Software Qual J
DOI 10.1007/s11219-011-9138-7

System-specific static code analyses: a case study in the complex embedded systems domain

Holger M. Kienle · Johan Kraft · Thomas Nolte

© Springer Science+Business Media, LLC 2011

Abstract In this paper, we are exploring the approach to utilize system-specific static analyses of code with the goal to improve software quality for specific software systems. Specialized analyses, tailored for a particular system, make it possible to take advantage of system/domain knowledge that is not available to more generic analyses. Furthermore, analyses can be selected and/or developed in order to best meet the challenges and specific issues of the system at hand. As a result, such analyses can be used as a complement to more generic code analysis tools because they are likely to have a better impact on (business) concerns such as improving certain software quality attributes and reducing certain classes of failures. We present a case study of a large, industrial embedded system, giving examples of what kinds of analyses could be realized and demonstrate the feasibility of implementing such analyses. We synthesize lessons learned based on our case study and provide recommendations on how to realize system-specific analyses and how to get them adopted by industry.

Keywords Static analyses · Software quality · Bug checkers · Style checkers · Embedded systems

1 Introduction

In this paper, we relate our approach to complement and improve upon the currently established approaches of static checking of source code. Our approach is based upon identifying a set of dedicated static analyses that extract relevant information from the target system's source code with the goal to enable assessment of the system with respect

H. M. Kienle (✉) · J. Kraft · T. Nolte
Mälardalen University, Västerås, Sweden
e-mail: hkienle@acm.org

J. Kraft
e-mail: johan.kraft@mdh.se

T. Nolte
e-mail: thomas.nolte@mdh.se

Published online: 28 April 2011

to key concerns, primarily quality attributes. We call these kinds of analyses *system-specific analyses* as opposed to more generic ones. The key concerns of the target system need to be identified first and depend on stakeholders' concerns. The domain of our case study is embedded systems. In many respects, the quality attributes of these systems are especially stringent and demanding. Examples of important quality attributes in this domain include real-timing, reliability, safety, security, resource usage, and heterogeneity (Ebert and Salecker 2009).

To demonstrate our approach, we present an industrial case study of a complex embedded system implemented in C. We are focusing on C because it is most often used by industry to realize embedded systems. According to Ebert and Jones (2009), "more than 80 percent of all companies are using C and to some degree C++," referring to companies in the embedded systems domain. At the same time, C is known to "suffer from a number of potential fault modes" that often degrade software quality by introducing faults in the code (Hatton 2004).

Embedded systems developers in industry have to juggle several challenges: There is a steady growth of embedded systems that habitually were purely mechanical devices toward mechatronic systems. Also, as embedded systems are becoming ever more complex and feature-rich, there is a corresponding growth in software complexity (e.g., 1 GB of software in a premium car) (Ebert and Jones 2009). Developers of embedded software have to accommodate these growth trends without sacrificing quality. In contrast to enterprise and desktop software, the correctness of an embedded *real-time* system also depends on *timeliness* in the sense that the latency between input and output must not exceed specific limits (i.e., the deadline). Instrumenting the code for debugging may alter its temporal behavior and thereby cause a *probe effect* (McDowell and Helmbold 1989). Because of the criticality of embedded software, quality control has to be a concern during the entire life-cycle and needs to be supported by the software process. Developers of embedded software have to show that they produce software with the appropriate care in order to avoid liability (Åkerholm et al. 2009).

Our case study falls into the category of *complex embedded systems*, which are large industrial software systems, safety- and/or business-critical, and typically with a long maintenance history causing significant legacy issues (Andersson 2005). These systems often contain millions of lines of code and are developed and maintained by dozens or hundreds of engineers over many years. These systems are mechatronic, controlling functions in consumer cars, trucks, heavy industrial vehicles, process automation, power grids, industrial robots, etc. As a result, safety of complex embedded systems is a key quality attribute, because malfunction may harm people. Furthermore, reliability and availability are important because these systems are often also business-critical.

We examine the feasibility of our approach with a case study that involves the control system for an industrial robot developed at ABB Robotics (cf. Sect. 4.1). It is a large and complex software system that has evolved for almost 20 years to date and therefore has significant legacy issues. For instance, many of the original architects, designers, and developers are no longer available at the company, so a lot of system knowledge has been lost, and the amount of code has increased significantly over the years due to a steady feature growth.

1.1 Code analysis practices in the embedded systems industry

It is common practice in industry to leverage static analyses to improve the quality of embedded systems. In fact, static analysis "has become in fields with high liability risks a

mandatory part of verification" (Chelf and Ebert 2009). Chelf and Ebert state that in the embedded systems domain "static code analysis is applied to detect defects after the code successfully compiles and before the start of code inspections and unit testing" (Chelf and Ebert 2009). In our experience, however, it is used later in the process, as a last phase during unit testing. To check for bugs, developers and testers can rely on a number of general-purpose tools, ranging from compiler warnings to sophisticated static source code analysis tools such as CodeSonar (GrammaTech), Coverity Static Analysis (formerly Coverity Prevent), PC-lint (Gimpel), PolySpace (MathWorks), and QA·C (PRQA).[1]

Depending on their purpose, static analyses can be classified as either bug checkers or style checkers (Hovemeyer and Pugh 2004). The former locates code that violates a certain correctness property, while the latter looks for violations of coding guidelines. In principle, both kinds of analyses may be able to point out code that may lead to failures. In contrast to system-specific analyses, both bug checkers and style checkers are typically realized as generic approaches in the sense that they are applied to many software systems, perhaps across a wide spectrum of different kinds of systems.

The embedded systems industry often uses coding guidelines that promise to increase the quality of the source code by defining a *(safe) subset* (e.g., in terms of better maintainability and prevention of common mode failures) for a certain programming language (Hatton 2004). An example of a widely used coding standard is MISRA C (initially released in 1998 and updated in 2004), which has been developed by the Motor Industry Software Reliability Association (MISRA) for use of C in programming automotive electronics. MISRA C consists of over one hundred coding rules; most of these can be checked by static code analysis techniques (Hills 2005). It is now widely used across the whole spectrum of embedded software, including safety-critical systems.

