control structures like if-then-else for types. This opens the possibility to change whole block of functionality dependent on configuration parameters. For example, if a driver is configured only for sending messages, the receive functionality is not needed at all and is not existent in the final driver. In Listing 5 we show the selection of functionality for the receive interrupt behavior. The *Canary* template-metaprogam, here implemented as a class, inherits from the *if_then_else* class template. That meta control structure selects the second or the

```
1  // Supported CAN versions
2  enum Versions {
3      CAN_20A,
4      CAN_20B
5  };
6
7  template<Versions version>
8  class CAN_ID;
9
10 // Specialization for the CAN 2.0A CAN-ID specification
11 template<>
12 class CAN_ID<CAN_20A> {
13     public:
14         typedef uint16_t IdType;
15
16         enum Constants {
17                 idLength=11
18         };
19         ...
20 };
21
22 // Specialization for the CAN 2.0B CAN-ID specification
23 template<>
24 class CAN_ID<CAN_20B> {
25     public:
26         typedef uint32_t IdType;
27
28         enum Constants {
29                 idLength=29
30         };
31         ...
32 };
33
34 typedef typename CAN_ID<CANConfig::version>::IdType IdType;
```

Listing 4. Template specialisations of the CAN_ID type depending on the version

```
1  template<typename user_spec>
2  class Canary : public if_then_else <
3                              user_spec::useReceiveInt,
4                              ReceiveInterrupt,
5                              NoReceiveInterrupt
6                          > {
7      ...
8  };
```

Listing 5. Using a template meta control structure to (de)activate the use of the receive interrupt

third parameter dependent on the value of the first one. Thus, the *Canary* inherits from *ReceiveInterrupt* only if *useReceiveInt* is true in the configuration *user_spec*. This mechanism has the advantage, that features that are not present in the driver due to configuration does not cost anything (no code memory and no RAM).

The template-metaprogramming can be further used, to adapt a driver to hardware, where certain hardware features are not present. In such a case, a feature must be emulated by software. To only include the software emulation layer when it is needed, template specialization can be used to decide the usage. The compiler chooses the optimal fitting specialization for the platform. Since partial evaluation creates only the functions that are needed, and unused specializations are not considered and therefore not present in the final driver. An example for this would be a CAN hardware, which has no support for hardware ID filters. On such hardware, a device driver has to emulate the ID filter functionality in software if the application demands this. If the emulation is once implemented, it can be used on all hardware platforms that do not provide the feature. However platforms providing the feature do not need the emulation and the driver is created without it. With this mechanism used, creating a flexible, reusable and resource-optimal driver architecture is possible.

3. Evaluation

This paragraph deals with the results of the implementation. To show the advantages of our approach we compare the size of a set of example applications. On one side we use our highly adaptable device driver and on the other side an implementation from Atmel [at9], which is written in plain C.

We consider three different examples, describing usual use cases. All applications use CAN 2.0B, because the Atmel driver has no means to adapt itself to use CAN 2.0A only. The evaluated examples are: sending one message, receiving one message using polling and receiving one message using interrupts.

	Canary		Atmel	
example application	text	ram	text	ram
Sending	1248	17	4214	21
Receiving polling mode	1650	21	4168	21
Receiving interrupt mode	1950	29	n.a.	n.a.

Table 1. Program sizes and used RAM for different example applications

The compilation of all applications was done with gcc in version 4.3.4 and optimization switched on with -Os, leading to size optimized executables. For our driver we provided additional compilation flags concerning C++ code generation, which are: -fno-exceptions and -fno-rtti. These disable exception handling and runtime type informations, which are not used in our driver.

The result are contained in Table 1. It is clearly visible that our driver implementation is much more efficient in terms of code size. The RAM usage of our device driver is mostly better, but the interrupt driven version has an 8 byte overhead due to a delegate style interrupt handler. This delegate allows us to switch the interrupt callback at runtime, which is special feature that we offer. The RAM usage of the Atmel driver is constant, because it is not configurable and therefore uses the same data structures and functions for all examples. The interrupt example could not be measured, because, the driver from Atmel has no means to support interrupts natively.

To show the potential of our approach, we give additional measurements. As discussed, our driver can be configured in many ways. One of this is the minimal configuration of the third example supporting CAN 2.0A only. Doing so, saves additional 16.4% of code size and 19% RAM usage.

4. Conclusions and Outlook

In this paper we presented how template-metaprogamming is exploited for creating highly adaptable device driver. We applied our concepts for the development of a CAN device driver for an embedded platform, being adaptable, configurable and portable. The user has the possibility to configure the driver by using declarative descriptions. This descriptions are interpreted by template-metaprograms during the template instantiation phase. To abstract from low-level hardware issues, we introduced a new concept – RegisterMaps – allowing the development of portable driver architectures.

We evaluated the resulting driver architecture against a plain C implementation of a vendor's hardware driver. Our driver shows always better results in terms of code size and RAM usage. This is caused by its high configurability, because it contains only the needed functionality, but not more.

In the future we will apply our concepts on other types of devices to prove its generality. Furthermore, we will do measurements covering a broader range of configurations.

Acknowledgement

This work has partly been supported by the Ministry of Education and Science (BMBF) within the project "Virtual and Augmented Reality for Highly Safety and Reliable Embedded Systems" (VierForES).

References

AT90CAN32/64/128 software library. online, `http://www.atmel.com/dyn/resources/prod_documents/at90CANLIB_3_2.zip`. [(online), as at: 31.03.2010].

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling Step-Wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional. Published: Paperback.

Ellis, M. and Stroustrup, B. (1990). *The annotated C++ reference manual*. Addison-Wesley, Reading Mass.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect-Oriented programming. In Aksit, M. and Matsuoka, S., editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, page 220–242. Springer-Verlag.

Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schröder-Preikschat, W. (2006). A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the 2006 EuroSys conference on - EuroSys '06*, pages 191–204, Leuven, Belgium.

Object Management Group (2001). Complete uml 1.4 specification.

Prehofer, C. (1997). Feature-Oriented programming: A fresh look at objects. In Prehofer, C., editor, *ECOOP'97 — Object-Oriented Programming*, volume 1241, pages 419–433, Berlin/Heidelberg. Springer-Verlag.

Schulze, M., Fessel, K., Werner, P., and Steup, C. (2008). AVR-halib project website. online, `http://avr-halib.sourceforge.net`. [(online), as at: 25.02.2010].

Spencer, H. and Collyer, G. (1992). #Ifdef considered harmful, or portability experience with c news. In *the USENIX Summer 1992 Technical Conference*, page 185–197. USENIX Association Berkley.

Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002). AspectC++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60, Sydney, Australia. Australian Computer Society, Inc.

Tarr, P., Ossher, H., Harrison, W., and Sutton, S. (1999). N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 107–119, Los Angeles, CA, USA.

Unruh, E. (1994). *Prime number computation*. ANSI. ANSI X3J16-94-0075/ISO WG21-462.

Veldhuizen, T. L. (1995). Using c++ template metaprograms. *C++ Report*, 7(4):36–43. Reprinted in C++ Gems, ed. Stanley Lippman.