From Requirements to Specifications: A Case Study

Daniel Dietsch
Departement of Computer Science

Chair of Software Engineering
Albert-Ludwigs-University Freiburg

First Referee: Prof. Dr. Andreas Podelski
Second Referee: Prof. Dr. Peter Thiemann

Supervisor: Dr. Bernd Westphal

Thesis submitted for the Degree of Master of Science to the
Albert-Ludwigs-University Freiburg

· 2010 ·

Declaration

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg, April 22, 2010

Place, Date

Daniel Dietsch

Abstract

Formal software verification is concerned with the correctness of programs with respect to some specification. Although there exist examples of the usage of program verification tools and methods for large enterprises, the benefits remain inaccessible to most software developers and companies, because the usage of formal methods incorporates high entry costs: Expensive experts have to be employed or personnel has to be trained in expressing requirements and specifications of systems in formal languages. Especially small and medium sized enterprises do not have the necessary resources to provide this training or to hire experts. In this work we examine the path from informal requirements to verified programs on the basis of a case-study for a real-world embedded system. We use existing techniques to lower the complexity inherent to the creation of formal requirements and describe, how non-expert users can create specifications for programs from those requirements. The specifications can then be used by present program verification tools to decide if the program is correct with respect to the requirements or not.

Zusammenfassung

Formale Softwareverifikation beschäftigt sich mit der Korrektheit von Programmen in Hinblick auf eine Spezifikation. Obwohl es zahlreiche Beispiele für die Anwendung von Verifikationswerkzeugen und -methoden in großen Unternehmen gibt, erschließen sich deren Vorteile für die meisten Firmen und Softwareentwickler nicht, da der Einsatz formaler Methoden mit hohen Einstiegskosten verbunden ist: Entweder müssen teure Experten eingestellt oder das eigene Personal im Umgang mit formalen Sprachen für das Formulieren von Anforderungen und Spezifikationen geschult werden. Insbesondere kleine und mittelständische Unternehmen (KMU) verfügen nicht über die notwendigen Ressourcen, um ihre Mitarbeiter zu trainieren oder Experten anzustellen. Die vorliegende Arbeit beschäftigt sich daher mit möglichen Erleichterungen für diese Unternehmen. Wir betrachten den Ablauf, der von informellen Anforderungen zu verifizierten Programmen führt, auf der Basis eines Fallbeispiels für ein industriell entwickeltes eingebettetes System. Wir verwenden bereits vorhandene Techniken um die der Erzeugung formaler Anforderungen innewohnende Komplexität zu reduzieren und beschreiben, wie Benutzer, die nicht in der Anwendung formaler Methoden geschult sind, aus diesen Anforderungen Spezifikationen für ein bereits existierendes Programm erstellen können. Diese Spezifikation kann danach von bestehenden Programmverifikationswerkzeugen verwendet werden, um zu entscheiden, ob das Programm korrekt in Hinblick auf die Anforderungen an das System ist.

Acknowledgements

I would like to thank Andreas Podelski and all members of the Department of Software Engineering for the great work environment they provided and the many fruitful discussions we had since I started working there. In the context of this work, I would like to thank Amalinda Oertel for her guidance through the large amount of literature in requirements engineering and for the time she invested to contest my ideas. Furthermore, I want to thank Jochen Hoenicke, Jürgen Christ, Stefan Maus and especially Sergio Feo-Arenis for their valuable advice on the usage of VCC and some interesting conversations about the semantics of C, and Martin Schäf for his suggestions concerning scientific writing. Also, I would like to thank Axel Gembe for the explanations regarding the F.BZ 100 source code and hardware as well as for the discussions about usability of formal methods. Most important, I want to thank my advisor Bernd Westphal for the endless discussions, his patience with my sometimes too pragmatic approaches and his invaluable guidance through every aspect of this work: Without his paramount contributions, I could have never conquered the chasm between airy ideas and scientific writing. Last but not least I would like to thank Christin Gudopp for the many hours she sacrificed to correct this work.

On a personal level, I would like to thank Rebecca Albrecht for her continuing support during the dark weeks of writing as well as her patience with my writing-moods, and my parents, whose steady support through every phase of my life made everything possible in the first place.

Contents

1 Introduction

1.1 Motivation

Formal methods use mathematically-based techniques for the analysis of artifacts throughout the software development cycle. The usage of formal methods promises the *"cost-effective development of software with very low defect rates"* [1]. The cost-effectiveness of formal methods is due to the very low defect rate, which in turn allows dramatically lower maintenance costs. There is even reason to believe, that formal methods actually lower the overall costs [2]. But although by now there are numerous examples where formal methods are successfully applied in industrial settings (see [3] for a recent survey), they all take place in large enterprises. The main reason for this seems to be the high entry costs associated with formal methods [4]. The high entry costs are mainly caused by the high level of expertise needed for the successful application of formal methods [1, 5], which is not present in most developers. Large enterprises have the necessary resources to bridge this gap by providing extensive training or employing expensive experts. But small and medium sized enterprises (SMEs) usually can not afford those resources, although they are equally concerned with the development of safety-critical software systems. Because they cannot afford the entry costs, they cannot benefit from the later occurring reduction in maintenance-cost.

Most of the reported examples for the use of formal methods come from large, safety-critical systems like airplanes [6], railroad systems [7–10], floodbarriers [11, 12] or physical access-control systems [13]. Those systems have a need for high reliability as their failure puts human lives and large amounts of money at risk. Therefore, the broad use of formal methods despite their high entry costs is justifiable. Some SMEs are providing products with an equal need for high reliability, but due to the complexity of formal methods, they are commonly not able to employ them without taking large financial risks. This complexity of formal methods caused some responses among the research community prompting for a better usability of formal methods; Clarke and Wing stated more than ten years ago that formal software verification should be applicable *"with as much*

ease as compilers" [14] and Craigen et al. are convinced that *"*[verification-]*tools need to be integral parts of the development environment"* [15].

Nowadays many tools try to stand up to this challenge by providing easier user interfaces and by requiring lesser user interaction (e.g. [16–20]). But to the best of our knowledge, they still do not find wide application outside of large enterprises or research.

This work tries to find a way to introduce formal methods to SMEs without the need for extensive training. As a starting point, consider the following, not uncommon scenario:

In a small company, there is a programmer concerned with developing a software for an embedded system in C. The embedded system is going to be used in a safety-critical environment, e.g. it is responsible for the release of an airbag or for detecting fire and signaling an alarm. On his desk lies a document written by his boss, which contains all requirements for the system. Our developer is at the point where he wants to check whether his software fulfills those requirements or not. He knows that the system needs a high degree of reliability and therefore wishes to use formal program verification. But because he is an average programmer, he has no formal education in logic or modelling programs [1, 5], and therefore he wants to use a tool that works directly on the C code of the program. His first problem lies in the requirements document provided by his boss: All the requirements talk about the behaviour of the system as a whole, and not about the software. Inspecting the tools he wants to use, he discovers, that they need a specification in terms of the program, so he concludes: I need a way to express the requirements in terms of the program.

In the following, we will describe how we approached the problem of the developer. During this process, we also identify some practical obstacles that have yet to be addressed by research.

1.2 Research Question

The present work attempts to answer the following main question: How could SMEs with limited resources and non-expert personnel employ formal methods to verify, that the software of an embedded system is correct with respect to a set of requirements for this embedded system? An answer to this question depends on several sub-questions:

- When can we say that the software of an embedded system is correct with respect to the requirements for the whole embedded system?
- What kinds of requirements are verifiable on the software alone?

- What challenges have to be faced by developers in SMEs when using available tools and techniques for program verification, and how can we overcome them?

It is clear that the answers to those questions depend on a multitude of aspects, of which we can only address a few. Therefore we make some assumptions about the setting in which we search for an approach and the approach itself:

- We restrict the approach to programs written in C, because we are familiar with tools that can be used to verify C code.
- We consider only embedded systems for a number of reasons:
  - They are typically smaller, therefore we can perform our experiments faster.
  - Their requirements are closer to the system, which could make the relationship between requirements for the system to requirements for the software easier.
  - There is a greater likelihood for a safety-critical application of the embedded system, therefore we suspect that companies are more willing to invest resources in the application of formal methods.
- We consider only embedded systems that consist of one component, i.e. a single microprocessor with a single program running on it. The reason for this is that we do not want to concern ourselves with the relation between requirements for a whole embedded system and the requirements for the various single devices it could be made of.
- We require that the program directly interacts with the hardware. We assume that there is no additional abstraction layer between program and hardware (i.e. no operating system, no underlying middleware which encapsulates memory access or interrupts). The reason for this restriction is, that any additional abstraction layer between hardware and software could depreciate the significance of the verification results. We would have to make sure that the abstraction layer itself cannot violate the requirements, either by taking it into account or by assuming it is already verified.
- We assume that the hardware of the embedded system is already correct, as we do not provide support to analyze the hardware.

We use the following criteria to guide our search for an appropriate approach and justify some of our choices:

- Whenever possible we try to maximize the expected usability for our target group, the non-expert developers in SMEs. If we can choose between better analysis results and better usability, we opt for better usability. After all, utilizing only a small part of the improvements offered by formal methods is better than using none because the entry cost would get too high. This also includes the hiding of as much formalisms as possible from the target group.
- We try to design our approach with possibilities for automation in mind. If possible, we will try to divide the work by separating parts that we imagine can be automated (e.g. by providing new tools) from parts that will always be "hand-made". The possibility of automation allows for an easier concealment of formalisms and therefore for a higher likelihood of acceptance by the target group.
- We favor the reuse of artifacts or activities already available in embedded system development in SME over the introduction of new artifacts. Again, our target group is indeed interested in the results provided by formal methods, but they are unlikely to perform massive changes to their development process just to try it out. Therefore our goal must be to introduce the benefits of formal methods with as little change to their development process as possible.

1.3 Document structure

The rest of the document is organized as follows:

- Chapter 2 discusses the relation between requirements, system and software and provides an outline for our approach. We also define here what it means for a software to be correct with respect to the requirements for the system.
- Chapter 3 explains the three main aspects of our approach: How a non-expert user can formalize requirements, how he can relate them to the software and how this relation can be used to verify the correctness of the software with respect to the requirements.
- In Chapter 4 we present our case-study, which is based on an authentic embedded system. Furthermore, we report the challenges we had to face along the way and how they influenced our approach.
- Chapter 5 discusses the related work.
- In Chapter 6 we discuss our conclusions and possible future work.

2 Analysis

If we want to show that a program fulfills a set of requirements for an embedded system, we must first examine the relation between the requirements and the embedded system. This understanding is crucial, as we need to find out what we actually show with formal methods and how the correctness result of a program verifier relates to the correctness of an embedded system with respect to its requirements. In the following, we describe what we mean by requirements for embedded systems, or more generally, for products.

Requirements are a part of almost every product development, regardless of the product. Bridges, airplanes, kernel drivers or kitchen tools: If products are to be build, someone has to imagine what the product should be. Those requirements differ naturally in their size, their complexity and their level of abstraction, commonly defined by the complexity of the product: The greater the complexity of a product gets, the more abstraction takes place in certain stages of the development cycle; partly to tame that complexity as a whole, partly because specialized people attend to different parts of the product and want to abstract away things not important for them [21–24].

People view different parts of the product from different, abstract points of view, which, in turn, leads to more and different abstractions. The kind of abstraction is, for this argument at least, insignificant. But the nature of it is significant, as it demands a mapping from the abstract concepts to their concrete counterparts – without such a mapping, even an implicit or unused one, the best abstraction is useless, as it is no longer a suitable tool to simplify the thinking about the product.

If an abstraction was created in a reflective way – that is, by looking at an already existing concrete thing and describing it with a certain abstraction – it might be easy to get such a mapping. But requirements formulate an abstract idea of a yet non-existent product; there is no concrete thing (at least not yet) and therefore the mapping always contains a certain uncertainty. This unsureness does not only stem from the different possible implementations, but also from the overwhelming amount of implicit information contained in even the best requirements. Consider the following example requirement:

"The central unit has to refresh the system status on its display every 60s."

If we now wanted to verify this requirement for a given product, we would immediately observe some problems: What is the system status, what is the central unit and what exactly is the display of the central unit? The answers to those questions may be obvious if we had the necessary prior knowledge, if we had read the complete requirements document or if we simply had any experience in the corresponding area of expertise. Because we have those questions, we can now conclude that it is not enough to have requirements, one also has to know about the *domain* they stem from. Dines Bjørner defined domains in [25] as follows:

> *"A domain is (i) an area of human activity and/or (ii) an area of semi- or fully mechanised activity and/or (iii) an area of nature that can be described, and parts of all of which that can be potentially be subject to partial or total computerisation. We understand a domain when we can describe it in an objective way."*

Analogical, M. Jackson described in [23] that *"most computing problems are located in the real world – the physical world of employees, customers, lifts with doors and buttons, web sites, telephone switches, warehouses, aeroplanes, motor cars, railway trains, bank accounts and nuclear power plants"*.

Both, Jackson and Bjørner, talk about the same thing: The reasons to develop new products with new software. The distinction is only in scope, as domains are a part of the real world. As such, domains do not only provide the reason to build products, but also the terms to describe and constrain them. Early requirements are therefore necessarily stated in terms of the domain and have to view the product as black-box. They cannot look inside the product and talk about its inner workings. Ideally, they are the requirements that must be met to solve this certain problem in the domain, nothing more and nothing less. Therefore, we see the product as a solution to a problem in a domain.

A specialization of such a product is of course an embedded system. A system may consist of different components, i.e. functional entities consisting of software, a hardware on which the software runs and input and output devices that connect the hardware to the environment. As mentioned before, we assume that our system consists only of a single component, so that the requirements as well only talk about that one component.

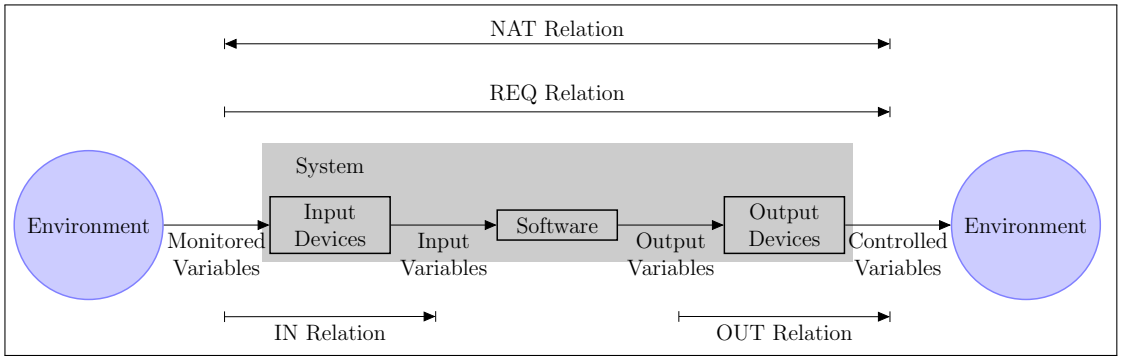

Figure 2.1: The Four Variable Model [26].

The connection between the domain and the system exists at the input and output devices of the hardware. This principle was introduced as four variable model by [26], which described the relation between requirements and software as a set of four relations (NAT, REQ, IN, OUT) over four variables (monitored, input, output, controlled). Figure 2.1 shows the scope of the relations over a system and its environment. The system is seen as an abstract composition of software and input and output devices. We are mainly interested in the correspondence between (a) input and monitored variables and (b) output and controlled variables. In the four variable model, the input variables are functions that map points in time to values of hardware registers. The monitored variables are functions, that map points in time to environmental quantities, i.e. values like temperature, pressure or states of buttons. Now the relation IN between the domain of monitored variables and the range of input variables describes the correspondence between hardware registers and environment, that is, our domain. Analog, the relation OUT describes a correspondence between output variables and controlled variables, i.e. again between hardware registers and environment.

We can see that the connection between software and domain is an indirect one; the hardware lies in between and provides another abstraction. Nevertheless, if the software controls the hardware, it needs to have access to all those in- and outputs that represent elements of the domain. But although the software is bound to know some representations of the domain, software is no longer a direct part of the domain and vice versa. Simply put: Software does not know what a light is.

This insight might seem trivial, but it helps with the question of when is the software correct with respect to the requirements for the system. A software may not know, what a light is, but the developer of the software does. By his design he chooses representations for elements inside a domain. Strictly speaking, we cannot show that a software fulfills requirements from a domain because the software does not directly influence the domain. But we can decide whether the software representatives of domain elements behave exactly like requested by

the system requirements or not. In order to do so, we have to make the relation between elements of the domain and their representatives in the software explicit.

If we interpret requirements for a system as rules for the behavior of elements in the domain, we can show that the representatives of those elements in the software have the same behavior. In order to define what it means for a software to be correct with respect to the requirements of the system, we need a notion for the concepts "software", "requirements" and "system":

Definition 1. *Let AP be a set of atomic propositions. A labeled state transition graph is a triple $G = (S, T, l)$, where:*

- *S is a set of states.*
- *T is the transition relation over S with $T \subseteq S \times S$. T is total, i.e. $\forall s \in S. \exists s' \in S. (s, s') \in T$.*
- *$l : S \to 2^{AP}$ is the labeling function which labels all states with a set of atomic propositions.*

A labeled state transition graph describes a system, whereas requirements constrain the set of possible state transition graphs, i.e. requirements describe a set of state transition graphs. Requirements also bring their own set of atomic propositions that express conditions over the elements in the domain. But of course, they normally allow other atomic propositions inside the system if it is not explicitly forbidden. We say the labeled state transition graph of a system is an element of the set of labeled state transition graphs described by the requirements if the system fulfills its requirements. We also use the notion of labeled state transition graphs to describe the software of the system, because then we can give a definition for the correctness of software with respect to the requirements of the system:

Definition 2. *The labeled state transition graph $G = (S, T, l)$ over the set of atomic propositions AP is a software. Furthermore, $\mathcal{G}_R$ is the set of labeled state transition graphs representing the systems allowed by the requirements and AP' is the set of atomic propositions of the requirements that are relevant to the software.*

A software is correct with respect to the requirements for a system if there exists a function $f : AP' \to AP$ such that $G' = (S, T, f^{-1} \cdot l)$ and $G' \in \mathcal{G}_R$.