Coding guidelines can be automatically enforced using customization features of static source code analysis tools. For MISRA C, there are dedicated checkers (e.g., QA·MISRA C); alternatively, there are generic tools with customization features that provide a pre-configuration for MISRA C rules (e.g., Coverity Extend and PC-lint). While these general approaches to static code checking can be valuable to improve upon quality, they are often difficult to apply to a large existing code base, and difficult to introduce into an existing development process (cf. Sect. 2) Specifically, they often report many rule violations of which a significant percentage are false positives, reported violations are difficult to verify, and the impact of individual (kinds of) violations on software quality is difficult to assess.

While static analyses have a number of potential drawbacks (cf. Sect. 2), they require relatively little manual intervention and deliver "objective" results. Thus, they can be used cost-effectively before more manual verification techniques such as code inspections (Chelf and Ebert 2009).

1.2 System-specific analyses: motivation and definition

In this paper, we are arguing for system-specific analyses that are tailored to a specific system because such analyses can take advantage of domain/system knowledge that is not available to more general analyses. System-specific analyses have the potential to

[1] When developing safety-critical code, Holzmann (2006) establishes the rule that code must be checked by the compiler with the most pedantic warnings enabled and by at least one static code analyzer, preferably several.

specifically target certain characteristics of the system that in turn have a strong impact on a set of quality attributes. Furthermore, they can be designed with the explicit goal to alleviate shortcomings of more generic analyses, resulting in, for instance, less false positives and easier verification of violations.

We believe that in order to achieve quality goals, the development process of embedded systems should leverage both generic and system-specific analyses. If generic analyses are already employed, they should be augmented with more dedicated analyses that take into account specifically the target system's peculiarities. Matsumura et al. (2002) investigated a legacy system and found that more than 32% of the failures were related to violations of system-specific coding rules that exist in the system—such rules would be good candidates for system-specific analyses.

We propose the following (working) definition for system-specific analyses:

> A system-specific analysis is specifically tailored to the system that it targets with the goal to expose a particular characteristic of the system, which has been identified by its stakeholders to be of major concern with respect to the system's quality.

Note that a system-specific analysis is only introduced if it is motivated by a concrete concern and this concern can be traced (ultimately) to a quality attribute. This is an important consideration because any analysis—regardless of where it sits in the analysis spectrum (cf. Sect. 2)—does incur costs. Since the analysis leverages unique characteristics of the target system, it will be typically the case that it needs to be custom-made in some way—only rarely will a generic analysis directly align. The effort of a custom-made analysis strongly depends on the chosen tool support and on the extent that the tool infrastrucuture already accommodates the needs of the system-specific analysis. Also note that even though we restrict our discussion in this paper to static techniques, the definition does not preclude dynamic analyses or hybrid approaches. Also, even though we only address analyses at the code level, one can imagine analyses that operate at the meta-model or architectural level.

Based on our experiences, we have identified the following (ideal) requirements for system-specific analyses: Each analysis should have a low false-positive rate, produce a manageable amount of issue reports that are easy to check or verify, produce timely results (e.g., seconds in interactive use; hours for batch processing), and justify the costs that it incurs (e.g., in terms of development and maintenance effort). Each analysis also needs a rationale that explains its existence (e.g., to improve upon a quality, to check a design constraint, or to point out a fault) and should be easy to specify and change with the help of suitable tool support.

The paper is organized as follows. Section 2 introduces the notion of an analysis spectrum, ranging from generic to system-specific analyses. We discuss tradeoffs of analyses situated in the spectrum and give analysis examples. Section 3 focuses on related work in static analyses and tools that are applicable for system-specific analyses. Now that we have established the background, we describe in Sect. 4 a case study that illustrates how system-specific analyses can be realized for a complex embedded system in industry. We first introduce the case study's system in Sect. 4.1 and then give concrete examples of suitable system-specific analyses in Sect. 4.2. We have started to implement these analyses with the help of Understand Analyst; this tool's capabilites and our experiences with it are described in Sect. 4.3. Section 5 provides our lessons learned based on the case study and introduces several hypotheses that generalize beyond the case study. Section 6 concludes the paper.

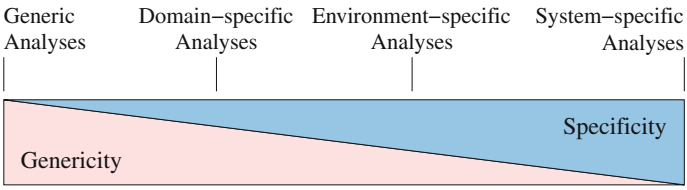

Fig. 1 Analysis spectrum (genericity vs. specificity)

2 Analysis spectrum

Code analyses can be classified with several dichotomies. For example, static versus dynamic, sound versus unsound, multi-threaded versus single-threaded, and multi-language versus single-language (Jackson and Rinard 2000; Kienle and Müller 2001). We introduce another dichotomy in this paper: generic versus system-specific.

While generic analyses can be applied to an open-ended number of systems, system-specific analyses are only applicable to one particular system. Upon closer inspection, one can identify a spectrum of analyses—rather than a dichotomy—that depend on the analyses' genericity or specificity as depicted in Fig. 1. With decreasing genericity (and increasing specificity), one can identify generic, domain-specific, environment-specific, and system-specific analyses in the spectrum.

As discussed before, **generic analyses** are situated at the one extreme in the spectrum. The most generic analyses or often language-agnostic analyses; for instance, text-based clone detection (Roy and Cordy 2008), textual differencing, and simple metrics such as lines-of-code counts. Static program slicing (Harman and Hierons 2001) is also generic because this technique is applicable and useful for any kind of system.[2] However, specific slicing tools are typically only applicable to a particular programming language. Then there are static code checkers (cf. Sect. 1) that are also language-specific but do not take the target system's domain or environment into account.

Domain-specific analyses are targeting particular kinds of systems that have certain common characteristics. Static analyses for real-time and concurrent systems are domain-specific, targeting properties of the program that are especially relevant for their respective domains. There are also analyses that target systems based on relational databases by extracting information from SQL queries (Nagy et al. 2010). Marinescu and Jurca (2006) introduce a model to describe the domain of enterprise applications and provide tool support to extract a model instance from an existing software system. Their tool supports Java enterprise applications that use MySQL or MS Access for persistency.

In this paper, we are concerned with the domain of embedded systems. Key characteristics of this domain are multi-tasking and constraints on execution-time and resource usage. Analyses for race conditions, WCET, and stack space meet the requirements of this domain. Another example of domain-specific analyses are the ones targeting Web sites (Kienle and Müller 2001). For the web domain, the unique need arises for analyses to detect cross-browser inconsistencies because a Web application must be executable across different browsers (Eaton and Memon 2007).

Environment-specific analyses are more specific than domain-specific ones because they are restricted to a certain combination of operating system, platform, libraries, frameworks, etc. An example of an environment-specific approach is a reverse engineering

[2] While slicing originated in imperative languages, there are also slicers for other programming paradigms.

analysis by Pinzger et al. (2003) that targets systems based on the COM/COM+ platform. The analysis leverages environment-specific knowledge such as framework calls and meta-data information stored in so-called type libraries. Similarly, Perin et al. (2010) present a reverse engineering analysis that targets Java 2 Enterprise Edition (J2EE), including identification of transactional methods (specified in XML deployment descriptors). Renggli et al. (2010) have developed static coding rules against the API of the Seaside web application framework, which is implemented in Smalltalk. Client code can be checked against 30 rules that target "common problems that repeatedly appear." These rules address issues such as possible bugs, bad style, and suboptimal code.

Finally, at the other end of the spectrum are **system-specific analyses**. Since system-specific analyses are only applicable to a single system, they are rarely described in the research literature, which is naturally interested in more generic approaches. However, there are system-specific analyses for the reverse- and re-engineering of legacy systems described in the literature. For example, Cecatto et al. have implemented tool support for the migration of a large industrial system based on proprietary technology (BAL and C-ISAM) to Java. System-specific analyses include goto elimination (Ceccato et al. 2008a, b), data model migration (Ceccato et al. 2008a, b), and detection of inconsistent locking (Ceccato and Tonella 2010).

Note that the specificity of a tool may differ from the specificity of the analysis technique itself. While Nagy et al. (2010) describe a domain-specific analysis that would be applicable to a broader range of systems containing SQL queries, their actual tool is tailored to an industrial system based on a proprietary imperative language that contains MS-SQL queries. One could also further refine the spectrum. For example, a more general kind of system-specific analysis would be one that targets a product line.

In practice, the introduced types of analyses blur into each other, meaning that a particular analysis or tool may cover a range in the spectrum. Generic code analysis tools offer customization features that make it possible to realize more specific analyses. Conversely, a system-specific analysis may turn out to be applicable to other, similar systems as well—however, in this case, the broadened applicability is accidental, not intentional. While individual system-specific analyses can have unintentionally broader applicability, the combination of suitable analyses for a particular system is unique.

2.1 Tradeoffs of generic and system-specific analyses

Quality assurance for (embedded) systems is often addressed with the help of both bug and style checkers, which are on the generic side of the analysis spectrum because they strive to be applicable to many software systems across a wide spectrum of different kinds of systems.

While these general approaches can be valuable to improve upon quality, in practice they suffer from the following, well-documented shortcomings:

Low effectiveness in terms of failure prevention: In the best case, a rule violation in the code corresponds to an actual failure.[3] However, a rule violation typically can only point out a fault in the code (e.g., an uninitialized variable), which may or may not lead to failure. In practice, only a fraction of rule violations prevent failures. As a result, different coding rules differ vastly in how successful they are in preventing failure.

[3] A *failure* is a diversion of the actual system behavior from the expected one at run-time. A *fault* is a point in the program that may lead to failure. Thus, fault is a static property of a system, while failure is a dynamic one (Hatton 2005).

Boogerd and Moonen (2009) have conducted a case study with an industrial embedded system, implemented in C, on the relation between MISRA 2004 violations and observed faults. One of their research questions is whether source code lines that trigger MISRA violations are more likely to contain faults. Of the 88 MISRA rules that they could measure, only 10 did perform consistently well in predicting faults. Also problematic, a previous study involving another embedded system in the same domain (Boogerd and Moonen 2008) found a different set of well-performing rules—only a single rule overlaps. Thus, these rules appear rather brittle in the sense that their effectiveness varies greatly across software projects.

According to Chelf and Ebert (2009), one of the root causes of failures and accidents involving embedded systems are uninitialized variables. Recognizing this notion, MISRA C 2004 has a rule (9.1) which requires local variables to be initialized before use. However, the studies by Boogerd and Moonen did not find a correlation between this rule and software failure.

Many reported rule violations: If code checkers are applied to existing industrial systems, they tend to report an overwhelming number of rule violations. Hatton (2004) reports on three sample systems that exhibited 1 rule violation according to MISRA C 1998 for every 2, 7, and 14 lines of code; thus, in this study depending on the system 7–50% of the code would have to be inspected and potentially changed. Similarly, Boogerd and Moonen (2009) report on a study where 30% of the lines of code in their system triggered non-conformance warnings with respect to MISRA C 2004 rules.

Thus, simply eliminating all rule violations is not practical. The pay-off in doing so is questionable anyways since the impact of rule violations on failure is highly uncertain (see above). Perhaps ironic, as discussed by Adams (1984), eliminating rule violations, has the real danger of introducing new defects, which may result in more faults.

Many false positives: Generic checkers that look for particular coding errors (e.g., buffer overruns or race conditions) are *unsound*[4] and as a result can report 30% or more false positives (Kremenek et al. 2004). High false-positive rates significantly decrease usability. In addition, once violations are resolved, the percentage of reported false positives tends to increase (Hovemeyer and Pugh 2004).

In their work on a Seaside-specific code checker, Renggli et al. (2010) report on a study where they manually evaluated the number of false positives of their checker versus a generic checker for Smalltalk in an industrial code base. The generic checker had a false-positive rate of 67% (940 out of 1,403 reported issues), whereas their API-specific checker had 24% (12 out of 51). They say that "while [generic] rules perform well on traditional Smalltalk code, there is an increasing number of false positives when applied to [environment]-specific code."