As we earlier pointed out, domain requirements are ambiguous. But the ambiguity of domain requirements is essential. If we talk about alarms, diodes or buttons, we can imagine several things for each of those terms. The difference is given by the context provided by a domain. An alarm from an anti-virus program is entirely different from an alarm of a smoke detector, but only by mentioning

the context we can distinguish between them. When we want to show, that a system fulfills its requirements, we want to know, if it fulfills its requirements inside its domain. To verify that, we need a precise description of the domain, which allows us to distinguish between all the implicitly excluded meanings of the terms used in describing the requirements. We would need the whole domain knowledge in a formalized form. Because this is clearly impracticable, we have to rely on the developer to assign the right meaning to certain parts of the software by giving a relation between parts of the domain and parts of the software. Given such a relation, we can show that each relation between parts of the domain that expresses a requirement has a counterpart in the software, which relates parts of the software in the same way. Therefore, if we want to verify that a software satisfies our requirements, we will either need a relation from the terms of the domain to the terms of the software, such that we can encode the meaning chosen by the developer, or we need to explicitly state the domain knowledge in a way the preferred verification method can understand.

Now that we know what we want to verify, we turn our attention toward the methods we can use to verify it. Tools for program verification always need a specification of the properties of the software in terms of the software. While requirements for the system talk about the systems outside behaviour, those tools can only see the inside, that is, the software. But given the function from domain parts to software parts, we can encode requirements as properties of the software. Then a tool can examine whether the relationship between software representations of domain parts is the same as the relationship between domain parts with each other.

The selection of an appropriate tool is an important choice for our undertaking. The tool should be mature enough to be usable in an industrial setting, it should be aimed at verifying programs, not only finding bugs, and it should require as little expertise from its users as possible. Furthermore, the encoding of the relation between software representations of domain parts depends naturally on the specification language used by the tool. Ideally it should support every property expressible by the requirements which are relevant for the software.

Today there exists a wide variety of tool-supported formal methods to aid the development of programs. In order to make an appropriate selection, we first turned to various surveys of formal methods and their industrial appliance [3, 15, 27–31]. The first distinction that we can make is between *sound* and *unsound* tools: A sound tool guarantees that if it reports that there is no error, there is none. But it may report false errors, due, for example, an overapproximation of the possible program executions. An unsound tool, on the other hand, does not guarantee that there are no errors left if it cannot find any.

In our understanding, the class of unsound tools also entails all representatives of bounded model checking (BMC), a special class of the model checking technique

introduced by Clarke and Emerson [32]. BMC does not explore the whole state space of a program, but only to a certain path length, called the bound. Errors which occur only on longer paths are therefore not detected. Examples for unsound tools, that can be used to analyze C programs, include the famous static analyzer lint (developed 1979 by Bell Laboratories), CBMC [18], Saturn [33] and EUREKA [34] as well as commercial tools like Coverity [20] or Klocwork [35]. Unsound tools allow the use of efficient heuristics to analyse programs and are therefore typically much faster than sound tools. Nevertheless, they can never prove the absence of bugs, and that is what we are interested in. Therefore we have to discard all the unsound tools, although they provide very useful information to developers.

We also do not consider formal methods that require extensive user interaction like constructing a proof or a model manually. These tools may be appropriate for experts in formal methods, but as we already mentioned, our target group does not consist of experts. Examples include the interactive theorem provers PVS [36] and Isabelle/HOL [37] as well as Bogor [38].

Furthermore, methods that aim at generating verified programs from specifications, models or requirements are equally unusable for us, as we want to verify already existing programs. An example for this class of methods is the B-method [39].

The remaining tools are in principal feasible for our approach. Because commercial tools require more resources from SMEs, we do not want to rely on them. This excludes PolySpace [40], CodeSonar [41] and Astree [42].

This leaves us with SATABS [19], SLAM [16], Blast [43] and VCC [17]. In principal, all those tools are usable for our purpose. We favor the VCC as we already have some experience with it. VCC and its sources are freely available for non-commercial use from [44]. VCC was build to verify the Microsoft Hyper-V hypervisor [45] as part of the Verisoft XT project [46], in which our department is involved. Essentially, a hypervisor is an additional abstraction layer between operating systems and the hardware, which allows multiple operating systems to run on the same hardware platform side by side. Because VCC has to handle the complexity of a low-level program with considerable size (Hyper-V has approximately 100.000 LOC), we expect it to be usable for the programs from our target group as well. Besides a high level of automation and scalability, VCC also provides a tight integration in Microsoft Visual Studio [47], a commonly used integrated development environment (IDE). This integration allows an easy reporting of verification errors, comparable to the error messages provided by compilers [48].

The input to VCC is C code extended with annotations, which consist of function pre- and post-conditions, assertions, type invariants and specification code [17].

Those annotations are similar to the annotations found in other tools that work with annotated code, namely ESC/Java [49,50] for Java and Spec# [51] for C#. The annotations are specified by new keywords and a mixture between C and first-order logic (see Section 3.3.2 for details).

Let us recall our scenario from Chapter 1: We said our developer has a document written by his boss that contains the system requirements. We already assumed that the personnel in SMEs do not have the expert knowledge in modelling and formal logics that is necessary to successfully employ formal methods. Therefore we cannot assume that said document already contains a formal representation of the requirements. But our tool needs a formal input, so we have to formalize the requirements somewhere along the way to this input. The difficulty here lies not necessarily in the formalization itself, but in the difference between the expressiveness of requirements written in natural language and the properties a tool like VCC can verify. First of all, we expect general requirement documents to contain several requirements that have nothing to do with software, but rather with e.g. the color of the casing of the embedded system. We are not concerned with filtering such non-software requirements, as it is rather easy for a developer to decide whether a requirement belongs to this class or not. Second, there are requirements that do not talk about the behavior of the system but about how this behavior should be implemented, e.g. scalability, usability and the like. We call such requirements non-functional requirements and we cannot analyse them either. Their fulfillment has to be checked separately. The remaining requirements describe software-controlled behavior of the system. According to the definitions of [52], we can further distinguish those remaining requirements in the following classes:

- *Safety* requirements describe that *"something bad must never happen"*. They describe states of the system that must never occur, like simultaneous green lights in a traffic light system controlling a crossroad. In general, every violation of a safety requirement can be verified by giving a finite sequence of states that is permitted by the system and that contains a state where the property does not hold.

- *Liveness* requirements describe that *"something good has to happen eventually"*. In contrast to safety requirements, they specify not the absence of bad things, but the presence of good things. Conversely, they can only be violated in infinite sequences of states because the good thing may happen at any time.

- *Bounded Response* requirements provide time bounds to properties, that is they state that something must occur (like liveness) or something may not occur (like safety) in a certain time interval. For example, *"At most 10 seconds after the button press the green light has to go on."* is a bounded

response requirement. To verify bounded response requirements one needs information about the worst-case execution time of a system, which in turn depends heavily on the hardware and the environment [53]. Nevertheless, due to the boundedness of time this property can again be falsified with a finite sequence of states in which one state violates the property and it is known that this sequence of states lies within the time bounds.

We are aware of the additional classes of requirements, like duration requirements that specify that inside a given time interval a system has to or may not be inside a state for a given time, or probabilistic properties that specify the probability of being in a certain state, but because those classes belong again to entirely different areas of research, namely hybrid systems and probabilistic model checking, we do not discuss them any further. In fact, all of the previously described tools including the VCC are not able to verify anything but safety properties of software. But a preliminary analysis of an available real-world example project suggests that the requirements are in large parts real-time or liveness properties. Unfortunately, to the best of our knowledge there does not exist a tool that allows the direct verification of C code against real-time properties.

We already said that we need to formalize the requirements. Because we assumed that the personnel of SMEs are not trained in the use of formal logics, we have to provide a method that allows them to formalize requirements anyway. Fortunately there already exist numerous approaches to the formalization of requirements that support non-expert users, e.g. *structured natural language* and *requirement pattern systems*. Structured natural language approaches use a subset of natural languages like English and provide a predefined semantic to this subset. They also require a certain structuring of the requirement documents. An example of this approach can be found in [54, 55]. This class of formalization is not as expressive as requirement pattern systems, as the current state of the art does not provide a transformation in common formal logics like LTL, CTL or TCTL, but instead defines a new, own formal language. Furthermore, there are still practical problems as some ambiguities arise even inside the subset of the language, which have to be handled manually.

Requirement pattern systems utilize recurrences in written requirements to define abstract templates, that can be instantiated with atomic propositions. A template normally contains natural language description of the situation where it can be used along with examples and a mapping to one or more formal logics. A requirement pattern system pools those templates and provides some ordering, e.g. by the range of application, over the templates. Examples of such requirement pattern systems can be found in [5, 56–58].

We believe that requirement pattern systems are better suited to help formalize requirements, as they involve less changes in the general development process. To apply a pattern system, one only needs the system and additional space in the

requirements document or an additional document to write down the formalization. In approaches using structured natural language one needs training in the proper use of the subset of the language and one has to adhere to the strict structuring of the document, which requires larger changes in the existing processes in SMEs. Therefore we use requirement pattern systems instead of structured natural language to formalize requirements.

We close this section with a short summary of the main findings of this preliminary analysis regarding the mandatory elements of the approach:

- Embedded systems are solutions to a problem in a certain domain.
- Requirements for embedded systems describe the desired behavior of the embedded system in terms of elements of the domain.
- A software is correct with respect to the requirements for an embedded system if there exists a mapping from elements of the domain to elements of the software, such that the behavior of the software elements over time is the same as the required behavior of domain elements over time.
- We want to use a program verification tool which is sound, not commercial, does not require extensive knowledge in formal methods from the user and can work directly on C code.
- We use VCC as program verifier because it is among the tools that match our criteria and, as it is already used in our department, has the benefit of requiring a shorter training period. Nevertheless, other tools could be used in its place.
- Not all properties expressible in system requirements are verifiable with suitable tools, because either the properties have nothing to do with software or they express how a certain behavior should be implemented or the tools require too much expert knowledge to be used by SMEs.
- Tools which can formally verify properties of C code require a formal specification, therefore we have to formalize the requirements.
- Users without expert knowledge in formal methods cannot formalize requirements on their own, therefore we use requirement pattern systems to provide the expert knowledge needed, since they are developed to overcome this problem. We do not use structural natural language because it requires more changes in the already existing development process in SMEs.

3 Approach

This chapter describes our approach to the problem and is divided into three sections:

- We begin by stating our definition of requirements, and how they can be captured in a formal language with the help of requirement pattern systems in Section 3.1. This also includes a short background of the formal languages used and an equally short overview of the requirement pattern systems presented in [5, 58].
- Section 3.2 defines which artifacts need to be created in our approach to capture all necessary information for the transformation from formal requirements to an input for a program verifier.
- In Section 3.3 we define the formal specification that together with the program can be used as an input to a program verification tool. We also give an overview over the annotations used by VCC and show a toy example for the whole approach.

3.1 Requirements

We already described in Chapter 2 that we have two choices if we want to verify that a software satisfies our requirements: Either we give a relation from elements of the domain to elements of the software or we formalize the domain knowledge in a way the preferred verification method can understand. In our approach, we opt for the first alternative, because creating an explicit representation of a domain requires additional resources, which are already sparse in SMEs. A translation can be given comparatively easy, because we leave the huge amount of domain knowledge where it belongs: In the heads of the developers. In the following, we define some terms and concepts necessary to give a precise description of the translation itself and the process to obtain it.

3.1.1 Domain Requirements

A *domain requirement* is a constraint on a system in terms of the domain. The constraint must be met by the implementation of the system. The domain parametrizes the meaning of a domain requirement by providing the context for the terms occurring in the requirement. A single domain requirement must have the following properties:

- *Cohesive*: It addresses only one aspect of the system.
- *Unambiguous*: There is only one interpretation for the requirement.
- *Atomic*: It does not contain conjunctions.
- *Prescriptive*: It describes only aspects of a system which must be enforced, not aspects which are enforced by laws of nature. In other words, a system can fulfill the requirement or it cannot, it is not inherent to the environment that the requirement will be fulfilled.

Furthermore, we require that the set of domain requirements is *consistent*, i.e. there exists a system which fulfills all of the domain requirements.

We call domain requirements *formal* if they are written in a formal logic (like, e.g. first-order logic). Analogical we call domain requirements written in natural language *informal*, regardless of the degree of structuring.

In Chapter 2 we explained that we need to formalize the requirements somewhere along the way to the program verifier and that it is beneficial to formalize requirements as early as possible in the development cycle. We further explained, why we want to use pattern systems in general and why we select the pattern systems presented in [5, 58]. The next two sections give a short background of the formal logics (Section 3.1.2) used by the selected requirement pattern system (Section 3.1.3).

3.1.2 Background: The Temporal Logics CTL and TCTL

The following two sections give the definition of the branching time logics CTL and TCTL according to [59] and [60] respectively. CTL and TCTL are used by [5, 58] to specify their requirement pattern systems.

3.1.2.1 CTL

CTL Syntax: Let AP be a set of atomic propositions. The syntax of CTL formulas is inductively defined as follows:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EX\phi_1 \mid E\phi_1 U \phi_2 \mid A\phi_1 U \phi_2$$

where $p \in AP$ and ϕ_1, ϕ_2 are CTL formulas.

CTL Semantics: The semantics of CTL is defined with respect to Kripke structures. A Kripke structure M is a tuple $M := \langle S, s_{init}, \mu, E \rangle$, where

- S is a finite set of states,
- $s_{init} \in S$ is an initial state,
- $\mu : S \to 2^{AP}$ gives an assignment of truth values to atomic propositions in each state and
- E is a binary relation over S giving the possible transitions.

A path is an infinite sequence of states $(s_0, s_1, \ldots) \in S^\omega$ such that $\langle s_i, s_{i+1} \rangle \in E$ for all $i \geq 0$.

Given a CTL-formula ϕ and a state $s \in S$, the satisfaction relation $(M, s) \models \phi$ (meaning ϕ is true in M at s) is defined inductively as follows (because M is fixed, we abbreviate $(M, s) \models \phi$ to $s \models \phi$):

$s \models p$	iff $p \in \mu(s)$.
$s \models \neg\phi$	iff $s \not\models \phi$.
$s \models \phi_1 \wedge \phi_2$	iff $s \models \phi_1$ and $s \models \phi_2$.
$s \models EX\phi$	iff $s' \models \phi$, for some state s' such that $\langle s, s' \rangle \in E$.
$s \models E\phi_1 U \phi_2$	iff for some path $(s_0, s_1, \ldots)$ with $s = s_0$, for some $i \geq 0$, $s_i \models \phi_2$ and $s_j \models \phi_1$ for $0 \leq j < i$.
$s \models A\phi_1 U \phi_2$	iff for all paths $(s_0, s_1, \ldots)$ with $s = s_0$, for some $i \geq 0$, $s_i \models \phi_2$ and $s_j \models \phi_1$ for $0 \leq j < i$.

The Kripke structure M satisfies ϕ iff $(M, s_{init}) \models \phi$.

A CTL formula ϕ is called satisfiable iff there is a Kripke structure M such that $M \models \phi$.

Abbreviations: The CTL syntax is commonly extended by the operators EF, AF, EG, AG, EW and AW which are defined as follows: Let ϕ be a CTL-formula, the constant **false** is equivalent to $\phi \wedge \neg\phi$, the constant **true** is equivalent to $\neg$**false**:

$$
\begin{array}{ll}
EF\phi & \equiv E\ \textbf{true}\ U\phi \\
AF\phi & \equiv A\ \textbf{true}\ U\phi \\
EG\phi & \equiv \neg AF\neg\phi \\
AG\phi & \equiv \neg EF\neg\phi \\
E\phi_1 W\phi_2 & \equiv (E\phi_1 U\phi_2) \vee EG(\phi_1) \\
A\phi_1 W\phi_2 & \equiv (A\phi_1 U\phi_2) \vee AG(\phi_1)
\end{array}
$$

3.1.2.2 TCTL

TCTL explicitly adds time to the syntax and semantics of CTL. The formulas of TCTL are essentially CTL formulas extended with timing constraints and time quantifiers. Furthermore, the semantics of TCTL are no longer defined with respect to Kripke structures, but to a map from points in dense time to states. Because TCTL operates on dense time, the X-operator becomes futile and is dropped.

TCTL Syntax: Let AP be a set of atomic propositions, V be a set of variables and $\mathbb{Q}$ be the set of rational constants.

The syntax of TCTL formulas ϕ is inductively defined as follows:

$$\phi := p \mid (x + c) \leq (y + d) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid E\phi_1 U\phi_2 \mid A\phi_1 U\phi_2 \mid x.\phi$$

for $c, d \in \mathbb{Q}$, $p \in AP$ and $x, y \in V$.

TCTL Semantics: Let $t \in \mathbb{R}^+$ be a point in time, S be a set of states and $\mu : S \to 2^{AP}$ be a labeling function, which labels every state with atomic propositions. Then

- $\rho : \mathbb{R}^+ \to S$ is a map assigning points in time to a state and is called a computation. ρ satisfies the following condition:

  There exists an interval sequence $I_0 I_1 I_2 \ldots$ such that whenever two time values t and t' belong to the same interval I_i, $\mu(\rho(t))$ equals $\mu(\rho(t'))$.