The developers of the Coverity tool made the experience that more than 30% false positives reported by a tool causes problems in the sense that users ignore the tool or have little trust in it (Bessey et al. 2010). GrammaTech's Anderson (2008) says that a rate of more than 50% "is usually considered unacceptable," but that risk-aware developers "are often prepared to accept a false-positive rate of as much as 75–90%."

Rule violations are difficult to verify: Since static analyses check for complex properties, such as race conditions, the reported violations can be inaccurate or difficult to understand (Hovemeyer and Pugh 2004). As a result, it may be difficult for the tool user to determine whether a violation is a true or false positive. Since many developers

[4] For bug checkers, soundness means the ability to detect all possible errors of a certain kind (Xie et al. 2005).

work under time pressure and thereby are eager to resolve violations quickly, there is the tendency to wrongly classify violations as false positives.

The Coverity developers note that explaining violations is often more difficult than finding them: "for many years we gave up on checkers that flagged concurrency errors; while finding such errors was not too difficult, explaining them to many users was" (Bessey et al. 2010).

All of the above drawbacks make it difficult to effectively employ generic static analyses in the development process. This problem exacerbates if generic static analyses are introduced to a large, existing system, like in our case study. For a large code base, static analyzers will generate a large number of violations [and the reported set of violations changes unpredictably with modifications of the code base and new tool versions (Bessey et al. 2010)].

Many tools recognize this problem and try to mitigate it by allowing the user to customize and filter the reported violations, but customizations are tedious to specify and maintain. For example, style checkers usually allow for suppressing warnings found to be irrelevant by annotating the corresponding code locations. Tool customizations are also needed to focus on the most critical violations in the context of the particular target software. Boogerd and Moonen (2009) caution that "adherence to a complete coding standard without customization may increase, rather than decrease, the probability of faults."

In order to be applicable to a large, industrial code base, a static analysis has to mitigate the problems discussed above. On the one hand, the analysis should be as unintrusive as possible, adding low overhead to the existing development process, and producing no spurious results; on the other hand, the analysis should show immediate pay-offs (e.g, improving the software's quality and reducing failures). System-specific analyses have the benefit that they can be developed to match the needs of a particular system. For example, an analysis could be tuned such that it stays below a certain false-positive rate—even if this may mean that many true-positives are missed. Similar considerations hold for the analysis' execution speed, clarity of violation reports, etc.

A new system-specific analysis is added as the opportunity or need for it arises. For example, repeated failures that can be traced to a common bug pattern (e.g., a wrong API call or failure to check the return value of a certain call) are good candidates for a system-specific analysis. Once the analysis is in place it may uncover other (dormant) bugs in the code and prevent the introduction of new bugs. Matsumura et al. (2002) make the point that many "rules" that can be checked by system-specific analyses are not explicitly designed for but rather an unintended consequence of the design. Thus, these rules do not show up in specification and design documents; instead, they emerge during system maintenance and evolution.

System-specific analyses can be straightforward specializations of more general analyses or bug patterns. For example, bug checkers can check for generic inconsistencies in the code: if the return value of a function call is always checked except in a few instances it may hint that a coding rule has been violated (Anderson 2008). However, reporting these kinds of inconsistencies as violations for all functions can produce many false positives and alienate the tool users. In contrast, a refined system-specific analysis that would report such violations only for certain function calls (e.g., that are known to be critical in some sense) can be much more effective.

System-specific analyses are often rather trivial compared with what fully-fledged code checkers can do, but they can be very effective, resulting in a superior "return on

investment." In fact, Coverity's developers call it a myth that "more analysis is always good" (Bessey et al. 2010): "it seems that adding more [sophisticated] analysis is always good. Bring on path sensitivity! Theorem proving! SAT solvers! Unfortunately, no."

A major concern for system-specific analyses is that it may require significant expertise and resources to develop them. Thus, there needs to be good tool support that makes it relatively straightforward to specify analyses. We identify implementation techniques for system-specific analyses in Sect. 3 and describe our approach for the case study in Sect. 4.3.

3 Related work

There is a vast body of related work in the areas of static code analyses, checkers, and tools. In the following, we highlight relevant work in these areas and discuss techniques that are applicable to system-specific analyses.

Typical examples of popular static code analyses are construction of call graphs (Murphy et al. 1998) and program slicing (Harman and Hierons 2001). There are also dedicated analyses for real-time systems [e.g., worst case execution-time analysis (Lv et al. 2009)] and concurrent/multithreaded systems [e.g., detection of race conditions and deadlocks (Engler and Ashcraft 2003)].

An important property for static analyses is *soundness*. According to Jackson and Rinard (2000), an analysis is sound if its results are guaranteed to hold for all program executions. In practice, sound analyses are often difficult or impractical to realize for and apply to complex code bases. For example, a sound pointer analysis with high precision does not scale to large code bases. Bug checkers are typically unsound (Bessey et al. 2010) and as a result produce both false positives and false negatives. GrammaTech's Anderson (2008) says that for an unsound analysis "the real measure of the effectiveness… is how well it simultaneously balances the false-positive rate with the false-negative rate." The same is true for analyses in the reverse engineering and program understanding domains, which are often unsound. The rationale for this is that even if the result of an analysis is unsound, it may give valuable information to a reverse engineer. Our system-specific analyses can be unsound, but since they are targeting a particular system the number of false positives should be small. In fact, one can argue that if a certain system-specific analysis cannot be realized with few false positives it has missed a key requirement and should not be deployed in the first place.

Perhaps the work closest in spirit to ours is an approach for the detection of faulty code by Matsumura et al. (2002). Their approach is based on the observation that legacy systems have implicit coding rules that "are quite specific to the particular software" and thus cannot be easily detected by generic checkers. Violations of such rules are detected with *faulty code patterns* that are specified with a pattern description language. They have evaluated their approach with a legacy system written in C and found that they were able to describe 76.9% of faulty code with their pattern language and that for these cases they could detect 86.8% of system failures. They "believe that [their] method is useful and practical in enhancing the reliability and reducing the maintenance cost" (Matsumura et al. 2002). Another close work is a bug checker for Java that looks for typical bug patterns (Hovemeyer and Pugh 2004). The authors present 18 language-specific analyses that strive to detect common bugs related to Java programming. While these patterns are more general than the system-specific analyses we are envisioning, the authors' goal are similar to ours. They are also interested to "produce output that is easy for programmers to

understand" and believe that analyses need not necessarily "perform deep analysis" in order to be effective. They also make the point that less generic analyses can be more easily tuned to "achieve an acceptably low rate of false warnings."