  This condition ensures that the concatenation $\mu \cdot \rho$, which maps points in time given by $\mathbb{R}^+$ to the atomic propositions given by 2^{AP}, changes its values at most at ω points.

- ρ_t is the prefix of ρ up to time t. It is a map from $[0, t)$ to S obtained by restricting the domain of ρ.

- ρ^t is the suffix of ρ at time t. It is a computation defined by $\rho^t(t') = \rho(t+t')$ for every $t' \in \mathbb{R}^+$.

If ρ' is some map from $[0, t)$ to S, then its concatenation with ρ, denoted by $\rho' \cdot \rho$, is defined by:

$$\text{for } t' \in \mathbb{R}^+ : (\rho' \cdot \rho)(t') = \begin{cases} \rho'(t') & \text{if } t' < t \\ \rho(t' - t) & \text{otherwise} \end{cases}$$

A TCTL-structure is a tuple $T = \langle S, s_{init}, \mu, f \rangle$, where

- S is a set of states,
- $s_{init} \in S$ is an initial state,
- $\mu : S \to 2^{AP}$ is a labeling function which assigns to each state the set of atomic propositions that are true in that state, and
- f is a collection of computations ρ over S satisfying the properties
  - $\forall \rho \in f, t \in \mathbb{R}^+ : \rho^t \in f$ and
  - $\forall \rho, \rho' \in f, t \in \mathbb{R}^+ : \rho(t) = \rho'(0) \implies \rho_t \cdot \rho' \in f$.

Given a TCTL-structure T, a state $s \in S$, an environment function $\varepsilon : V \to \mathbb{R}^+$ and a time value $t \in \mathbb{R}^+$, the satisfaction relation $(T, s, t) \models_\varepsilon \phi$ is defined inductively as follows (again, because T is fixed we abbreviate $(T, s, t) \models_\varepsilon \phi$ with $(s, t) \models_\varepsilon \phi$):

$(s,t) \models_\varepsilon p$	iff $p \in \mu(s)$.
$(s,t) \models_\varepsilon (x + c) \leq (y + d)$	iff $\varepsilon(x) + c \leq \varepsilon(y) + d$.
$(s,t) \models_\varepsilon \neg\phi$	iff $(s,t) \not\models_\varepsilon \phi$.
$(s,t) \models_\varepsilon \phi_1 \wedge \phi_2$	iff $(s,t) \models_\varepsilon \phi_1$ and $(s,t) \models_\varepsilon \phi_2$.
$(s,t) \models_\varepsilon x.\phi$	iff $(s,t) \models_{[x \mapsto t]\varepsilon} \phi$.
$(s,t) \models_\varepsilon E\phi_1 U \phi_2$	iff for some $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $(\rho(t'), t + t') \models_\varepsilon \phi_2$ and $(\rho(t''), t + t'') \models_\varepsilon \phi_1$ for all $0 \leq t'' < t'$.
$(s,t) \models_\varepsilon A\phi_1 U \phi_2$	iff for every $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $(\rho(t'), t + t') \models_\varepsilon \phi_2$ and $(\rho(t''), t + t'') \models_\varepsilon \phi_1$ for all $0 \leq t'' < t'$.

A TCTL structure T satisfies a TCTL formula ϕ, written $T \models \phi$, iff $(T, s_{init}, 0) \models_{[V \mapsto 0]} \phi$. A TCTL formula ϕ is called satisfiable iff there is a TCTL-structure T such that $T \models \phi$.

The environment function ε gives the valuation of all the free variables in ϕ. The time value t gives the current time, and is used to bind the variable x while evaluating $(x.\phi)$ as shown in the following example:

$$\phi = x.E(y.y \leq x + 2 \vee p)U(z.z \leq x + 10 \wedge q)$$

Applying the semantics gives:

$$(s,t) \models_\varepsilon \phi \text{ iff for some } \rho \in f \text{ with } \rho(0) = s, \text{ for some } t' \geq 0,$$
$$(\rho(t'), t+t') \models_{[x \mapsto t]\varepsilon} (z.z \leq x + 10 \wedge q \text{ and } (\rho(t''), t+t'') \models_{[x \mapsto t]\varepsilon} (y.y \leq x + 2 \wedge p)$$
$$\text{for all } 0 \leq t'' < t'.$$

Unfolding ε and applying the semantics again gives:

$$(s,t) \models \phi \text{ iff for some } \rho \in f \text{ with } \rho(0) = s, \text{ for some } t' \geq 0,\ q \in \mu(\rho(t')) \text{ and}$$
$$(t + t' \leq t + 10) \text{ and for all } 0 \leq t'' < t', \text{ either } (t + t'' \leq t + 2) \text{ or } p \in \mu(\rho(t'')).$$

Abbreviations: TCTL also defines – analogous to CTL – additional temporal operators around the U-operator, namely EF, EG, AF, AG, EW and AW:

$$\begin{aligned}
EF\phi &\equiv E\ \mathbf{true}\ U\phi \\
AF\phi &\equiv A\ \mathbf{true}\ U\phi \\
EG\phi &\equiv \neg AF\neg\phi \\
AG\phi &\equiv \neg EF\neg\phi \\
E\phi_1 W \phi_2 &\equiv (E\phi_1 U \phi_2) \vee EG(\phi_1) \\
A\phi_1 W \phi_2 &\equiv (A\phi_1 U \phi_2) \vee AG(\phi_1)
\end{aligned}$$

Furthermore, because the explicit quantification over points in time can be confusing in larger formulas, TCTL defines the following abbreviations over the intervals I and I':

$$\begin{aligned}
E\phi_1\ {}_{I'}U_I\ \phi_2 &\equiv x.E(y.(y \in I' + x \rightarrow \phi_1))U(z.(\phi_2 \wedge z \in I + x)) \\
A\phi_1\ {}_{I'}U_I\ \phi_2 &\equiv x.A(y.(y \in I' + x \rightarrow \phi_1))U(z.(\phi_2 \wedge z \in I + x))
\end{aligned}$$

This abbreviation carries over to any temporal operator based on the U-operator.

Extending even further, TCTL allows formulas of the form $E\phi_1 U_{\approx c}\, \phi_2$ and $A\phi_1 U_{\approx c}\, \phi_2$ where $\approx \in \{<, \leq, >, \geq\}$ is a relational operator and $c \in \mathbb{N}^+$. In the following, we give only the definition for $E\phi_1 U_{\approx c}\, \phi_2$, as $A\phi_1 U_{\approx c}\, \phi_2$ is defined analogical:

$$\begin{aligned}
E\phi_1 U_{<c}\, \phi_2 &\equiv E\phi_1\ {}_{[0,c)}U_{[0,c)}\ \phi_2 \\
E\phi_1 U_{\leq c}\, \phi_2 &\equiv E\phi_1\ {}_{[0,c]}U_{[0,c]}\ \phi_2 \\
E\phi_1 U_{>c}\, \phi_2 &\equiv E\phi_1\ {}_{(c,+\infty)}U_{(c,+\infty)}\ \phi_2 \\
E\phi_1 U_{\geq c}\, \phi_2 &\equiv E\phi_1\ {}_{[c,+\infty)}U_{[c,+\infty)}\ \phi_2
\end{aligned}$$

Again, this abbreviation carries over to any temporal operator based on the U-operator.

Note that [60] also allowed = as relational operator, but [61] showed that this is one of the two reasons for undecidability of the satisfiability problem of TCTL-formula. Therefore Konrad et al. did not use this operator in [5], and hence we do not give its definition here.

3.1.3 Background: Requirement Pattern Systems

A *requirement pattern* is an abstract description of a class of recurring requirements. The principal idea behind such classifications originated from the success of design patterns [62] in object-oriented programming: Design patterns classify different kinds of recurring programming problems and provide expert knowledge in form of solutions to those problems. A design pattern describes scenarios, in which it can be applied and explains the specific aspects of the problem in an easy understandable way to a broad audience of developers.

Requirement patterns have the same objective: They also provide expert knowledge, but for formulating requirements. The idea here is to give descriptions of recurring system behaviors and let the user select the appropriate one. In turn the user gets expressions in one or more formal logics to precisely describe the desired behavior and is thus freed from the task of finding the right formal expression.

A *requirement pattern system* is a structured collection of requirement patterns. The main purpose of the system is to support the user in selecting the right pattern for a given problem. It may provide a flow-chart with different questions or a classification, which allows the browsing and/or filtering of the pattern collection.

We already described (see Chapter 2) that our requirements are captured with the help of such a pattern system and that we use the system described by Konrad et al. [5], which is an extension of [58] by Dwyer et al.

Such systems are not only helpful for the user, they also allow an easier and more streamlined transformation from the formal domain requirements to the input of the verification program. We explain this in more detail in Section 3.3. For now, lets have a closer look at a requirement pattern.

Every pattern is defined with respect to some *scope*. A scope defines how the pattern instantiation relates to system states. There are five different scopes:

- *Globally* means the requirement must be fulfilled under every circumstance.
- *Before* Q means the requirement must be fulfilled until Q holds in a system state.

- *After* Q means the requirement must be fulfilled forever after any system state in which Q holds.
- *Between* Q *and* R means the requirement must be fulfilled in between any two system states, from which Q holds in the first state and R holds in the second state. In other words, if $AG(Q) \wedge AG(\neg R)$ holds, then the requirement does not need to be met.
- *After* Q *until* R means the requirement must be fulfilled after any system state in which Q holds, until a system state in which R holds. Therefore, if $AG(Q \wedge \neg R)$ holds, then the requirement has to be met in all states except the first.

- **Pattern Name and Classification**
  Bounded Recurrence: Periodic Real-time Specification Pattern
- **Structured English Specification**
  scope *", it is always the case that"* P *" holds at least every "* c *" time unit(s)."*
- **Pattern Intent**
  This pattern describes the periodic satisfaction of a propositional formula. Intuitively, it captures the property that in every c time unit(s), the proposition P has to hold at least once. The proposition P holding more often than every c time units or holding continuously is considered a correct behavior in this pattern.
- **Real-time Temporal Logic Mappings**

$$\text{TCTL:} \begin{cases} \text{Globally :} & AG(AF_{\leq c}\, P) \\ \text{Before } R : & A(((F_{\leq c}\, (P \vee R)) \vee AG(\neg R))WR) \\ \text{After } Q : & AG(Q \rightarrow AG(AF_{\leq c}\, P)) \\ \text{Between } Q \text{ and } R : & AG((Q \wedge \neg R) \rightarrow A(((AF_{\leq c}\, (P \vee R)) \vee AG(\neg R))WR)) \\ \text{After } Q \text{ until } R : & AG((Q \wedge \neg R) \rightarrow A((AF_{\leq c}\, (P \vee R))WR)) \end{cases}$$

- **Examples and Known Uses**
  This pattern is commonly used in embedded systems, as these systems commonly perform periodic tasks. For example, a watchdog has to become active at least every c time unit(s) and verify that certain system constraints are not violated. Additionally, embedded systems often have to perform specific services periodically, such as sending a heart beat (a message denoting that the embedded system is functioning correctly) to other embedded systems using a communication device, e.g., a controller area network (CAN) bus.
- **Relationships**
  The untimed version of the bounded recurrence property, expressed as $\square(\diamond P))$ in LTL, can be found in several publications (such as [63]). It is commonly used to specify the absence of non-progress cycles in a system.

Figure 3.1: Bounded Recurrence Requirement Pattern [5].

Now consider the structuring of the bounded recurrence pattern shown in Figure 3.1:

- *Pattern Name and Classification* shows the name and the classification according to the pattern system. The classification aids the user in selecting the appropriate pattern, as it groups different system behaviors. Figure 3.2 and 3.3 show this classification.

- *Structured English Specification* is a representation of the requirement in *structured* English. Table 3.3 shows this structure for all requirement patterns provided by Konrad et al.
- *Pattern Intent* is a prose description of the system behavior the pattern captures.
- *Real-time Temporal Logic Mappings* contains the representations in different formal logics (we only show TCTL, but Konrad et al. provides mappings for other logics as well). They contain free atomic propositions (in the example P, R, and Q) that have to be given by the user.
- *Examples and Known Uses* provides concrete examples of system behaviors where this pattern could be successfully applied.
- *Relationships* contains notes to literature and , if applicable, the relationship to other patterns in the same system.

After a user finds a suitable pattern through reviewing the pattern system, he can now write down the adequate formula by looking up the real-time temporal logic mapping and instantiating the given formula with the necessary atomic propositions.

Principally, for our problem the main benefit of requirement pattern systems is, that they enable a non-expert user to formalize requirements. In which formal language the pattern system maps, is in contrast subordinate, as it is sufficient that the formalism has precisely defined semantics. Of course this could change if we decided to perform analysis on the requirements itself, i.e. if we wanted to check them for consistency (e.g. by a satisfiability check of the conjunction of formal requirements). For now, it is only important that the formal logic can be translated to specifications the verification method can understand.

Name	**Description**	**CTL Formula**
Absence	P is false.	$AG(\neg P)$
Existence	P becomes true.	$AF(P)$
Bounded Existence	Transitions to P-states occur at most 2 times.	$\neg EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P))))))$
Universality	P is true.	$AG(P)$
Precedence	S precedes P.	$A(\neg P\ W\ S)$
Response	S responds to P.	$AG(P \rightarrow AF(S))$
Precedence Chain 1-2	P precedes S,T.	$\neg E(\neg P\ U\ (S \wedge \neg P \wedge EX(EF(T))))$
Precedence Chain 2-1	S,T precedes P.	$\neg E(\neg S\ U\ P) \wedge \neg E(\neg P\ U\ (S \wedge \neg P \wedge EX(E(\neg T\ U(P \wedge \neg T)))))$
Response Chain 1-2	S,T responds to P.	$AG(P \rightarrow AF(S \wedge AX(AF(T))))$
Response Chain 2-1	P responds to S,T.	$\neg EF(S \wedge EX(EF(T \wedge EG(\neg P))))$
Constrained Chain Patterns	S,T without Z responds to P.	$AG(P \rightarrow AF(S \wedge \neg Z \wedge AX(A(\neg Z\ U\ T))))$

Table 3.1: Requirement patterns for scope *globally* by Dwyer et al. [58]

Name	**Description**	**TCTL Formula**
Minimum Duration	If P is true, it remains true for at least c time units.	$AG(P \vee A(\neg P\ W\ AG_{\leq c}(P)))$
Maximum Duration	If P is true, it is true for at most c time units.	$AG(P \vee A(\neg P\ W\ (P \wedge AF_{\leq c}(\neg P))))$
Bounded Recurrence	P holds at least once in every interval c time units long.	$AG(AF_{\leq c}(P))$
Bounded Response	If P holds, then S holds at least once after at most c time units.	$AG(P \rightarrow AF_{\leq c}(S))$
Bounded Invariance	If P holds, then S holds from that moment on for at least c time units.	$AG(P \rightarrow AG_{\leq c}(S))$

Table 3.2: Requirement patterns for scope *globally* by Konrad et al. [5]

We already noted that what can be checked and what not depends on the selected program verifier. VCC and the other tools we considered cannot check real-time properties, therefore all patterns in Table 3.2, which rely on quantitative real-time bounds, and their respective counterparts in the other scopes cannot be checked with our approach. For the patterns shown in Table 3.1 it is not clear: The absence- and universality pattern with scope *globally* are verifiable, but the others have to be examined more thoroughly before we can give a definite answer.

Because such an examination is out of scope for this work, we concentrate on the absence- and universality pattern.

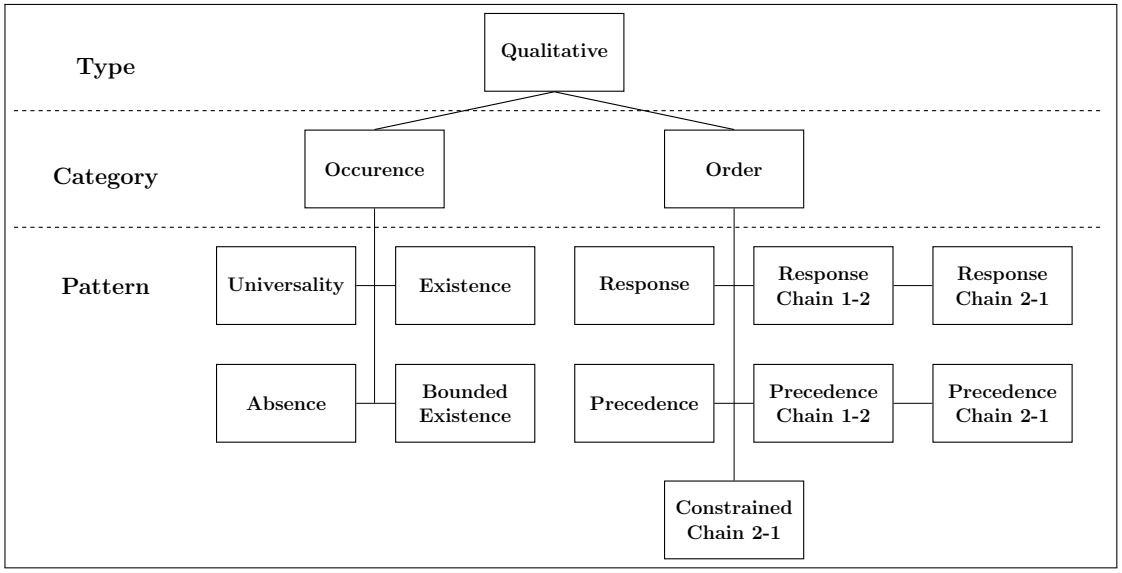

Figure 3.2: Requirement pattern classification by Dwyer et al. [58].

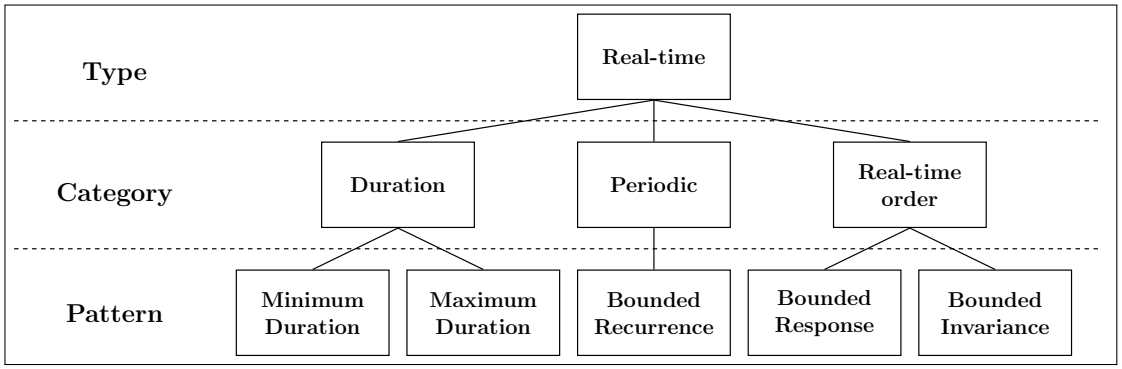

Figure 3.3: Additional classifications by Konrad et al. [5].

Start	property	::=	scope *","* specification *"."*
Scope	scope	::=	*"Globally"* | *"Before "* R | *"After "* Q | *"Between "* Q *" and "* R | *"After "* Q *" until "* R
General	specification	::=	qualitativeType | realtimeType
Quali -tative	qualitativeType	::=	occurenceCategory | orderCategory
	occurrenceCategory	::=	absencePattern | universalityPattern | existencePattern | boundedExistencePattern
	absencePattern	::=	*"it is never the case that "* P *" holds"*
	universalityPattern	::=	*"it is always the case that "* P *"holds"*
	existencePattern	::=	P *" eventualy holds"*
	boundedExistencePattern	::=	*"transitions to states in which "* P *"holds occur at most twice"*
	orderCategory	::=	*"it is always the case that if "* P *"holds"* (precedencePattern | precedenceChainPattern1-2 | precedenceChainPattern2-1 | responsePattern | responseChainPattern1-2 | responceChainPattern2-1 | constrainedChainPattern1-2)
	precedencePattern	::=	*", then "* S *" previously held"*
	precedenceChainPattern1-2	::=	*" and is succeeded by "* S *", then "* T *" previously held"*
	precedenceChainPattern2-1	::=	*", then "* S *"previously held and was preceded by "* T
	responsePattern	::=	*", then"* S *" eventually holds"*
	responseChainPattern1-2	::=	*", then"* S *" eventually holds and is succeeded by "* T
	responseChainPattern2-1	::=	*" and is succeeded by "* S *", then "* T *" eventually holds after "* S
	constrainedChainPattern1-2	::=	*" then "* S *" eventually holds and is succeeded by "* T *", where "* Z *" does not hold between "* S *" and "* T
Real -time	realtimeType	::=	*"it is always the case that "* (durationCategory | periodicCategory | realtimeOrderCategory)
	durationCategory	::=	*"once "* P *"becomes satisfied, it holds for "* (minDurationPattern | maxDurationPattern)
	minDurationPattern	::=	*"at least "* c *" time unit(s)"*
	maxDurationPattern	::=	*"less than "* c *" time unit(s)"*
	periodicCategory	::=	P *" holds "* boundedRecurrencePattern
	boundedRecurrencePattern	::=	*"at least every "* c *"time unit(s)"*
	realtimeOrderCategory	::=	*"if "* P *" holds, then "* S *" holds "* (boundedResponsePattern | boundedInvariancePattern)
	boundedResponsePattern	::=	*"after at most "* c *" time unit(s)"*
	boundedInvariancePattern	::=	*"for at least "* c *" time unit(s)"*

Table 3.3: Structured English grammar for the requirement pattern system by Konrad et al. [5]

3.1.4 Signature of Formal Requirements

This section explains how we construct one of the sets used for the explicit mapping between elements of the domain and their counterparts in the software. First, consider the following informal domain requirements:

R_1: If the sensor measures a temperature above 50°C, an alarm has to be sent within 50ms.

Assume a user formalizes the requirements by using the appropriate patterns from the requirement pattern system described in Section 3.1.3. This yields the following formal domain requirements expressed as TCTL formulas:

F_1 : $AG((\text{temperature above } 50°\text{C} \rightarrow AF_{\leq 50ms}(\text{alarm sent})))$[1]

[1]Bounded response pattern with scope *globally*.

We can see that in F_1 some informal remains are left; namely *"temperature above 50° C"* and *"alarm sent"*. We call such remains *domain phenomena*. We can now define the *signature* of the component requirements by collecting all domain phenomena from the set of formal component requirements and declare them atomic propositions. However, we also allow a further refinement of the domain phenomena by introducing additional symbols:

- A predicate symbol for *"above"*, for example $>(x, y)$.
- Function symbols for *"50° C"* and *"temperature"*, for example $C(50)$ and $Tempr$.

If we replace the domain phenomena in F_1 with the new symbols, we get the following formula F_2:

$$F_2 : AG(>(Tempr, C(50)) \rightarrow AF_{\leq 50ms}(alarm\ sent)))$$

With the formula F_2 we get the signature Sig_{F_2} shown in Table 3.4. In general, we allow such *state formulas* instead of atomic propositions. A state formula is a propositional formula constructed from logic operators $\wedge$, $\vee$, $\rightarrow$, $\neg$ and predicate and function symbols. State formulas may be constructed by the user to support, e.g. the automatic inference of dependencies between atomic propositions in the requirements, but they are not necessary.

As far as TCTL is concerned, every state formula behaves exactly like an atomic proposition; it can assume either `false` or `true` for every state $s \in S$. The

Symbol	**Preliminary interpretation**
$>(x, y)$	*"x is above y"*
$C(50)$	*"50° C"*
$Tempr$	*"temperature"*
alarm sent	*"alarm sent"*

Table 3.4: Signature Sig_{F_2} of formula F_2

signature of the formal domain requirements provides us with a set of symbols that we can use to define the mapping between the domain on one side and software and hardware on the other side. Because we do not know yet, how the soft- and hardware side looks, the interpretation in Table 3.4 is preliminary: The appropriate semantics has to be given by the combination of hard- and software.

3.2 Connecting Software and the Real World

3.2.1 The Interface between Requirements and Software

Our next step is the creation of a relation between domain phenomena and *program fragments*. We call this relation *Interface between Requirements and Software*, I_{RS}. As we restricted ourselves to C programs (see Section 1.2), our program fragments are C fragments.

Definition 3. *Let P be a C program and τ be a valid type with respect to P. A program fragment f_P is defined as follows:*

$$f_p := \begin{cases} v : \tau & \text{if } v \text{ is a declared variable in } P \text{ of type } \tau. \\ m :: a_1 : \tau_1 \dots a_n : \tau_n \to \tau & \text{if } m \text{ is a declared function in } P \text{ and } a_1 : \tau_1 \dots a_n : \tau_n \text{ are parametrizing variables for } n \geq 0 \text{ and } \tau \text{ is the type of the return value of } m. \\ exp :: \tau & \text{if } exp \text{ is a valid expression of type } \tau \text{ in } P. \end{cases}$$

Additionally, the set F_P contains all program fragments f_p in P.

The relation I_{RS} has to be created manually by the responsible developer, but as he already has to be familiar with the domain phenomena to be able to develop the software, this should pose no difficulty. If something is manipulated or observed by the software, it is expressible as fragment of the program code, i.e. there exist program fragments for every domain phenomenon. Those fragments can be found at the interface between software and hardware. A developer can sequentially go through the list of domain phenomena (provided by the signature of the formal domain requirements Sig_R) and select the appropriate program fragment for each domain phenomenon. The sequential processing further reduces the complexity for the developer. Our relation I_{RS} is similar to the relations IN and OUT of the four variable model (see Chapter 2): It also captures the correspondence between the environment (called domain by us) and the software. But while the four variable model introduced abstract variables, we directly use program fragments for our relation.

Because it is later beneficial to already have a distinction between input and output (see Section 3.3.1 why), we also want to have a classification of program fragments in input, output and auxiliary fragments. The classification has to be given manually by the user and is defined as the set $Cl := \{IN, OUT, AUX\}$ where the classification is given according to the following rules:

- A variable or function is classified as IN if it is directly mapped to the underlying abstraction layer, i.e. the hardware, via direct memory mapping

or interrupt vectors, that serves as input to the software. An input to the software may change its value non-deterministically and must not be written to by the software.

- A variable or function is classified as OUT if it is directly mapped to the hardware that serves as output of the software. An output must not be read by the software.
- A program fragment that cannot be classified as IN or OUT is classified as AUX.

Symbols from Sig_{F_2}	**Program fragments from F_P**	**Signature or type**	**Classification Cl**
$>(x, y)$	`>`	`> :: char -> char -> bool`	AUX
$C(50)$	`convertValue(50)`	`50 : int`	AUX
		`convertValue :: int -> char`	AUX
$Tempr$	`k`	`k : char`	IN
Alarm sent	`snd_buf[0] == 2 && snd_buf[1] == 0 && snd_buf[2] == 2 && snd_buf[3] == 4 && CTS == 1`	`snd_buf : char[]`	OUT
		`CTS : bit`	OUT

Table 3.5: The interface between requirements and software is a relation between symbols from the signature of the formal domain requirements and program fragments.

For our example component requirement F_2, the relation I_{RS} could look like shown in Table 3.5. We said earlier that the symbols come from the signature of the formal domain requirements. But where do the program fragments come from?

If we already had a list of available inputs and outputs to begin with, we could choose appropriate program fragments by either choosing variables or functions directly from this list or formulating expressions over them. The next section describes how such a list can be obtained.

3.2.2 Software Description

A *software description* classifies memory-mapped variables and interrupt handler functions of a program as inputs and/or outputs. It has to be created manually by the developer of the software and is essentially a one-sided interface description: It describes the interface between software and hardware from the software's

point of view. We expect that this description can be easily given either during the actual development of the software as well as afterwards; the examples of real-world software available to us already defined their inputs and outputs in separate header files, so that they are readily available. Even if this is not the case, developers of the software need to know this interface and should be able to provide the necessary information.

Names N_P	**Signature** Sig_P	**Addresses** A_P	**Classification** Cl
`k`	`a : char`	`0xFF00`	IN
`intervall_timer`	`intervall_timer :: -> void`	`0x14`	IN
	...		
`snd_buf[0]`	`snd_buf[0] : char`	`0xFF02`	OUT
`snd_buf[1]`	`snd_buf[1] : char`	`0xFF04`	OUT
`snd_buf[2]`	`snd_buf[2] : char`	`0xFF06`	OUT
`snd_buf[3]`	`snd_buf[3] : char`	`0xFF08`	OUT
`CTS`	`CTS :: bit`	`0xFF29`	OUT

Table 3.6: An example of a software description.

Table 3.6 shows an example of a software description. It gives the name of a variable or function and its type or signature, its memory address and its classification per row. A developer can create such a description easily by looking up the declarations of the memory-mapped variables and interrupt functions. We define a software description as follows:

Definition 4. *Let P be a C program. $D_{SW} \subseteq \{N_P \times Sig_P \times A_P \cup \{-\} \times Cl\}$ is a software description, where*

- *N_P is the set of names of variables and functions of P.*
- *Sig_P is the set of variable types and function signatures of P.*
- *A_P is the set of explicit memory addresses and interrupt vectors used in P. $\{-\}$ is used for instances where there is no hardware counterpart, i.e. for auxiliary variables or functions.*
- *$Cl = \{IN, OUT, AUX\}$.*

With the software description, we can now define the relation I_{RS} as follows:

Definition 5. *Let P be a C program and R be a set of formal domain requirements. $I_{RS} \subseteq Sig_R \times F_P$ is a relation between domain phenomena and program fragments, where*

- *Sig_R is the signature of R.*
- *F_P is the set of program fragments of P.*

Now lets recall our informal domain requirement from the beginning of Section 3.1.4:

R_1: If the sensor measures a temperature above 50°C, an alarm has to be sent within 50ms.

If we have a software description and the I_{RS} we can now say that the domain phenomenon *"temperature"* from R_1 corresponds to the software input variable `k` of type `char` at memory address `0xFF73`. With the memory address we can now take the next step by describing the other side of the interface between hardware and software.

3.2.3 Hardware Description

The hardware defines the available inputs and outputs to our software. But a hardware port is also a connection between the hardware and the domain. With a list of all available hardware ports we can map domain phenomena to them and together with the relation I_{SH} we can ensure, that the software is connected to the right hardware ports for the right domain phenomena.

Because in typical embedded system developments, the hardware is provided by a third-party manufacturer, we can rely on an already existing documentation in form of comprehensive specifications. Usually, large vendors provide an exhaustive documentation of their micro-controllers, which includes tables listing all input and output ports as well as interrupt addresses. We are particularly interested in the mapping between hardware ports or interrupts and register addresses or interrupt vectors, and we call such a mapping a *hardware description*.

Definition 6. *The relation $D_{HW} \subseteq N_{HW} \times A_{HW}$ is a hardware description where*

- *N_{HW} is the set of names of hardware ports and interrupts.*
- *A_{HW} is the set of memory addresses of the hardware.*

Address	**Special Function Register (SFR) Name**	**Symbol**	**...**
FF00H	Port 0	P0	...
FF02H	Port 2	P2	...
FF04H	Port 4	P4	...
FF14H	A/D conversion result register 0	ADCR0	...
FF15H			
...	...	...	...

Table 3.7: A fragment of the special function register list from the user manual of an 8-bit micro-controller [64].

Table 3.7 shows an excerpt of the special function register list from the user manual [64] of an 8-bit micro-controller. The columns *Address* and *SFR Name* provide the mapping between memory addresses or interrupt vectors and an hardware port. Because the software has to reference variables to this memory addresses to use the hardware inputs and outputs, we can now precisely define the interface between hardware and software, both in terms of hardware ports and interrupts or variables and function names. The interface between hardware and software is defined as follows:

Definition 7. *Let D_{SW} be a software description and D_{HW} a hardware description. The relation $I_{SH} \subseteq D_{SW} \times D_{HW}$ is the interface between hardware and software and*

$$(a, b) \in I_{SH} \text{ iff } \pi_{A_P}(a) = \pi_{A_{HW}}(b).$$

3.2.4 The Interface between Requirements and Hardware

So far we have described the relation I_{RS} from domain phenomena to program fragments and the relation I_{SH} from software to hardware. Now it is easy to define a third relation between domain phenomena and hardware called the *Interface between Requirements and Hardware*:

Definition 8. *Let Sig_R be the signature of the formal domain requirements R and D_{HW} be a hardware description and $(a, i) \in I_{RS}$.*
The relation $I_{RH} \subseteq Sig_R \times D_{HW}$ is an interface between requirements and hardware. I_{RH} is constructed as follows:
If i has the form

- $v : \tau$, *then* $(a, b) \in I_{RH}$ *if* $\exists (i', j') \in \pi_{N_P, N_{HW}}(I_{SH}).i' = v \wedge j' = \pi_{N_{HW}}(b)$
- $m :: a_1 : \tau_1 \ldots a_n : \tau_n \to \tau$ *then*
  $\exists (i', j') \in \pi_{N_P, N_{HW}}(I_{SH}).i' = m \wedge j' = \pi_{N_{HW}}(b)$

- $exp : \tau$ *then* $\forall v : \tau, m :: a_1 : \tau_1 \ldots a_n : \tau_n \rightarrow \tau \in exp.$
  $((\exists (i', j') \in \pi_{N_P, N_{HW}}(I_{SH}).(i' = v \vee i' = m) \wedge j' = \pi_{N_{HW}}(b)) \rightarrow (a, b) \in I_{RH})$

Informally, I_{RH} relates domain phenomena to hardware names through the occurrences of variables or functions in the program fragments of I_{RS}. If the variables or functions are classified as IN or OUT, we can look up the hardware counterpart in the relation I_{SH}.

I_{RH}							
I_{RS}		I_{SH}					
		D_{SW}				D_{HW}	
Sig_R	F_P	N_P	Sig_P	A_P	Cl	A_{HW}	N_{HW}
$>(x, y)$	`>`	`>`	`> :: char -> char -> bool`	–	AUX		
$C(50)$	`convertValue(50)`	`50`	`50 : int`	–	AUX		
		`convertValue`	`convertValue :: int -> char`	–	AUX		
$Tempr$	`k`	`k`	`k : char`	`0xFF00`	IN	`0xFF00`	Port 0
$Alarm\ sent$	`snd_buf[0] == 2`	`snd_buf[0]`	`snd_buf[0] : char`	`0xFF02`	OUT	`0xFF02`	Port 2
	`&& snd_buf[1] == 0`	`snd_buf[1]`	`snd_buf[1] : char`	`0xFF04`	OUT	`0xFF04`	Port 4
	`&& snd_buf[2] == 2`	`snd_buf[2]`	`snd_buf[2] : char`	`0xFF06`	OUT	`0xFF06`	Port 6
	`&& snd_buf[3] == 4`	`snd_buf[3]`	`snd_buf[3] : char`	`0xFF08`	OUT	`0xFF08`	Port 8
	`&& CTS == 1`	`CTS`	`CTS : bit`	`0xFF29`	OUT	`0xFF29`	Port mode register 9

Table 3.8: The relations I_{RS}, I_{SH} and I_{RH} together.