Implementations of static code analyses are often based on data flow analysis and abstract interpretation (Hankin 1998). There are many approaches to building system-specific analyses on top of existing tools. The key questions are: how to obtain a suitable fact base of the source code, and how to write analyses using the fact base? C/C++ fact bases can be obtained from compiler front-ends (e.g., EDG) and reverse engineering tools developed by academia [e.g., CPPX (Dean et al. 2001), gccXfront (Power and Malloy 2002), and XOgastan (Antoniol et al. 2003)] as well as industry (e.g., Bauhaus, Source-Audit, SolidFX, and Understand). Approaches exhibit different trade-offs in terms of robustness, scalability, completeness, and soundness. Our choice, Understand (cf. Sect. 4.3), offers a robust, scalable parser that produces an unsound fact base (e.g., because of missing pointer analysis).

There are approaches that simplify the writing of analyses with dedicated querying and manipulation languages. Similar to Understand, many tools offer an API that provides access to the fact base with a high-level programming or scripting language. For example, CodeSonar has a C API to check for properties in the code base (Anderson 2008). There are also approaches that allow declarative specification of fact base queries or of the analysis as a whole. For example, PAG is a program analyzer generator to realize flow-based, interprocedural analyses for compilers (Martin 1998). The TAWK tool extends the pattern syntax of `awk` so that abstract syntax trees can be matched (Atkinson and Griswold 2006). Patterns have associated actions that are coded in C with special syntax to denote pattern variables. Metal is a specification language to implement bug checkers that has the power to realize context-sensitive, interprocedural analyses (Hallem et al. 2002). While being easy to use is one of Metal's requirements, its power and generality comes at the price of a steep learning curve.

There are also approaches from the program understanding and reverse engineering communities (Kienle and Müller 2010). The GUPRO reverse engineering tool parses source into a graph structure, which is then queried via a graph query language called GReQL (Kullbach and Winter 1999). GReQL queries looks similar to SQL and have predicates based on first order logic. Semantic Grep is based on binary relations calculus and pattern matching (Bull et al. 2002). It allows queries such as "show all functions in X.c" or, more advanced, "show all function calls from X.c to Y.c." The tool is based on the established tools `grok` and `grep`, transforming its queries into commands for these tools. The IntensiVE tool allows to verify code patterns such as coding convention or architectural guidelines with the goal to prevent deterioration of the code base during system evolution (Brichau et al. 2010). Code patterns are specified with a logic-based query language in which queries are expressed as logic conditions over the program's parts. Revealer is a tool for architectural recovery, based on syntactic analysis (Pinzger et al. 2002). It allows searching for complex patterns in source code—corresponding to "hotspots"—of a specific architectural view. For instance, the tool can be instructed to extract the relevant program statements of socket communication. The patterns are expressed in XML notation.

4 Case study: applying system-specific analyses to a complex embedded system

In this section, we describe our experiences in realizing system-specific analyses for a complex embedded system in industry. We introduce the system (cf. Sect. 4.1), describe

our proposed analyses (cf. Sect. 4.2), and brief discuss our implementation approach (cf. Sect. 4.3).

4.1 The industrial robotics system

The system under study is a control system for industrial robots from ABB Robotics. The software system architecture was conceived in the early 1990s and has evolved considerably since then. Thousands of changes have been performed over this time, by hundreds of software developers, of which many are no longer at the company. Some parts of the system have hard real-time constraints, while others have strong focus on performance, as the robot's maximum speed in a production process is mainly limited by the computational performance of the controller.

In essence, the robot controller has an object-oriented design, but is mainly implemented in C. It consists of approximately 2500 KLOC distributed in 400–500 so-called classes, which are organized in a set of subsystems. The robot controller consists of three computers: the axis computer, a DSP which controls the motors of the robot, the input/output computer, and the main computer, which is the most complex part. The axis computer is truly safety-critical, but not very complex, while the main computer is more focused on performance; a deadline violation results in a failure (stop of the robot), but the axis computer keeps the system safe by stopping the motors and applying the brakes if control data is not received in time. Thus, the axis computer is truly safety-critical, while the main computer is mostly business-critical.

The main computer is an industry-PC solution using high-end Intel processors and the VxWorks[5] real-time operating system from Wind River, Inc. The software system in the main computer consists of more than 60 tasks, which are scheduled using preemptive fixed priority scheduling.[6] The tasks communicate through message queues and shared data areas. Many tasks consist of several services, sometimes over 200, which are activated by messages from other tasks, or from a timer. A task typically spends most of the time blocked, waiting for incoming messages. When a message arrives, or a timer expires, the task executes the service corresponding to the event which occurred. During the execution of a service, any incoming messages are buffered in the message queue for later processing.

At this stage of our work, we are examining only a part of the whole system, the motion control subsystem, which is responsible for generating the motor references and brake signals required by the axis computer. To give an indication of the size of the subsystem, Table 1 provides several code metrics.

The axis computer sends requests to the main computer with a fixed, high rate, over 200 Hz. The axis computer expects a reply in the form of motor references within a certain time. This mainly depends on three large tasks in the motion control subsystem, as depicted in Fig. 2. There are also other tasks in this subsystem, but these three are the largest, by far, as each of these consists of 1,500–2,500 functions and 100–200 KLOC each, even though some code is shared. We refer to these large tasks as *A*, *B,* and *C*. The tasks *B* and *C* have high priority and run frequently with a fixed period. Task *A* executes mostly in the beginning of each robot movement and has lower priority, but produces data required by *B*. The *B* task processes the data and forwards the result in smaller parts to *C*, which makes

[5] http://www.windriver.com/products/vxworks/.

[6] Embedded real-time systems are often designed and implemented as a set of tasks where a task is "the basic unit of work from the standpoint of a control program" (RTCA 1992). It may be realized as an operating system process or thread.