We can use this relation to validate if the software is correctly "wired" to the hardware. Table 3.8 shows the relations I_{RS}, I_{SH} and I_{RH} together in one view. As we will see in the next section, the relation I_{RS} already provides us with enough information to verify if the software is correct with respect to the formal domain requirements. But the additional relations I_{SH} and I_{RH} allow someone familiar with the hardware and the requirements to decide, if the domain phenomena correspond to the correct hardware ports. Errors in the I_{RS} for example if a domain phenomena is mapped to the wrong part of the software, are hard to find without having a second source of validation. Moreover, those errors can only be found by someone who is familiar with the software. In SMEs this could mean, that the same person that made the error has to find it. We know from experiences in software testing [65], that this is not a good idea; developers are biased towards their software and have a higher probability of overlooking their own errors [66].

The relation I_{RH} can be used to find errors in the relation I_{RS}. Such an error could be, for example, that an input domain phenomena is connected to a variable that is no input and has always a fixed value. If the variable is coincidentally initialized with a value that satisfies the original requirement, we see a positive result from the program verifier, but the specification derived from the domain requirements is wrong. Even more, in certain scenarios it could even be beneficial for someone to insert wrong mappings in the I_{RS} for example to finish the job faster. Because I_{RH} relates hardware and domain requirements, we can validate the correctness of the I_{RS} by checking, if I_{RH} actually relates the right domain phenomena to the right hardware ports. Because I_{RH} is constructed from the

other relations I_{RS} and I_{SH}, any error made in those relations is carried over to I_{RH}. The benefit here is, that one does not need to be familiar with the software to check this relation, it is sufficient to know the hardware. This enables, for example, the hardware developer to review the work of his colleagues.

3.3 Program Verification

3.3.1 Software Specification

After obtaining all the necessary connections between domain phenomena and software, we can express the formal domain requirements in terms of the software. We take every domain requirement from the set of domain requirements and replace the occurring domain phenomena with the program fragments given through the relation I_{RS}. Let us recall the domain requirement from Section 3.1.4 in which we already replaced some of the domain phenomena with state formulas:

$$F_2: \quad AG(>(Tempr, C(50)) \rightarrow AF_{\leq 50ms}(alarm\ sent)))$$

If we look up the I_{RS} in Table 3.5 and substitute the corresponding program fragment for every symbol, we get:

$$\begin{aligned} SWS_2: \quad & AG(\texttt{k} > \texttt{convertValue(50)} \rightarrow AF_{\leq 50ms}(\texttt{snd\_buf[0]} == \texttt{2} \\ & \&\& \ \texttt{snd\_buf[1]} == \texttt{0} \ \&\& \ \texttt{snd\_buf[2]} == \texttt{2} \\ & \&\& \ \texttt{snd\_buf[3]} == \texttt{4} \ \&\& \ \texttt{CTS} == \texttt{1})) \end{aligned}$$

The substitution has to be done for every domain requirement in the set of domain requirements. We call the resulting set the *software specification* $Spec_{SW}$.

The resulting formulas now contain only state formulas over predicate and function symbols, which are elements of the software description. Those elements of the software description also contain a classification in IN, OUT and AUX. This classification now carries over to the new formulas: If every element in the formula is classified as IN, the formula does not describe a specification, but rather an assumption about the environment of the system. We call such formulas *environmental assumptions*. A program verifier cannot show that an environmental assumption is valid, but it can use the assumption in the proof of the remaining elements in $Spec_{SW}$.

If a program verifier accepts TCTL-formula as input, we are finished at this point and can run the verifier on the program P together with the set $Spec_{SW}$. But unfortunately, TCTL is not the specification language of choice for program verifiers.

In principal, which tool we use to verify the program does not matter as long as it matches our criteria (see Chapter 2). For the VCC, we need to transform TCTL or CTL formula to annotations of the program. Before we can think about translating the formulas to those annotations, we have to take a look at the VCC and its annotation-language.

3.3.2 Program Verifier

VCC provides a syntax for the specification of type invariants and function contracts with pre- and post-conditions [17]. It further allows to define specification code, which can be used to capture information not directly available in the code, e.g. the history of a variable.

Function contracts are defined through the following keywords:

- `requires` states which condition has to hold if the method is called.
- `ensures` states which condition has to hold if the method returns.
- `maintains` is the combination of `requires` and `ensures` in one condition.
- `writes` defines what parts of the program state can be modified by the method.

Type invariants are stated with the `invariant` keyword and are checked whenever the object is wrapped (with the `wrap` keyword) or unwrapped (with `unwrap`). Figure 3.4 shows a simple program with a type invariant: The struct `point` has two fields of type `int`, namely `x` and `y`. We define an invariant over those fields with the keyword `invariant` inside the structs definition. The invariant states that `x` and `y` have to be larger or equal to 0 and less or equal to 1000. The function `funA` takes a reference to a struct of type `Point`, `A`, and tries to write in one of its fields. Because it requires that `A` is wrapped, it can assume that the invariant holds at function entry. VCC now ensures in every call to `funA` that the type invariant for `Point` holds. In Figure 3.4(a) we violate the type invariant on purpose and VCC reports the violation as expected. We can use type invariants to express environmental assumptions over input variables.

```
#include "vcc.h"

struct Point {
  int x;
  int y;
  invariant(x>=0 && y>=0
    && y<=1000 && x<=1000)
};

void funA(struct Point* A)
writes(A)
maintains(wrapped(A))
{
  unwrap(A);
  A->y = 1001;
  wrap(A);
}
```

(a) Invariant(y<=1000) of Point fails on wrap.

```
#include "vcc.h"

struct Point {
  int x;
  int y;
  invariant(x>=0 && y>=0
    && y<=1000 && x<=1000)
};

void funA(struct Point* A)
writes(A)
maintains(wrapped(A))
{
  unwrap(A);
  A->y = 1000;
  wrap(A);
}
```

(b) Verification succeeded.

Figure 3.4: Type invariants with VCC.

Whether we specify it or not, VCC verifies that the program does not violate memory safety properties. In contrast to modern object-oriented languages like Java or C#, C is not type-safe, i.e. objects are a chunk of memory at an address and a type defines the length for this chunk. The VCC memory model addresses the resulting problems like aliasing or disjointness of objects, but requires in turn the explicit annotation of every possible write of a function [67]. Therefore, before we can verify if the software is correct with respect to the domain requirements, we have to annotate the program with the correct `writes` clauses. For structs and unions we also need the `maintains(wrapped(struct))` and wrap and unwrap statements before and after every write. This leads to a considerable increase in size of the actual program. Figure 3.5 shows an example program without annotations, Figure 3.6 shows the same program after adding all the necessary annotations. In this example, the size in SLOC was increased through the annotations by a factor of 1.66. The overhead through annotations is further discussed in Section 6.1.

In function `main()` in Figure 3.6 we can see a couple of other things:

- The function requires `program_entry_point()`, which states that this is the entry point of the program and all global variables and structures are mutable at this point.

- The function is also allowed to write to the whole heap, which is specified with `write(set_universe())`.
- The `while`-loop in function `main` has to be annotated with loop-invariants that state which changes to the program state have been made during the execution of the loop. VCC loop invariants have to hold at the loop entry and the loop exit, but not in between.

```
#include "vcc.h"

struct _Inputs {
  volatile int x;
  volatile int y;
};

struct _Outputs {
  int x;
  int y;
};

struct _Inputs Input;
struct _Outputs Output;

void SignalAlarm(){
  Output.x=1;
}
```

```
void ResetAlarm(){
  Output.x=0;
}

void main(){
  Input.x = 0;
  Input.y = 0;
  Output.x = 0;
  Output.y = 0;

  while(1){
    if(Input.x >=100){
      SignalAlarm();
    }
    if(Input.y >0){
      ResetAlarm();
    }
  }
}
```

Figure 3.5: An example program without annotations. Verification with VCC will fail because it cannot be proven that the structs are writable.

```
#include "vcc.h"

struct _Inputs {
  volatile int x;
  volatile int y;
};

struct _Outputs {
  int x;
  int y;
};

struct _Inputs Input;
struct _Outputs Output;

void SignalAlarm()
writes(&Output)
maintains(wrapped(&Output))
{
  unwrap(&Output);
  Output.x=1;
  wrap(&Output);
}
```

```
void ResetAlarm()
writes(&Output)
maintains(wrapped(&Output)){
  unwrap(&Output);
  Output.x=0;
  wrap(&Output);
}

void main()
writes(set_universe())
requires(program_entry_point()){
  Input.x = 0;
  Input.y = 0;
  Output.x = 0;
  Output.y = 0;

  wrap(&Output);
  wrap(&Input);

  while(1)
  invariant(wrapped(&Input))
  invariant(wrapped(&Output)) {
    unwrap(&Input);
    if(Input.x >=100){
      SignalAlarm();
    }
    if(Input.y >0){
      ResetAlarm();
    }
    wrap(&Input);
  }
}
```

Figure 3.6: The example program from Figure 3.5 with the necessary annotations.

Our example program does not show another important type of annotations, the `assume()` and `assert()` statements. Their semantics is straight-forward:

- `assume(P)` states that `P` can be assumed to be true. VCC can then use `P` from that point on in the correctness-proof for the program.
- `assert(P)` states that `P` must be true at this program location. If `P` is not true, VCC will report that the `assert`-statement is violated and that the program could not be verified.

Now that we know the annotation language of VCC to some extend, we can start to encode the software specification in this language.

3.3.3 Machine-Level Specification

To verify a C program with the VCC we have to annotate the program code, such that the annotation corresponds to the software specification, i.e. we need a specification in the specification language provided by the program verifier.

Definition 9. *Let PV be a program verification tool, $Spec_{SW}$ be the set of software specifications and P be a program. A machine-level specification (MLS) MLS^P_{PV} is a specification for P with the following characteristics:*

- *MLS^P_{PV} is without further processing a valid input to the program verifier PV, i.e. MLS^P_{PV} must be error-free with respect to the syntax and semantics of the specification language defined by PV.*
- *If the program verifier PV reports, that P together with MLS^P_{PV} is correct and the relation I_{RS} is correct, then the program is correct with respect to the domain requirements expressed through the set of software specifications $Spec_{SW}$.*

We can now create such an MLS from the software-specification by creating a transformer for the formal logic in which the software-specification is written, in our example TCTL and CTL. Because we used requirement pattern systems to formalize our domain requirements, we can utilize the subset of the formal logic defined by the pattern system. We just have to define a transformer for each pattern. Such a transformer amounts to an observer for the program, i.e. specification code which observes the behavior of the software at the necessary points and switches in a designated state if the observed behavior is violating the specified property. Examples of observer-based approaches to formal verification of programs can be found in [68–71]. Because the construction of such observers for the whole requirement pattern system is out of scope for this work, we annotate the code manually.

The small example program from Figure 3.6 could be a implementation of a very simple embedded system: It has two inputs (`Input.x` and `Input.y`) and two outputs (`Output.x` and `Output.y`). The software sets one of those outputs to 1 whenever the value of the input `x` is greater or equal to 100, and to 0 when this is not the case.

Consider the following informal domain requirement:

R_1: The acoustic alarm signal may only be disabled by a user pressing the disable-button.

We formalize the requirement with the universality pattern from Table 3.1:

F_1 : AG(acoustic alarm signal disabled $\rightarrow$ user pressed disable-button)

Then we extract a software description from our sample program, which is rather easy because there are only four variables. Since the program is not mapped to any memory address, we simply omit them:

Names N_P	**Signature** Sig_P	**Addresses** A_P	**Classification** Cl
`Output.x`	`Output.x : int`	–	OUT
`Output.y`	`Output.y : int`	–	OUT
`Input.x`	`Input.x : int`	–	IN
`Input.y`	`Input.y : int`	–	IN

The relation between domain phenomena and software is also rather easy; we imagine the button is memory-mapped to `Input.y` and the acoustic alarm to `Output.x`:

Symbols from Sig_{F_1}	**Program fragments from** F_P
acoustic alarm signal disabled	`Output.x == 0`
user pressed disable button	`Input.y > 0`

With this relation we get the following software specification:

$$SWS_1 : \quad AG(\texttt{Output.x == 0} \rightarrow \texttt{Input.y > 0})$$

Because the formula contains input as well as output program fragments, we know that this not an environmental assumption but a valid software specification. Now we want to check if the program fulfills the specification or not. For the universality pattern, we can write an `assert(P)` statement at every program point that writes to one of the output (OUT) or auxiliary (AUX) program fragments used in the formula. In this case, `P` is `Output.x == 0` $\rightarrow$ `Input.y > 0`. Because VCC cannot infer pre- and post-conditions necessary for the correctness-proof of some functions, we further have to insert them if VCC fails to verify single functions in which we added `assert`-statements. For example, VCC failed to verify the correctness of the function `ResetAlarm()`. If we add the antecedence of the implication in our `assert`-statement as precondition, it succeeds. The annotated program is shown in Figure 3.7.

We can perform the verification task incrementally and thus supply the needed pre- and post-conditions. This ensures, that functions which do not actively change the program state such that a property holds but rather rely on changes in other functions, have the necessary preconditions. Conversely, if the function changes the program state such that the satisfaction of an `assert`-statement is ensured or violated, we have to insert the asserted predicate as post-condition.

If the verifier reports for (a) the program entry point or (b) a function where we cannot add more pre- and post-conditions, we found a valid error.

```
#include "vcc.h"

struct _Inputs {
  volatile int x;
  volatile int y;
};

struct _Outputs {
  int x;
  int y;
};

struct _Inputs Input;
struct _Outputs Output;

void SignalAlarm()
writes(&Output)
maintains(wrapped(&Output)){
  unwrap(&Output);
  Output.x=1;
  assert((Output.x == 0)
    ==> (Input.y > 0));
  wrap(&Output);

}

void ResetAlarm()
writes(&Output)
requires(Input.y > 0)
maintains(wrapped(&Output)){
  unwrap(&Output);
  Output.x=0;
  assert((Output.x == 0)
    ==> (Input.y > 0));
  wrap(&Output);
}
```

```
void main()
writes(set_universe())
requires(program_entry_point()){
  Input.x = 0;
  Input.y = 0;
  Output.x = 0;
  assert((Output.x == 0)
    ==> (Input.y > 0));
  Output.y = 0;

  wrap(&Output);
  wrap(&Input);

  while(1)
  invariant(wrapped(&Input))
  invariant(wrapped(&Output))
  {
    unwrap(&Input);
    if(Input.x >=100){
      SignalAlarm();
    }
    if(Input.y >0)
    {
      ResetAlarm();
    }
    wrap(&Input);
  }
}
```

Figure 3.7: The annotated example program. The inserted annotations are highlighted.

Now we can run the VCC to verify if the software is correct with respect to the domain requirements. But VCC reports, that the `assert`-statement in function `main` did not verify. Why is that? At this point, there are two choices: Either the program or the specification is incorrect.

If we inspect the program closer, we see that the assertion cannot hold at this program location because it is right after the initialization of the program; and, of course, when we start the system, the acoustic alarm is disabled and no user pressed the disable-button. While we wrote the requirements we thought about a running system and forgot, that there are special circumstances when the system is started the first time. To fix this, we can change the domain requirement and explicitly encode the environment:

R_1': The acoustic alarm signal may only be disabled by a user pressing the disable-button or when the system initializes.

We now have to redo the steps by providing a new formal domain requirement, which contains new domain phenomena and in turn needs an extended software description:

F_1' : AG(acoustic alarm signal disabled $\rightarrow$ (user pressed disable-button $\vee$ system initializes))

For the domain phenomena *system initializes* we have to introduce an auxiliary variable $init$: int that tracks where the system initializes. We add the tuple (system initializes, $init == 1$) to the I_{RS} and get a new software specification:

$$SWS_1' : \quad AG(\texttt{Output.x == 0} \rightarrow (\texttt{Input.y > 0} \vee \texttt{init == 1}))$$

Furthermore, the program has to be augmented by the additional code to observe the system initialization; for this purpose VCC provides the macros `spec()` and `speconly()`. `spec()` is used to declare specification variables or functions, while `speconly()` is used as guard for code fragments other than declarations. Figure 3.8 shows the modified program; the modifications are highlighted. If we run the VCC on this program, the verification succeeds.

```
#include "vcc.h"

struct _Inputs {
  volatile int x;
  volatile int y;
};

struct _Outputs {
  int x;
  int y;
};

struct _Inputs Input;
struct _Outputs Output;

spec(int init;)

void SignalAlarm()
writes(&Output)
maintains(wrapped(&Output)){
  unwrap(&Output);
  Output.x=1;
  assert((Output.x == 0)
    ==> ((Input.y > 0) || init == 1));
  wrap(&Output);
}

void ResetAlarm()
writes(&Output)
requires((Input.x >=100 || init == 1))
maintains(wrapped(&Output)){
  unwrap(&Output);
  Output.x=0;
  assert((Output.x == 0)
    ==> ((Input.y > 0) || init == 1));
  wrap(&Output);
}
```

```
void main()
writes(set_universe())
requires(program_entry_point()){
  speconly(init = 1;)
  Input.x = 0;
  Input.y = 0;
  Output.x = 0;
  assert((Output.x == 0)
    ==> ((Input.y > 0) || init == 1));
  Output.y = 0;
  speconly(init = 0;)

  wrap(&Output);
  wrap(&Input);

  while(1)
  invariant(wrapped(&Input))
  invariant(wrapped(&Output))
  {
    unwrap(&Input);
    if(Input.x >=100){
      SignalAlarm();
    }
    if(Input.y >0)
    {
      ResetAlarm();
    }
    wrap(&Input);
  }
}
```

Figure 3.8: The modified example program verifies successfully. The modifications are highlighted.