Table 1 Metrics of available subsystem code

Metric	Value
Total lines of code	1,082,934
Executable statements	448,853
Declarative statements	371,605
Functions	9,604
Source files	1,699

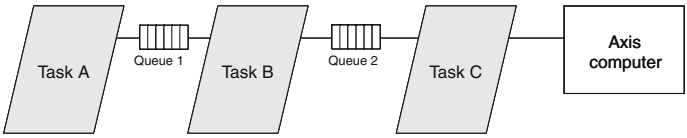

Fig. 2 The main tasks of the motion control subsystem

the final processing and sends motor references to the axis computer. The data is sent through message queues. The interface of the motion control subsystem in *A* receives orders to move the robot and other commands from a client application (not shown in Fig. 2), which decides how to move the robot.

4.2 Examples of system-specific analyses

In this section, we motivate the benefits of introducing system-specific analyses for the case study that can be applied to ABB's system (and perhaps to other similar systems). We give concrete examples of a range of possible analyses and explain why they can have a positive impact on the system's quality. Based on previous work with this industrial partner, we believe that these analyses have the potential to improve upon the current development practice, resulting in an increase in the system's maintainability, robustness, and safety.

In the follow sections, we describe seven system-specific analyses. These analyses are meant to show the range of possible analyses. Of course, there is an open-ended number of potential analyses for each system.

4.2.1 Task include structure

A typical technique for a system-specific analysis is to take advantage of naming conventions and architectural constraints that exist in the system. Such rules and constraints are often not explicitly documented, but in the heads of (senior) developers. As a result, they are easily violated during maintenance activities, specially by less experienced developers.

A typical example of a rule in the ABB system is the naming conventions for file names when implementing tasks. The implementation of a task *X* is split into four files: *X*.h, *X*.c, *X*ts.h, and *X*ts.c. This rule can be easily checked and also enables us to check the inter-file constraints described next.

Each task offers a set of public interface methods, corresponding to the available task services, and the task proper, which constitutes the task's internal implementation. The public interface methods are executed by other tasks (i.e., the clients) and act as proxies, sending service requests to the task proper via a message queue by calling `ipc_send()`.

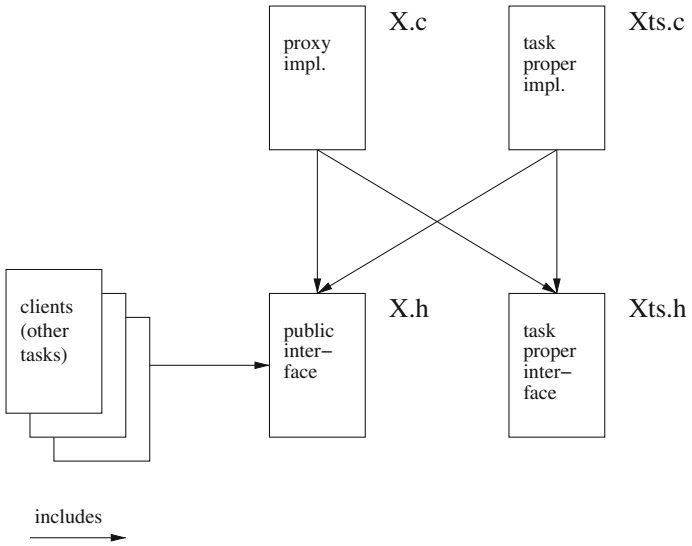

Fig. 3 Files and include dependencies for a task

The task proper handles these requests in its IPC dispatch loop with `ipc_receive()`. The public interface methods are defined (as C functions) in *X*.h and implemented in *X*.c. The task proper consists of the task's IPC loop and is implemented in the files *X*ts.c and *X*ts.h. The IPC loop of a task can be identified because it uses a standard coding pattern (cf. Sect. 4.2.3).

The dependencies between the files is depicted in Fig. 3. Importantly, the following constraints must not be violated:

– *X*.h should not include *X*ts.h and vice versa.
– Only *X*ts.c and *X*.c should include *X*ts.h, no other files.

These dependencies and constraints can be seen as a conceptual architecture (where files represent "components" and include relationship constitute the dependencies between components). A system-specific analysis can check that these architectural constraints are not violated. Such a check is highly desirable because the include constraints realize a decoupling of the task's interface from the internal implementation, hiding the details of the messaging behavior. Violations of this architecture will probably lead to declining quality of the system. For example, if a client was using an internal task's service directly by including and calling a function in *X*ts, the message queuing mechanism could be circumvented, resulting in unpredictable behavior.

4.2.2 Task IPC data structures

This system-specific analysis illustrates another typical technique, taking advantage of naming conventions coupled with structural constraints on data types.

An important constraint in the implementation of a task is that its public services (the interface methods) have to match the defined IPC message data-types and message codes. Public services are declared in *X*.h of the form *X*_*s*() where *s* is the name of a particular service (see left side of Fig. 4). For each service, there needs to be

X.h

```
/* declaration of services */
...
int X_s1(...);
int X_s2(...);
...
```

*X*ts.h

```
/* service identifiers     */
enum X_ipc_cmd
{
  X_cmd_s1,
  X_cmd_s2,
  ...
}

/* service messages */
typedef struct {
  ...
} X_MSG_s1;
typedef struct {
  ...
} X_MSG_s2;
...

/* union of all service messages */
typedef union {
  X_MSG_s1 s1;
  X_MSG_s2 s2;
  ...
} X_MSG;
```

Fig. 4 Data structures for a task's IPC (*X*.h and *X*ts.h)

- a corresponding enum member *X*_cmd_*s* in *X*_ipc_cmd that identifies the IPC message (i.e., service identifier).
- a struct typedef *X*_MSG_*s* that defines the service's message format.
- a union member *s* of type *X*_MSG_*s* in a union *X*_MSG, which contains all of the task's service messages.

The above structures are defined in *X*ts.h as illustrated on the right side of Fig. 4.

This coding pattern enables service *s* to send an IPC message (identified with *X*_cmd_*s* and having the message format *X*_MSG_*s*) to the task proper. A system-specific analysis can assure that for each service there is a unique, corresponding service identifier and message structure. This check is especially relevant in the following scenarios. When a new message is introduced, existing message structures should not be reused (even though the message may have the same fields). This is a form of defensive programming: if the message structure for one service is changed, this change will not affect other services if the message structure is not shared. More importantly, when a service is removed then all corresponding code should be eliminated as well in order to avoid dead code and deterioration of the code base. Having such a check is of practical importance for maintenance because the number of services can be large. For example, task *C* has more than 100 services.

4.2.3 Task IPC dispatch loop

While the previous system-specific analysis is focused on constraints on data structure, the following analysis is an example that checks constraints on code that manipulate data structures.

Figure 5 gives a stylized example of a task's IPC dispatch loop. When a message is received properly, it is passed to a message dispatcher function, here ipc_disp(), which

Fig. 5 Code structure of a task's IPC dispatch loop (*X*ts.c)

```
/* IPC loop */
void Xts()
{
  int status, timeout;
  short cmd;
  X_MSG msg;

  while (1)
  {
    status = ipc_receive (&msg, &cmd, timeout);
    if (status == TIMEOUT)
    {
      ...
    } else {
      ipc_disp (cmd, &msg);
   }
}

/* message dispatcher */
void ipc_disp (enum X_ipc_cmd cmd, X_MSG *msg )
{
  switch (cmd)
  {
    case X_cmd_s1:
      X_MSG_s1 *m= &msg->s1;
      ...
      break;
    case X_cmd_s2:
      ...
  }
}
```

contains a (often large) `switch` statement, where each `case` handles a service request. The case labels refer to the service identifiers, *X*`_cmd_`*s*.

An interesting system-specific analysis for this coding pattern is to compare the labels of the switch statement with the declared service identifiers (in the *X*_ipc_cmd `enum` type) to detect inconsistencies. Also, for each service function *X*_*s*() there needs to be a corresponding case label *X*`_cmd_`*s*. If this is not so it hints at obsolete or else unused code that should be removed from the system. A similar analysis could also find service identifiers that are present in the task message dispatcher, but which have no corresponding service function, maybe due to an incomplete earlier removal of the service. The switch case for this service can thereby be safely removed in order to make the message dispatcher's code shorter and more maintainable.

4.2.4 Task scheduling priority

The following example illustrates that there are often vital constraints for a system that are trivial to check for. This represents an ideal case for a system-specific analysis, because while the analysis is cheap to implement (and the meaning of a reported violation is easy to grasp), it has a high impact on the system's quality.

For embedded systems using fixed-priority scheduling, it is desirable that task priorities are static (i.e., they do not change during run-time) because this makes the system's response time more predictable. However, dynamic priority changes are supported by most real-time operating systems.

```
bytes = ipc_receive ( ... IPC_NODELAY ... );
...
if (bytes <= 0 )
{
  (void) os_change_priority( task_id, max_priority );
  prio_change = TRUE;
}
```

Fig. 6 Example of priority change (*X*ts.c)

The ABB system mostly uses static priorities so that the system is easier to maintain and predict, but there are a few cases where priorities are changed at run-time in order to optimize the temporal behavior (e.g., to prevent starvation due to empty message queues). For example, task *B* boosts *A*'s priority if a critical message queue drops below a certain length. Figure 6 shows an example of a priority change in response to an empty message queue. In order to change priorities, tasks can call a wrapper function, `os_change_priority()`, that abstracts away from the actual operating system call for VxWorks and Windows.[7] In the wrapper, priorities are changed with the system calls `taskPrioritySet()` for VxWorks and `SetThreadPriority()` for Windows.

A simple system-specific analysis that analyses the calls to priority-changing functions (i.e., the wrapper and the two system calls) can assure that code changes do not introduce new dynamic priority changes. Such a check is important because such changes may alter the system's timing behavior drastically. Thus, these changes should always be done in a controlled manner, e.g., in coordination with an in-depth impact analysis involving senior developers, followed by a thorough dynamic analysis of the new system behavior. If this process is not followed then timing problems can show up late when the system has been already fielded. This rather simple coding constraints turns out to be also a design constraint and this design constraint is in turn the result of higher-level requirements for quality assurance.

4.2.5 IPC message in/out parameters

The following example shows that stylized annotations such as comments in code can be used to realize system-specific analyses.

As mentioned before, each service that a task offers is associated with an IPC message format defined in a `struct` (cf. Sect. 4.2.2). Figure 7 shows an example of a IPC message; it has a common header followed by three message-specific fields.

The implementation of the public interface for the service sets the relevant fields of the message, which is then sent to the task proper. If the sender expects a reply from the task proper a special IPC call is used that waits for a reply, `ipc_sendwait()`. For such cases, some of the message fields are used as return values. Thus, from the perspective of the sender, fields have "in" or "out" parameter passing semantics. The importance of the in/out semantics is reflected in the code by providing comments next to most message fields (see Fig. 7). These comments consistently read "In" or "Out" in C commenting style.

Naturally, the proxy implementation and the task proper have to agree on the fields' semantics. Continuing the example, Fig. 8 shows typical code for the message sender and

[7] The system runs on VxWorks in production but a Windows build is also used, during development and early testing phases.

Fig. 7 Example of IPC message with in/out parameters (*X*ts.h)

```
typedef struct
{
  IPC_HEADER header;
  int field1;                /* In  */
  BOOL field2;               /* Out */
  STATUS field3;             /* Out */
} X_MSG_s42;
```

X.c

```
STATUS X_s42 ( int mg )
{
  /* set In parameters */
  X_MSG_s42 msg;
  msg.field1 = mg;

  /* send message */
  ipc_sendwait( ... &msg ... );

  /* use Out parameters */
  if( msg.field2 ) {
    ...
  }

  return msg.field3;
}
```

*X*ts.c

```
...
X_MSG_s42 *msg;
ipc_receive( ... &msg ... );

msg->field3 =
  do_computation (
    ...
    msg->field1 ...,
    &msg->field2 );

ipc_answer( ... &msg ... );
...
```

Fig. 8 Example how an IPC message is assembled and sent (*left*), and processed by the receiver (*right*)

receiver. The sender first assembles the message by setting the "in" fields (here `field1`). Once the sender receives the reply it can access the "out" fields (here `field2` and `field3`). Analogously, the receiver of the message reads out the "in" fields and sets the "out" fields.