4 Case Study

This chapter provides a case-study of our approach with the goal of determining if it is applicable in real-world scenarios and what issues can arise during its application.

The chapter is structured as follows:

- Section 4.1 describes the setting in which the case-study was conducted, the source of our real-world example and some limitations resulting from this source.
- In Section 4.2 we discuss how informal domain requirements were structured and obtained. We also give an example of an informal domain requirement, which we use in the following for our case-study.
- The Section 4.3 explains how we used the requirement pattern systems to formalize the domain requirements and reports, which patterns have been used.
- In Section 4.4 we describe how we created the software description and how we made some observations regarding hardware and compiler characteristics.
- Section 4.5 and 4.6 are rather short; the first provides the relation between domain requirements and software and the second shows the resulting software specification.
- In Section 4.7 we explain in detail how we prepared the example source code such that it could be verified with VCC. We also explain what problems we had during this process and why we had to limit our expectations.
- The chapter ends with Section 4.8 in which we explain how the MLS was created. The section also shows the annotated part of the program necessary for our example and gives the results of the verification with VCC.

4.1 Setting

The case study uses the central unit of a radio-based fire alarm system as real-world example. The artifacts for the case-study come from a cooperation between the Department of Software Engineering of the University Freiburg and the company Security Care GmbH (SeCa) [72]. The cooperation receives funding from the ZIM program [73] of the German Ministry of Economy and Technology (BMWi) [74]. SeCa's main business areas are the development of radio-based fire alarm systems and contract development of high-frequency radio based embedded systems.

The goal of the cooperation between SeCa and our department is the development of a verified radio protocol which will be used in a new fire alarm system. To accomplish this goal, the department was given access to all available artifacts of the company, including the source code and the requirement documents of the previous system. All provided artifacts are covered by a non-disclosure agreement that prevents the accurate reproduction of those artifacts.

The previous system, called cc100, is a radio-based fire alarm system which consists of a central unit, multiple sensors and input/output devices as well as repeaters, all interconnected via high-frequency radio. For this case-study, we use the central unit, called F.BZ 100. The requirements we are about to see are partly from the company itself, partly from the European norm for fire detection and fire alarm systems [75, 76]. Because the European norm is protected under copyrights, we do not reproduce the original wording of the norm in this case-study, but rather some of the derived requirements concerning the behavior the central unit. We also used some of the requirements developed for the new system, as most of them are similar to the old ones, but more precise since we were able to discuss them extensively with the respective stakeholders.

4.2 Obtaining Domain Requirements

After analyzing the provided requirement documents and the EN 54 documents it soon became clear, that the precision necessary to formalize the requirements was not achieved in the documents itself. Furthermore, many of the terms used in the documents were unfamiliar to us, so we needed to (a) build a more structured variant of the documents, containing all necessary information and (b) seek explanations for the unclear notions we found during our preliminary analysis.

First we designed a structure to capture the informal domain requirements. The structure is shown in Figure 4.1. You can see from left to right and from top to

bottom the following fields[1]:

- **4.1 Beispiel (ID: T02)**: The title of the requirement (Beispiel) and the position in the document (4.1).
- **ID**: The unique identification number (*ID*) for the requirement. A unique ID is necessary to identify the requirements regardless of their ordering in different views of the document.
- **Kategorie**: The *category* captures, if the requirement is an environmental assumption, a non-functional requirement describing how a certain function should be implemented or a functional requirement, which describes what functionality should be implemented.
- **Priorität**: The *priority* of the requirement. This field was necessary to decide what importance the requirement had for the company; if the requirement conflicted with another, we could alter or drop the requirement with the lower priority.
- **Status**: The *status* of a requirement describes the stability it has already reached; when a requirement was first captured, it was assigned the status *draft*. After it entered the review process, the status changed to *discussion*. After the review was completed and every change was incorporated, the status changed to *stable*. The status *verified* is used to indicate that we can provide evidence that the system fulfills the requirement.
- **Version**: The *version* field automatically tracks changes to the requirement via the revision number of the file containing the requirement. It also shows who changed the requirement last and when. The revisions were tracked with the version control system Subversion [77].
- **Beschreibung**: The *description* contains the natural language description of this requirement.
- **Prüfung**: This field describes, which *tests* have to be performed to show that the system fulfills this requirement. SeCa wants to obtain a certification of compliance with the EN-54 norm from an accredited testing institute, therefore all requirements originating from that norm have to be tested independently by this institute. Furthermore, SeCa does not want to rely on the formal verification of the system alone, partly because it is not clear if the necessary level of dependability can be achieved, partly because tests of the system show other properties like usability as well, which cannot be shown by formal verification alone.

[1]The documents created in this cooperation are mainly in German. The following list therefore uses the German terms found in Figure 4.1; a translation is given by the first *emphasized* word in the description.

- **Konflikte**: The *conflicts* field lists all requirements that conflict with the current requirement and describe the conflict in detail.
- **Formalisierung**: The *formalization* field contains – if applicable – the formal domain requirement.
- **Kommentare**: The *comments* field contains unanswered questions or remarks for that requirement. It was mainly used to capture questions regarding certain domain-specific terms between the meetings with the company representatives.
- **Quellen**: The *source* field contains the source of the requirement, that is, from which page in which normative document comes this requirement, which stakeholder formulated the requirement, etc.
- **Abgeleitet**: The *derived*-field contains links to all requirements derived from the current one.
- **Übergeordnet**: The *superordinate*-field contains links to all requirements the current one is derived from.
- **Anwendungsfälle**: The *use-case* field contains links to the use-cases in which the requirements occur.

4 Anforderungen

4.1 Beispiel(ID: T02)

ID	Kategorie	Priorität	Status	Version
T02	funktional, nichtfunktional, Umgebungseigenschaft	Skala von 1=unwichtig bis 5=sehr wichtig	Entwurf, Diskussion, Stabil, Abgeschlossen, Verifiziert/Validiert	Revision 6717 vom 22.02.2010 um 10:03 durch dietsch.

Beschreibung

Die eigentliche Beschreibung der Anforderung.

Prüfung

Beschreibung eines oder mehrerer Prüfkriterien, die entscheiden, ob die Anforderung erfüllt ist oder nicht.

Grund

Aus welchem Grund gibt es diese Anforderung, was ist das Ziel, das mit ihrer Hilfe erreicht werden soll.

Konflikte

Mit welchen anderen Anforderungen ist diese unvereinbar? Diese Konflikte müssen vor der Realisierung aufgelöst werden.

Formalisierung

Hier findet sich die Formalisierung dieser Anforderung.

Kommentare

Hier können Kommentare oder Anmerkungen zu dieser Anforderung notiert werden.

Quellen	Abgeleitet	Übergeordnet	Anwendungsfälle
Aus welcher Quelle stammt die Anforderung (Personen, Normen, Abteilungen, etc.)?	Welche Anforderungen sind von dieser Anforderung abgeleitet?	Von welcher Anforderung ist diese Anforderung abgeleitet?	Welche Anwendungsfälle sind von dieser Anforderung betroffen?

Figure 4.1: The document structure used to capture a domain requirement.

After the structure was established, we initially captured 20 system level requirements and our questions and remarks for them. We also captured 8 environmental assumptions. Except for the environmental assumptions, all requirements were unclear to us. In this early phase, our confusion emerged from the lack of familiarity with the domain; we therefore had to meet several times with representatives of the company to clarify and discuss our understanding of the requirements. During those meetings it became clear, that nearly all of the requirements depend on assumptions about the environment or the system state that are not or not sufficiently enough stated by the original source. Although the larger part of those assumptions were obvious to the stakeholders of the company, the formal versions of the original requirements allowed us to easily give counterexamples where the requirements were violated. This lead to the insight, that additional information like *"the system has to be up and running"* or *"only if there is no system failure"* were readily available to all participating parties – except us: We viewed the requirements from the view of verification engineers, that is, we wanted to prove that *every* possible execution of the system satisfies the requirements. The stakeholders viewed the requirements from a testing and design view; they implicitly assumed the missing information. It took overall six meetings or 30 hours and additionally a number of phone calls to clarify the requirements up to a point where we had incorporated all environmental assumptions we could think of. In this process, we also identified 2 new system requirements and 1 new environmental assumption.

Through this case study we consider one requirement, which is derived from three of the captured system requirements:

R_1: The central unit may be only in the state battery low if the battery is low.

R_1 is one of the informal domain requirements for the central unit that we could extract from the system requirements. In total, only one of the system requirements was not relevant for the central unit. For 4 system requirements it remains unclear how they affect the individual components of the system.

From the remaining 17 system requirements and 9 environmental assumptions we could derive 29 domain requirements and 10 environmental assumptions that concern the central unit. Among the 29 domain requirements where 4, that had only relevance for the hardware of the central unit and 2 non-functional requirements. That leaves us with 23 domain requirements and 10 environmental assumptions.

4.3 From Informal Domain Requirements to Formal Domain Requirements

After we had captured the informal domain requirements and the environmental assumptions concerning the central unit, we tried to apply the pattern system described in Section 3.1.3 to it. All environmental assumptions could be formalized using the universality pattern (see Table 3.1). In the formalization of the informal domain requirements we used 6 instances of the response pattern, 4 instances of the universality pattern, 1 instance of the absence pattern (for those three patterns see Table 3.1), 10 instances of the bounded response pattern and 1 instance of the minimum duration pattern (for the remaining two patterns see Table 3.2).

We already noted in Chapter 2 and 3.1.3, that tools which match our criteria cannot verify real-time properties. Furthermore, for almost all of the other patterns we could not answer the question whether it is possible to translate them to a MLS for the VCC to our satisfaction. Therefore we did not consider the 17 formal domain requirements formalized with the patterns response, bounded response and minimum duration for this case study. The environmental assumptions as well as the 5 formal domain requirements that were formalized using the universality and the absence pattern were considered, although we show only one of them, namely the example requirement R_1 from the previous section.

In order to formalize this requirement, we assumed the role of a non-expert user and tried to look-up the pattern with the aids provided by the pattern systems. First we had to decide which type (see Figure 3.2 and 3.3) our requirement had: As R_1 does not contain any explicit time constraint, we concluded that the type could not be *Real-Time* and thus had to be *Qualitative*. With this type, we had two choices between the categories *Occurrence* and *Order*. As the requirement does not talk about the order of elements but rather about the simultaneous occurrence of them, we selected the *Occurrence* category. Then, the question was which of the four remaining patterns should be used and which scope should be selected. The patterns *Existence* and *Bounded Existence* talk about something which has to happen eventually and something which has to happen eventually under certain circumstances. Both are not appropriate for this domain requirement, as it states that something has to hold all the time. As the requirement formulates something positive, we are left with the *Universality* pattern. Finally, the scope of the pattern has to be *Globally*, as it has to be fulfilled under every circumstance. With this selection, we formalized the domain requirement as follows:

F_1: AG(central unit is in the state battery low $\rightarrow$ the battery is low)

Symbol	Preliminary interpretation
central unit is in the state battery low	*"central unit is in the state battery low"*
the battery is low	*"the battery is low"*

Table 4.1: Signature Sig_{F_1} of formula F_1

The signature of the formal domain requirement followed directly and is shown in Table 4.1.

4.4 Obtaining a Software Description

After the domain requirements were formalized, we turned our attention towards the source code of the F.BZ 100. As explained earlier, the requirements were mainly elicited for the successor system of cc100, but naturally the design and implementation phases for this system were not completed at that point. Because the new system was not complete, we decided to carry out the case study on already existing code, although we already knew that this system would not fulfill them. But as our goal was not to prove the software of F.BZ 100 right but to test our approach, this was not an issue.

Our next step was the creation of a software description. The first observation we made was, that all inputs and outputs to the software were already neatly organized in two dedicated header-files. This made the extraction of the software description very easy, as we just had to understand the different declarations in the two header-files. This lead to the second observation: The compiler used for the F.BZ 100 (IAR 78000 C-Compiler [78]) provides special keywords to bind variables to hardware ports, namely the keywords `sfr` and `sfrp` as well as some other language extensions. This should not be the only interesting observation about the source code and the compiler, but we continue the discussion of this aspect in Section 4.7.

In Table 4.2 we see the part of the software description that is necessary to transport the formal domain requirement to an MLS.

Names N_P	**Signature** Sig_P	**Addresses** A_P	**Classification** Cl
`ADCR`	`ADCR : sfrp`	`0xFF14`	IN
`bald_low_batt`	`bald_low_batt : bool`	–	AUX

Table 4.2: The partial software description for the F.BZ 100 source code.

While we were determining whether a variable is an input or an output, we

came across another interesting property of the source code: The underlying hardware of the F.BZ 100 (a NEC μPD78F9418AGC-8BT [79]) provides several shift-registers in the PISO (parallel in, serial out) and SIPO (serial in, parallel out) variant. In general, shift registers come in three flavors: The already mentioned PISO and SIPO as well as the SISO (serial-in, serial-out) variants. SISO shift-registers work like queues with fixed length: One can write one bit of data to the beginning of a SISO shift-register, and the register shifts the last bit out (so, its lost). For the software, this amounts to:

1. Write a bit to the shift-register.
2. Set the *data advance* bit of the shift-register so it knows it has been written to.
3. Repeat until all the data you wanted to write is inside the shift-register.

Luckily, the PISO and SIPO variants used in the F.BZ 100 are friendlier: They are used to access the radio transmit and receive functions: Whenever bits have to be sent via radio, one has to write to an output PISO shift register, whenever bits are received they can be read from an input shift register in SIPO mode. The hardware takes care of sending or receiving the serial data stream, while the software can read the content of the stream as whole block. Therefore this is no trouble for us, as we can write or read the complete data at once from the register. But if a SISO register had been used, we would have had to introduce specification code to observe that a certain value is written to the register, as the value of the register depends on past operations that are not observable anymore in the current state of the software. Our approach did not foresee such problems and so far we cannot think of a way to handle such non-observables except the manual annotation with specification code. We will come back to this issue in Section 6.1, but for now we continue with the creation of the relation between domain requirements and software.

4.5 Creating the I_{RS}

Our next step was the creation of the relation between domain requirements and software. We already knew that we had to rely on the knowledge of the software developers to find this relation. We therefore asked in one of the cooperation meetings the responsible developer if he could show us, which parts of the F.BZ 100 software are responsible for the domain phenomena in our domain requirements. Although the development of the software dates back around one year, he was instantly able to pin-point the necessary program fragments and gave us the part of the relation shown in Table 4.3.

Symbols from Sig_{F_1}	**Program fragments from F_P**
central unit is in the state battery low	`bald_low_batt == 1`
the battery is low	`ADCR < 158`

Table 4.3: The part of I_{RS} that is required for the running example.

4.6 Generating a Software Specification

As we have already seen in Section 3.3.1, the creation of the software specification is rather easy. We just substitute every domain phenomena in the formal domain requirements with the corresponding program fragment from the I_{RS}. Therefore, the software specification for our running example is:

$$SWS_1\text{:} \quad AG(\texttt{bald\_low\_batt} == 1 \rightarrow \texttt{ADCR} < 158)$$

4.7 Preparing the Code

After we created the software description, we could begin with the actual verification task. We already mentioned in Section 4.4 that the source-code for the F.BZ 100 contained special compiler-specific keywords and constructs and that it was therefore necessary to change the code so that the VCC could compile it. We also mentioned in Section 3.3.2 that in order to verify if the software is correct with respect to the domain requirements, we would have to make annotations to indicate, which parts of the program state would change during the execution of the single functions. The source code for the F.BZ 100 had originally 6153 lines of code, the main-file, in which the whole implementation was stored, had 4639 lines of code and 98 functions.

The preparation of the code had to be conducted in two steps: Our first goal was to compile the program with VCC, the second goal was to sufficiently annotate the program, such that VCC could report any violation of the memory-safety properties.

In order to complete our first goal, we had to remove all the compiler-specific code while preserving the program semantics. Fortunately, SeCa had already done some of this work, because the company itself ran the program in a simulation environment to allow unit-tests and facilitate the debugging process. Therefore we could simply add the line `#define UNIT_TEST` at the beginning of the main file and got rid of most of the compiler-specific extensions. Examples of those extensions are shown in Figure 4.2. Furthermore, we had to make the following changes:

```
#ifndef UNIT_TEST
interrupt [INTWTI_vect] void intervall_timer(void)
#else
void intervall_timer(void)
#endif
```

(a)

```
#ifndef UNIT_TEST
#pragma memory=dataseg(LCD_DATA)
#endif

#ifndef UNIT_TEST
#pragma memory=default
#endif
```

(b)

```
#ifndef UNIT_TEST
#include "in78000.h"
#include "io789418.h"
#else
#include "fcu100fake.h"
#endif
```

(c)

```
#define bit bool
```

(d)

Figure 4.2: Examples for compiler-specific keywords and constructs: (a) shows the usage of the `interrupt` keyword to bind a function to an interrupt vector and (b) some compiler-specific pragmas. In (c) one can see that for the simulation environment fake variables instead of hardware bindings had to be defined and (d) shows the `bool`-type that is used for efficiently packed bit-variables. All those examples were unknown to VCC.

- Since VCC uses a stricter type-system than the IAR C-Compiler, we had to introduce additional casts whenever two types did not match exactly, e.g. from `unsigned int` to `int`.
- VCC did not like postfix operators in array indices. Statements of the form `x=a[b++];` had to be replaced by two statements: `x=a[b];b++`.
- VCC also did not like a postifx decrement on a bitfield variable. Operations of the form `x--;` on an `unsigned char x:4;` were replaced with operations of the form `x=x-1;`.
- VCC overrides the reserved C keyword `register` to declare hybrid variables that exist in specification code as well as in production code. The F.BZ 100 source code used this keyword at two locations to tell the IAR C-Compiler that those variables should be assigned to registers and not to the heap. We had to remove the keyword, but this should not have an impact on the verification results.
- A compiler-bug in VCC [80] posed a great difficulty: The initialization of a global struct variable with non-`const` fields crashed VCC. As the F.BZ 100 source code uses those several times, we had to do a lot of rewriting to cope with this compiler bug. An instance of this problem and our solution is shown in Figure 4.3.

```
typedef struct eeprom_default_s {
    void          *A;
    void          *B;
    unsigned char  C;
    unsigned int   D;
} eeprom_default_t;

const eeprom_default_t eeprom_defaults[] =
{{ u, v, w, x},{...},...};
```

(a)

```
const void*          eeprom_defaults_A[]  = {u, ...};
const void*          eeprom_defaults_B[]  = {v, ...};
const unsigned char  eeprom_defaults_C[]  = {w, ...};
const unsigned int   eeprom_defaults_D[]  = {x, ...};
```

(b)

Figure 4.3: Due to a compiler bug in VCC we had to change several global struct variables: (a) shows an original instance of the declaration and initialization of a global struct array, (b) shows the modification we had to perform: We removed the struct array and replaced it with four arrays for each field in the original struct. Furthermore, every access to the original struct array had to be replaced with an access to one of the new array.

While we could perform those changes to the code rather fast (it took around two weeks to modify the whole code), the second goal posed a greater problem: We wanted to annotate the whole program such that VCC could prove that the code does not violate memory-safety properties (e.g. index out-of-bounds, null-pointer dereferencing, overflows). In order to do that, we had to annotate `writes` clauses and function pre- and post-conditions to every function.

The F.BZ 100 code contains 425 declared variables, of which 14 are arrays and 8 are non-primitives, e.g. structs or unions, and 98 functions. One problem here was the selection of the correct ownership macros and/or functions (the VCC website lists 18 ownership macros/functions and 6 additional ones for arrays [44]). Because VCC is build for concurrent programs, it needs to proof that during a function call no external process can modify global variables written in the function, i.e. that global variables are *thread-local*. Therefore we had not only to annotate which variables are written in a function (recursively) but also, which of them are thread local. because different types of variables need different ownership annotations, we had a lot of trial-and-error work at hand.

For primitives, the annotations were easy enough (`writes(&primitive_name)` often did the trick), but for arrays and arrays of structs this was not the case. VCC ran constantly into issues while trying to prove that structs or arrays where thread-local or even typed. For example, the F.BZ 100 source code uses unions

to facilitate access to bit-fields. The unions are not disjoint, but rather provide different views on the same data. VCC required the introduction of the `backing_member` keyword to internally "flatten" the union [81]. Without the keyword, VCC was not able to proof that the members of the union typed (see Figure 4.4 for an example). Similar, we had to give on multiple occasions loop invariants to allow VCC to recognize that certain arrays were indeed typed. A typical example of a completely annotated part of the software is shown in Figure 4.5.

```
union example {
  struct {
    unsigned char A:1;
    unsigned char B:1;
    unsigned char C:1;
    unsigned char D:1;
    unsigned char E:1;
    unsigned char F:1;
    unsigned char G:1;
    unsigned char H:1;
  } bits;
  struct {
    unsigned char X:4;
    unsigned char Y:4;
  } bits_sview;
  unsigned char byte;
};
```

(a) Original source code.

```
union example {
  struct {
    unsigned char A:1;
    unsigned char B:1;
    unsigned char C:1;
    unsigned char D:1;
    unsigned char E:1;
    unsigned char F:1;
    unsigned char G:1;
    unsigned char H:1;
  } bits;
  backing_member struct {
    unsigned char X:4;
    unsigned char Y:4;
  } bits_sview;
  backing_member unsigned char byte;
};
```

(b) Modification with the `backing_member` keyword.

Figure 4.4: The `backing_member` keyword tells VCC, that the union is not disjoint but rather provides different views on the same data.

```
void error_beep(void)
maintains(thread_local(&sio_rx_buf) &&
     mutable(&zw_buf_tln.split.alarmbereich) &&
     mutable(&zw_buf_tln.split.adresse1) &&
     mutable(&zw_buf_tln.split.adresse2) &&
     mutable(&zw_buf_tln.split.adresse3) &&
     mutable(&zw_buf_tln.split.ssb.byte) &&
     mutable(&zw_buf_tln.split.rssi))
writes(&timer_10ms, &RUN, &bit_tln, &buff_uebergabe,
     &zw_buf_tln_xtra, &RTS, &MK0, &MK1,
     &zw_buf_tln.split.alarmbereich, &relais.bits
     &zw_buf_tln.split.adresse1,
     &zw_buf_tln.split.adresse2,
     &zw_buf_tln.split.adresse3,
     &zw_buf_tln.split.ssb.byte,
     &zw_buf_tln.split.rssi)
{
     unsigned char i;

     for(i=1; i<=3; i++)
  invariant(
  thread_local(&sio_rx_buf) &&
  mutable(&zw_buf_tln.split.alarmbereich) &&
  mutable(&zw_buf_tln.split.adresse1) &&
  mutable(&zw_buf_tln.split.adresse2) &&
  mutable(&zw_buf_tln.split.adresse3) &&
  mutable(&zw_buf_tln.split.ssb.byte) &&
  mutable(&zw_buf_tln.split.rssi))
     {
          relais.bits.hupe = 1;
          waitfor(10);
          relais.bits.hupe = 0;
          waitfor(10);
     }
}
```

(a)

```
void waitfor(const unsigned char cTime)
maintains(thread_local(&sio_rx_buf) &&
     mutable(&zw_buf_tln.split.alarmbereich) &&
     mutable(&zw_buf_tln.split.adresse1) &&
     mutable(&zw_buf_tln.split.adresse2) &&
     mutable(&zw_buf_tln.split.adresse3) &&
     mutable(&zw_buf_tln.split.ssb.byte) &&
     mutable(&zw_buf_tln.split.rssi))
writes(&timer_10ms, &RUN, &bit_tln, &buff_uebergabe,
     &zw_buf_tln_xtra, &RTS, &MK0, &MK1,
     &zw_buf_tln.split.alarmbereich,
     &zw_buf_tln.split.adresse1,
     &zw_buf_tln.split.adresse2,
     &zw_buf_tln.split.adresse3,
     &zw_buf_tln.split.ssb.byte,
      &zw_buf_tln.split.rssi)
{
  timer_10ms=cTime;
  while(timer_10ms>0)
  invariant(thread_local(&sio_rx_buf) &&
     mutable(&zw_buf_tln.split.alarmbereich) &&
     mutable(&zw_buf_tln.split.adresse1) &&
     mutable(&zw_buf_tln.split.adresse2) &&
     mutable(&zw_buf_tln.split.adresse3) &&
     mutable(&zw_buf_tln.split.ssb.byte) &&
     mutable(&zw_buf_tln.split.rssi))
  {
     vCheckRcvBuf();
     RUN=1;
  }
}
```

(b)

```
void vCheckRcvBuf(void)
writes(&bit_tln, &buff_uebergabe, &zw_buf_tln_xtra, &RTS,
   &MK0, &MK1, &zw_buf_tln.split.alarmbereich,
   &zw_buf_tln.split.adresse1,
   &zw_buf_tln.split.adresse2,
   &zw_buf_tln.split.adresse3,
   &zw_buf_tln.split.ssb.byte,
   &zw_buf_tln.split.rssi)
maintains(thread_local(&sio_rx_buf) &&
    mutable(&zw_buf_tln.split.alarmbereich) &&
    mutable(&zw_buf_tln.split.adresse1) &&
    mutable(&zw_buf_tln.split.adresse2) &&
    mutable(&zw_buf_tln.split.adresse3) &&
    mutable(&zw_buf_tln.split.ssb.byte) &&
    mutable(&zw_buf_tln.split.rssi))
{
    if(buff_uebergabe==1 && bit_tln==0 )
    {
    zw_buf_tln.split.alarmbereich = 0;
    zw_buf_tln.split.adresse1 = sio_rx_buf[0];
    zw_buf_tln.split.adresse2 = sio_rx_buf[1];
    zw_buf_tln.split.adresse3 = sio_rx_buf[2];
    zw_buf_tln.split.ssb.byte = sio_rx_buf[3];
    zw_buf_tln.split.rssi = sio_rx_buf[4];
    zw_buf_tln_xtra = sio_rx_buf[5];
    bit_tln=1;
    buff_uebergabe=0;
    RTS = 1;
    MK0 &= ~(MK0_INTR_PROT);
    MK1 &= ~(MK1_INTR_PROT);
    }
}
```

(c)

Figure 4.5: A typical example for a fully annotated part of the program. The highlighted lines represent the original program without annotations. VCC took 30,22s to verify the memory-safety of the three functions on an Intel Core2Duo T9300 2,5Ghz with 4GB RAM.

The main problem with annotations was not that we had a rather complex trial-and-error process at hand, but that VCC took very long for certain functions. Especially the `main`-function of the program (2148 lines of code) took more than five hours to verify, only to notice, that some array was not thread-local.

Another great issue was, that VCC does not report unreachable code per default, and that it was very easy to produce unreachable code if one uses the wrong annotations. One prominent example of this problem is shown in Figure 4.6: Up to this point we expressed that an array is required to be thread-local in a function with the annotation `requires(mutable(as_array( <array_name>,<array_size>)))`. But in some larger functions, VCC completed the verification unexpectedly fast, so we inserted guaranteed errors to the function, namely `assert(false)` statements, to find out if VCC actually verified the whole function. This was indeed not the case. VCC decided that after the error assign, everything was unreachable and therefore the function had to be correct. We then constructed a minimal example of this behavior and reported it to the VCC community via their Codeplex website [44], because we thought this had to be an error in VCC. But after some discussions, it turned out that our annotation was wrong. The correct annotation in this case is `requires(is_mutable_array(<array_name>,<array_size>))`. We further found out, that the problem of unreachable code is not unknown to the VCC community and that there is the command-line flag `smoke` to iteratively insert an `assert(false)` statement at every program point in order to find such unreachable code. But of course, this flag leads to a considerable drop in performance: The previous example from Figure 4.5 took with the `smoke` flag 72,76s to verify, which is more than two times more for a very small program. Furthermore, even with the `smoke` flag VCC could only that there is some unreachable code, not at which program locations. In order to find such code, one has still to manually insert `assert(false)` statements in every branch of the control flow. For large methods with (a) confusing control flow and (b) long VCC runtimes, this proved to be a main obstacle.

In summary, we underestimated the amount of work necessary to prepare the F.BZ 100 source code such that we could try our approach for all domain requirements. Inside the available time-frame, we were not able to annotate the whole program, let alone successful verify it. Therefore we selected a domain requirement for which only a part of the program is relevant and verified this part of the program.

```
#include <vcc.h>

unsigned char X,A;
unsigned char thearray[6];

void fun(void)
writes(&X)
requires(mutable(as_array(thearray,6)))
requires(mutable(&A))
requires((A>=0 && A<=5))
{
  unsigned char c;
  c=thearray[A];
}
```

(a)

```
#include <vcc.h>

unsigned char X,A;
unsigned char thearray[6];

void fun(void)
writes(&X)
requires(mutable(as_array(thearray,6)))
requires(mutable(&A))
requires((A>=0 && A<=5))
{
  unsigned char c;
  c=thearray[A];
  assert(false);
}
```

(b)

```
#include <vcc.h>

unsigned char X,A;
unsigned char thearray[6];

void fun(void)
writes(&X)
//requires(mutable(as_array(thearray,6)))
requires(is_mutable_array(thearray, 6))
requires(mutable(&A))
requires((A>=0 && A<=5))
{
  unsigned char c;
  c=thearray[A];
  assert(false);
}
```

(c)

Figure 4.6: Unreachable code not reported is per default: The function in (a) verifies successful. One could think that everything is fine, but if we add an `assert(false)` statement after the array assignment (see (b)), it still verifies successful. Only if we change the annotation for the array as shown in (c), the verification fails as expected.

4.8 Generating an MLS from the Software Specification

After we had prepared the relevant part of the program such that VCC could show the memory-safety properties, we generated an MLS as described in Section 3.3.3 for our running example. Recall the software specification from Section 4.6:

$$SWS_1\text{:} \quad AG(\texttt{bald\_low\_batt} == 1 \rightarrow \texttt{ADCR} < 158)$$

As we mentioned in the previous section, we selected this domain requirement because we were able to completely annotate the part of the program which is responsible for it and because the part of the program is sufficiently small to show it in this work. In Section 3.3.3 we said that for the universality pattern it is sufficient to annotate every write to an output or auxiliary variable in the program fragments used in the software specification with an `assert(P)` statement, where `P` is the state formula used in the universality pattern. For our software specification, `bald_low_batt` is the only occurring auxiliary variable. Therefore, we had to find every write to `bald_low_batt` in the program and annotate it with `assert((bald_low_batt == 1) ==> (ADCR < 158))`. The variable `bald_low_batt` occurs only at 6 points in the program, one being the declaration, three being reads from it and two being writes to it. All writes take place in the function `test_batt`, where we annotated the `assert` statements.

The function `test_batt` calls only one other function, namely `ad_convert`. This function does not call any other functions, but rather accesses hardware ports to read the current battery voltage. The actual annotation was straight forward and is shown in Figure 4.7. VCC was run with the flags `time`, `stats` and `smoke` and took 25,91s to verify both functions successfully. The system running VCC was the same as in Figure 4.5.

The astute reader may have noticed that there is a slight discrepancy between the software description and the functions shown in Figure 4.7: The types for `ADCR` and `bald_low_batt` do not match. This is due to the unit test environment constructed by SeCa. Because the compiler used in this environment does not recognize the types `bool` and `sfrp`, they were changed to `int` and `unsigned char`, respectively. We added the `volatile` keyword to simulate non-deterministic values in `ADCR`, as VCC assumes that volatile variables may change during every atomic instruction of the program code [17].

```
#define _min_val_batt 158
volatile unsigned char ADCR;
int bald_low_batt;
// rest of the declarations omitted

void test_batt(void)
writes(&TESTBATT,&bald_low_batt,
  &fehler_notstrom,&ADCS,&ADIF,
  &ADS,&MK0,&MK1,&ADCR)
{
  unsigned int k;

  TESTBATT = 1;
  k = ad_convert(_ch_1);
  TESTBATT = 0;

  if(k < _min_val_batt)
  {
    bald_low_batt = 1;
    assert((bald_low_batt == 1)
      ==> (ADCR < 158);
    fehler_notstrom = 0;

    if(k < _leer_val_batt){
      fehler_notstrom = 1;
    }
  }
  else
  {
    fehler_notstrom = 0;
    bald_low_batt = 0;
  }
}
```

(a)

```
unsigned int ad_convert(unsigned char kanal)
writes(&ADIF,&ADS,&ADCS,&MK0,&MK1,&ADCR)
ensures(result==ADCR)
{
  unsigned char ucMK0_Bak, ucMK1_Bak;

  ADIF = 0;
  ADS = kanal;
  ADCS = 1;

  ucMK0_Bak = MK0;
  MK0 |= MK0_INTR_PROT;
  ucMK1_Bak = MK1;
  MK1 |= MK1_INTR_PROT;

  ADIF = 0;
  ADCS = 0;

  MK0 = ucMK0_Bak;
  MK1 = ucMK1_Bak;

  return ADCR;
}
```

(b)

Figure 4.7: The functions (a) `test_batt` and (b) `ad_convert` are responsible for measuring the battery. The highlighted lines are the annotations necessary to show, that the program is correct with respect to the domain requirements.

5 Related Work

The use of formal methods in software development processes is a constantly recurring theme of research in software engineering. We can distinguish several larger classes of concerns here: First, there are large methodologies like the B-Method [82], the Vienna Development Method (VDM) [83] or the RAISE method [84]. They all center around the idea that the rigorous application of formal methods in the software development process is beneficial, and they all use different flavors of formal languages to express common artifacts of this process, like requirements, design and specification. Furthermore, they provide extensive tool support for the different artifacts and development phases, be it to check requirements and specification for consistency or to check if a model of the system fulfills the requirements. They also rely heavily on refinement, that is, the development starts by formulating high level requirements in a formal language and continues with the stepwise refinement of those. Every iteration adds details to the formal representation and is then again checked for consistency. The process is repeated until the formal representation is detailed enough to allow a code generator to generate executable code.

Besides those large methodologies there exist some tools and frameworks that follow the same stepwise-refinement paradigm [85]: SCADE [86] generates C or ADA code from the high-level language LUSTRE [87], STATEMATE [88] does the same but for statechart specifications, and [89] generates C++ from RSML. Although very different in their implementation, all those tools and methods have a constructive, model-driven approach in common: They generate executable code from some high-level language and they all require the user to learn complex specification languages that are entirely different from common programming languages like C. In contrast to our work, they construct programs and therefore they do not provide support for analysing existing ones.

Notably, the RAISE method provides a new approach to software engineering as a whole, namely domain engineering [90–92]. In domain engineering, one tries to formalize the important parts of the domain to define the bridge between domain and software automatically. Like our approach, it is concerned with the relation between domain phenomena and software representations, but instead of

relying on the implicit domain knowledge of the developer, it demands an explicit formalization of this knowledge. This, again, requires much effort and training from the user, but promises great benefits through the automatic detection of errors in the transition from the domain to the software. A recent example for the application of domain engineering can be found in [93].

The connection between domain requirements and software specifications is apparently a one-way road: Although there exists a lot of literature on this topic in the field of program synthesis and model-driven development, the creation of specifications from requirements for already existing programs is somewhat neglected. As far as we know, there is currently only one approach concerned with this step, which is described by Seater et al. in [94]. Their approach describes an iterative process to obtain a specification in terms of the to-be-specified software from informal domain requirements represented as problem frames [95]. However, they currently lack some aspects which are — in our opinion — necessary to support the developer from the beginning: First, they still require a considerable amount of modelling experience, because they need to invent so-called breadcrumbs to bridge the gaps between domain requirements and the specification. A breadcrumb is an artificially inserted formulae that provides a connection between phenomena observable in the domain and phenomena observable by the software. While their function is the same as the function of our relation relation I_{RS}, they have to be invented and can be everything as long as it is expressible in Alloy. There is currently no help provided in obtaining those breadcrumbs, regardless if it would be very easy (e.g. just stating that an element from the domain is equal to an element of the software) or very complicated (e.g. introducing additional variables and formulating properties for them). Second, they do not show a source program or an input to a verification method and third, their formal domain requirements as well as their specifications are written in Alloy [96], which is a logical modelling language. It is neither clear nor easily deducible whether there exists an easy method to map an Alloy specification to a C program.

6 Discussion

6.1 Conclusions

We have shown that under the assumptions from Section 1.2 and for a small subset of the used requirement patterns, our approach is in principal feasible.

However, during our case-study we also discovered some remaining problems:

- The NEC C-Compiler used for the F.BZ 100 software provided extensions to the programming language that were unknown to VCC. As result, we had to modify the original code to be able to use VCC. Although we took great care in preserving the original semantics of the program, it remains questionable, whether the C semantics used by VCC are equivalent to the semantics used in the NEC C-Compiler, and therefore the integral validity of the verification results may be disputed. Furthermore, compilers may have subtle errors that can lead to failure of verified code. For example, modern, optimizing compilers can reorder instructions to improve the performance of the program. Although the compiler should preserve the semantics of the source language, the algorithms used for this reordering are very complex and their implementation may be erroneous, which in turn can lead to erroneous machine code. Similar to compilers, microprocessors can reorder instructions (out-of-order execution) to improve performance. In both cases, the results of the verification on the C code level may not be portable to the hardware level anymore. In other words: Although the correctness of the software has been proven, the system could behave unexpected and may even contain errors.

  This problem is not a new one, but the full extent of it just unfolds with the growing application of program verifiers in industrial settings. A notable contribution to this issue can be found in [97]:

  > You can't check code you can't parse. Checking code deeply requires understanding the code's semantics. The most basic requirement is that you parse it. Parsing is considered a solved

> problem. Unfortunately, this view is naive, rooted in the widely believed myth that programming languages exist.
>
> The C language does not exist; [...] While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler. Further, they equate compilation with certification. A file their compiler does not reject has been certified as "C code" no matter how blatantly illegal its contents may be to a language scholar. Fed this illegal not-C code, a tool's C front-end will reject it. This problem is the tool's problem.

As there will always be a large variety of different compilers, it may not be possible or necessary for researchers to develop tools for all of them. This does not lower the benefits of formal methods in general, but it shows that other quality-assuring methods like testing of the system are still necessary and important.

- We have seen that domain requirements for low-level software may deal with phenomena that are not directly observable in the software itself. Our discussion of shift-registers in Section 4.4 explained, that under certain circumstances the observable behavior of a system depends on a sequence of instructions by the software, which can not be traced back to a single program state, but only to a sequence of states. Furthermore, writes to a SISO shift-register may be performed in a loop, where in every iteration one bit is written and the data advance flag is set. As program verification tools have to abstract loops with invariants, the effects of those writes on the system behavior remain invisible to the tool. A solution may be the introduction of specification functions that record the complete shift-register during the execution of the loop, but this again requires expertise from the user. As we do not know which other hardware features may lead to unobservable behavior, we currently cannot give a generic solution to this problem.

- The annotation language used by VCC is very complex. Often it was not clear, which annotation had to be used to express changes of the program state. Even worse, wrong annotations easily led to unreachable code, which had to be detected manually by inserting `assert(false)` statements at every branch of the control flow of a method. In addition, the overhead for the memory-safety annotations of VCC is very high: For the part of the F.BZ 100 source-code shown in Figure 4.5 alone, the increase in size due to this annotations is by factor 2.42.

  Both observations, the error-proneness of the annotation process and the

great manual effort, constitute high obstacles for non-expert users. It may be conceivable to automatically infer the annotations necessary to express the changes to the program state by means of dataflow-analysis, but up to now this problem remains unsolved.

As we already noted in Section 4.7, we underestimated the amount of work necessary to prepare the whole F.BZ 100 source code. Unfortunately, we also could not evaluate the effectiveness of the proposed validation mechanism for the mapping between domain phenomena and program fragments: At the time of our case-study, the hardware developer responsible for the F.BZ 100 hardware had left the company, and we did not have the necessary knowledge of the hardware to conclusively perform the validation ourselves.

Nevertheless we were able to complete our approach for the example shown in Chapter 4. The core idea, the bridging between domain and software world, worked as expected. We gave a definition for the question of when is a software correct with respect to the requirements for the system in Chapter 2 and with this definition, we could show that our proposed bridging is useful as well as necessary for the creation of an MLS. In addition, we identified many challenges that have to be faced by developers in SMEs when concerned with program verification. The question of what kinds of requirements are verifiable on the software alone was answered in Chapter 2: While today the tool support for safety requirements is in principal sufficient, for the other classes it is not. The obstacles in form of required expertise are still too high to be tackled by non-expert users.

6.2 Future Work

In this work, the presented relations between domain phenomena and program fragments, as well as the software description and the formalized requirements were manually obtained and archived. This work was most of the time mechanical, but could be supported by tools:

- The formalization of informal domain requirements could take place in a tool that contains the various patterns and allows a user to choose between them. The user could then enter the state formulas in the appropriate pattern, which would allow the automatic creation of the signature of the formal domain requirements. Furthermore, changes of the requirements over time could be recorded with the help of, for example, existing version-control systems.
- The list of domain phenomena captured with the aforementioned tool could be read by another tool, which is integrated into a common IDE like Visual

Studio. Then the tool could provide the responsible developer with a list of domain phenomena, which could be related with program fragments marked by the user. This would allow the easy creation of the relation I_{RS}. The relation I_{RS} could then be tracked over the evolution of the program, and the developer could be prompted to decide, if changes to the program code or to the requirements break the relation or not. Furthermore, the relation of single variables to hardware ports could be inferred automatically by parsing the abstract syntax tree of the program and extracting the declared hardware addresses, thus automatically creating the relation I_{RH}. This would also allow the implicit construction of a software description. In addition, the extracted hardware addresses could be presented together with the state formulas in a side-by-side view to a hardware developer for validation. The hardware developer then would have to relate state formulas to hardware ports, and the tool could validate if the resulting relation I_{RH} corresponds to the relation I_{RS} between domain phenomena and program fragments.

Another important direction for future work is the automatic inference of annotations, that express which global variables a function reads or writes during execution. This would reduce the annotation overhead and the possibility for errors in the specification considerably.

As we already mentioned in the previous section, the nature of non-observable phenomena has to be examined more thoroughly, as we suspect that our shift-register example is not the only instance of this class of phenomena.

Last but not least the automatic annotation of the software specification through observer automatons has to be implemented, and its feasibility for the other patterns in the requirement pattern system has to be evaluated.

Bibliography

[1] Anthony Hall. Realising the benefits of formal methods. *J. UCS*, 13(5):669–678, 2007.

[2] Peter Amey. Correctness by construction: Better can also be cheaper. *CrossTalk: the Journal of Defense Software Engineering*, 2:24–28, 2002.

[3] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):1–36, 2009.

[4] Colin F. Snook and Rachel Harrison. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information & Software Technology*, 43(4):275–283, 2001.

[5] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 372–381. ACM, 2005.

[6] Gérard Berry. Synchronous design and verification of critical embedded systems using scade and esterel. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, page 2. Springer, 2007.

[7] Jonathan P. Bowen, Jonathan Bowen, Jonathan Bowen, Jonathan Bowen, Victoria Stavridou, Victoria Stavridou, Victoria Stavridou, and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8:189–209, 1993.

[8] Claude Hennebert and Gérard D. Guiho. Sacem: A fault tolerant system for train speed control. In *FTCS*, pages 624–628, 1993.

[9] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of b in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal*

Methods, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[10] Frédéric Badeau and Arnaud Amelot. Using b as a high level programming language in an industrial project: Roissy val. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2005.

[11] Klaas Wijbrans, Franc Buve, Robin Rijkers, and Wouter Geurts. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 419–424. Springer, 2008.

[12] Jan Tretmans, Klaas Wijbrans, and Michel R. V. Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[13] J. Barnes, R. Chapmann, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the Tokeneer enclave protection system. In *Proceedings of the 1st International Symposium on Secure Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 2006.

[14] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[15] Dan Craigen, Susan L. Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. In Jonathan P. Bowen and J. E. Nicholls, editors, *Z User Workshop*, Workshops in Computing, pages 1–5. Springer, 1992.

[16] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[17] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*,

volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[19] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[20] Coverity Static Analysis. http://www.coverity.com/.

[21] Steven D. Fraser, Frederick P. Brooks, Jr., Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas. "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering". In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 1026–1030, New York, NY, USA, 2007. ACM.

[22] Daniel Jackson. A Direct Path to Dependable Software. *Commun. ACM*, 52(4):78–88, 2009.

[23] Michael Jackson. The Real World. In Jim Davies, A. W. Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science: proceedings of the 1999 Oxford-Microsoft symposium in honour of Sir Tony Hoare*, pages 157–173. Prentice-Hall, 2000.

[24] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[25] Dines Bjørner. *Software Engineering 1: Abstraction and Modelling (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[26] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.

[27] Henrik C. Bohnenkamp, Holger Hermanns, David N. Jansen, Joost-Pieter Katoen, and Yaroslav S. Usenko. An industrial-strength formal method – a modest survey. In Tiziana Margaria, Bernhard Steffen, Anna Philippou, and Manfred Reitenspieß, editors, *ISoLA (Preliminary proceedings)*, volume TR-2004-6 of *Technical Report*, pages 284–295. Department of Computer Science, University of Cyprus, 2004.

[28] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Comparison between cpbpv, esc/java, cbmc, blast, eureka and why for bounded program verification. *CoRR*, abs/0808.1508, 2008.

[29] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[30] Farn Wang. Formal verification of timed systems: A survey and perspective. In *Proceedings of the IEEE*, page 2004, 2004.

[31] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electr. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[32] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[33] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[34] Alessandro Armando, Massimo Benerecetti, Dario Carotenuto, Jacopo Mantovani, and Pasquale Spica. The EUREKA Tool for Software Model Checking. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *ASE*, pages 541–542. ACM, 2007.

[35] Klocwork. http://www.klocwork.com.

[36] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[38] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276. ACM, 2003.

[39] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[40] PolySpace. http://www.mathworks.de/products/polyspace/.

[41] CodeSonar. http://www.grammatech.com/products/codesonar/overview.html.

[42] Astree. http://www.astree.ens.fr/.

[43] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[44] The Verifying C Compiler at Codeplex. http://vcc.codeplex.com/.

[45] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer, 2009.

[46] Verisoft XT: The Verisoft XT project. http://www.verisoftxt.de, 2007.

[47] Microsoft Visual Studio at MSDN. http://msdn.microsoft.com/en-us/vstudio/default.aspx.

[48] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[49] Greg Nelson. Extended static checking for java. In Dexter Kozen and Carron Shankland, editors, *MPC*, volume 3125 of *Lecture Notes in Computer Science*, page 1. Springer, 2004.

[50] Perry R. James and Patrice Chalin. Faster and more complete extended static checking for the java modeling language. *J. Autom. Reasoning*, 44(1-2):145–174, 2010.

[51] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview, 2004.

[52] Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 1 edition, October 2008.

[53] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[54] David Ferreira and Alberto Rodrigues da Silva. A requirements specification case study with ProjectIT-studio/requirements. In Roger L. Wainwright and Hisham Haddad, editors, *SAC*, pages 656–657. ACM, 2008.

[55] Carlos Videira, Joáo Leonardo Carmo, and Alberto Rodrigues da Silva. The projectit-rsl language overview. In Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and José Ambrosio Toval Álvarez, editors, *UML Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 269–272. Springer, 2004.

[56] Dipl.-Ing. Friedemann Bitsch. A way for applicable formal specification of safety requirements by tool-support. *FORMS 2003 - Symposium on Formal Methods for Railway Operation and Control Systems 2003*, 2003.

[57] Friedemann Bitsch. Safety patterns - the key to formal specification of safety requirements. In Udo Voges, editor, *SAFECOMP*, volume 2187 of *Lecture Notes in Computer Science*, pages 176–189. Springer, 2001.

[58] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE*, pages 411–420, 1999.

[59] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. Definition of CTL here.

[60] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1992. TCTL Definition.

[61] Salvatore La Torre and Margherita Napoli. A Decidable Dense Branching-Time Temporal Logic. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FSTTCS*, volume 1974 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2000.

[62] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar Nierstrasz, editor, *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.

[63] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[64] NEC Electronics Corporation. User's manual μpd789407a, 789417a subseries, April 2003.

[65] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, Heidelberg, 3. edition, 2005.

[66] Webb Stacy and Jean MacMillan. Cognitive bias in software engineering. *Commun. ACM*, 38(6):57–63, 1995.

[67] Ernie Cohen, Michal Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for c. *Electr. Notes Theor. Comput. Sci.*, 254:85–103, 2009.

[68] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *AMAST*, Workshops in Computing, pages 83–96. Springer, 1993.

[69] Henning Dierks and Marc Lettrari. Constructing test automata from graphical real-time requirements. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 433–454. Springer, 2002.

[70] Leszek Holenderski. Compositional verification of synchronous networks. In Mathai Joseph, editor, *FTRTFT*, volume 1926 of *Lecture Notes in Computer Science*, pages 214–227. Springer, 2000.

[71] Hartmut Wittke. *An Environment for Compositional Specification Verification of Complex Embedded Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, November 2005. Das Anhaengsel ist eine Disseration aus OL von jemandem, der sich viel damit beschaeftigt hat, Observer fuer - diese OSC-ES Pattern - TSAs als solche - Timing Diagrams zu bauen. Auf Seite 144 zitiert er zwei Dinge, und sagt "many others". Observer, Monitore, Testautomaten.

[72] Security Care GmbH. www.seca-online.de.

[73] BMWi. Zentrales Innovationsprogramm Mittelstand (ZIM). http://www.zim-bmwi.de/.

[74] Bundesministerium für Wirtschaft und Technologie (BMWi). http://www.bmwi.de/.

[75] DIN Deutsches Institut für Normung e.V - Normenausschuß Feuerwehrwesen (FNFW). Fire detection and fire alarm systems - Part 2: Control and indicating equipment; German version EN 54-2:1997, 1997.

[76] DIN Deutsches Institut für Normung e.V - Normenausschuß Feuerwehrwesen (FNFW). Fire detection and fire alarm systems - Part 25: Components using radio links and system requirements, German version EN 54-25:2005, 2005.

[77] Subversion. http://subversion.tigris.org/.

[78] IAR Systems. IAR 78000 C-Compiler V3.21A/386 included in IAR Embedded Workbench 2.20. http://www.iar.com/website1/1.0.1.0/50/1/, Last Visited 10.04.2010.

[79] NEC Electronics Corporation. μPD78F9418AGC-8BT, Form Factor QFP-80. http://www.necel.com/cgi-bin/nesdis/o003_e.cgi?article=UPD78F9418A.

[80] Stephan Tobies. Issure Tracker Entry: Heapified initialized structs crash the compiler. http://vcc.codeplex.com/WorkItem/View.aspx?WorkItemId=3896, Last Visited: 20.04.2010.

[81] Stephan Tobies. Union Flattening. http://vcc.codeplex.com/wikipage?title=Union%20flattening&referringTitle=Documentation, Last Viewed: 20.04.2010.

[82] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[83] C B Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[84] The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall International, 1995.

[85] Elaine Kant and David R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Trans. Software Eng.*, 7(5):458–471, 1981.

[86] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating Efficient Code From Data-Flow Programs. In *PLILP*, volume 22, pages 207–218, 1991. Special Issue on WOFACS'98.

[87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[88] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Eng.*, 16(4):403–414, 1990.

[89] Mats Per Erik Heimdahl and David J. Keenan. Generating code from hierarchical state-based requirements. In *RE*, pages 210–. IEEE Computer Society, 1997.

[90] Dines Bjørner. Domain engineering: a "radical innovation" for software and systems engineering? a biased account. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 100–144. Springer, 2003.

[91] Dines Bjørner. Domain engineering: A software engineering discipline in need of research. In Václav Hlavác, Keith G. Jeffery, and Jirí Wiedermann, editors, *SOFSEM*, volume 1963 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[92] Dines Bjørner. Domains as a prerequisite for requirements and software domain perspectives & facets, requirements aspects and software views. In Manfred Broy and Bernhard Rumpe, editors, *Requirements Targeting Software and Systems Engineering*, volume 1526 of *Lecture Notes in Computer Science*, pages 1–41. Springer, 1997.

[93] Mohammad Reza Nami, Mehdi Sagheb Tehrani, and Mohsen Sharifi. Applying domain engineering using raise into a particular banking domain. *SIGSOFT Softw. Eng. Notes*, 32(2):1–6, 2007.

[94] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement Progression in Problem Frames: Deriving Specifications from Requirements. *Requir. Eng.*, 12(2):77–102, 2007.

[95] Michael Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[96] Daniel Jackson. Alloy: A Logical Modelling Language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, page 1. Springer, 2003.

[97] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. CACM*, 53(2):66–75, 2010.