A system-specific analysis can check that in-parameters are indeed set by the proxy and that out parameters are indeed set by the task proper. Also, inconsistencies between the commenting and the actual implementation can be pointed out.

Most parameters are manipulated within the same functions that perform the IPC calls. There are only a few cases where a parameter is passed by reference. Such a case is illustrated in Fig. 8, where the receiver passes the address of `field2` in the `do_computation()` call. Since these cases are rare, the system-specific analysis does not need to necessarily perform an inter-procedural analysis. Instead, such cases could be approximated in an unsound manner. For example, the analysis could assume that `do_computation()` will set `field2` because it receives its address. While the resulting analysis would be slightly less precise, it would be easier to realize. This scenario nicely illustrates that certain properties of a system can reduce the complexity of a system-specific analysis because simplifying assumptions can be made without invalidating the analysis' utility.

4.2.6 Visibility constraints

As this example shows, system-specific analyses can also involve the C preprocessor.

The ABB system uses visibility constraints to describe the intended clients of functions. Functions are declared and defined by prepending `PUBLIC, INTERNAL`, and `PRIVATE` macros, where `PRIVATE` is translated to `static`, which in C/C++ means "file internal."

```
   ...
   if (sem_wait(SEM_BUF, 10000) != OK)
   {
      ERROR("Timeout while waiting for SEM_BUF");
      return -1;
   }

   /* semaphore obtained OK */
|  if (p->index < p->max)
|  {
|    p->index++;
|  } else {
|    p->index = 0;
|  }
   sem_give(SEM_BUF);
   ...
```

Fig. 9 Example of semaphore with timeout, protecting a critical section (marked by "|")

`PUBLIC` means that all code is allowed to call the function. In contrast, `INTERNAL` means that the function should be used only within the subsystem where it is defined.[8]

In the ABB system, all functions except `PRIVATE` are however globally visible, as there is only a single global namespace for non-private functions. Hence, the `PUBLIC` and `INTERNAL` labels are design constraints that are not actually enforced by the C compiler.

A system-specific analysis can check whether the restrictions of the `INTERNAL` declarations are actually adhered to by the implementation and can thereby identify unintended dependencies due to calls of `INTERNAL` functions from other modules.

4.2.7 Task and semaphore dependencies

The following analyses target potential failures related to multi-threaded programming, which is both error-prone and difficult to debug.

Tasks in the ABB system share data, typically state variables, through passing of pointers to global data structures. These are often quite large and divided into several nested `struct`/`union` definitions, which makes them difficult to comprehend and maintain. A system-specific analysis could identify what tasks are using what parts of these global data structures, and thereby to infer the data that is shared between multiple tasks.

When tasks share data in this manner, they should use semaphores in order to protect critical sections, i.e., where a task must have exclusive access to a particular variable in order to achieve thread-safety. Figure 9 gives an example how a critical section is protected by a semaphore so that the data structure accessed via `p` can be safely manipulated. Without the semaphore protection, a context switch could occur between the condition and the index increment, which could cause an erroneous system state where the index has been increased beyond its limit, i.e., a potential buffer overflow error. A system-specific analysis could check for functions which modify shared data without first acquiring the correct semaphore for a critical section. Such an analysis helps to identify transient faults, which otherwise are very hard to replicate and understand, as they depend on random execution-time variations which alter the relative timing between tasks.

For response-time analysis, since a semaphore may block the execution of a task, it is necessary to know which tasks are using which semaphores. For large systems, this

[8] The system is organized into logical units called subsystems, where each subsystem consists of a set of C files.

analysis would be very time consuming and error prone if performed manually. As an example, more than 60 tasks in the ABB main computer subsystem use altogether close to 200 semaphores. Therefore, another beneficial system-specific analysis would be to identify what semaphores are used by each task in order to facilitate response-time analysis. This can be further extended by also identifying the sections of code protected by semaphores (i.e., the program locations in which the semaphore's lock function is called, and the corresponding locations where the semaphore is released). This could allow for focused WCET-analysis[9] in order to estimate the worst case blocking time of the task, which is necessary input for response-time analysis. Apart from response-time analysis, this analysis could also be used to check for missing semaphore release operations etc.

Another interesting analysis, related to semaphores but actually more general, is to check whether the return values of specific functions are actually handled by the caller. The semaphore lock operation is a typical example of this issue, as it is often performed with a time-out. If the time-out occurs, the wait is aborted and an error code returned to indicate that the semaphore has not been obtained. In the ABB system, this is treated as an error, resulting in a controlled stop of the robot in order to meet safety requirements. An example is provided in Fig. 9.

However, if this check is forgotten when adding or modifying a critical section, a potential concurrency error is introduced which may lead to an erroneous, inconsistent system state, which in the worst case is detrimental for the system's safety.

All complex embedded systems use multi-threading in some form. The drawbacks of generic analyses—many reported false positives and violations that are difficult to verify—are especially pronounced for these kinds of analyses. Conversely, system-specific analyses have the potential to improve upon generic analyses above average in this respect.

4.3 Implementing the analyses

We have started to implement system-specific analyses by leveraging SciTools' Understand Analyst.[10] Understand is a GUI-based tool that can be used as an IDE for software development, but its primary strength are views for program understanding. Understand can process several different languages, but we are using it for C only.

4.3.1 Understand's graph model and token stream

Understand builds a database model of the source code, describing what symbols (i.e., variables, functions, data types, etc.) exist and where they are used. Understand resolves all symbols except calls via function pointers. Function pointer calls are detected, as well as their creation, but Understand does not have any support for the data-flow analysis required to connect these two. Preprocessor directives in the code are expanded with respect to specified preprocessor definitions and properly taken into account. Code that would be excluded by the preprocessing are included in the model, but marked as inactive.[11]

[9] Industrial tools for WCET analysis exists, for instance AiT (AbsInt) and RapiTime (Rapita Systems). However, these tools are challenging to apply to a large system and often require manual code annotations or run-time measurements.

[10] http://www.scitools.com.

[11] This can be useful because analyses can be written so that they do not have to be run repeatedly for different configurations.

example.c

```
typedef float* fltp;

int g;

void foo() {
  bar( 42 );
}

void bar( int i ) {
  fltp fp;
  g = i;
}
```

graph model (excerpt)

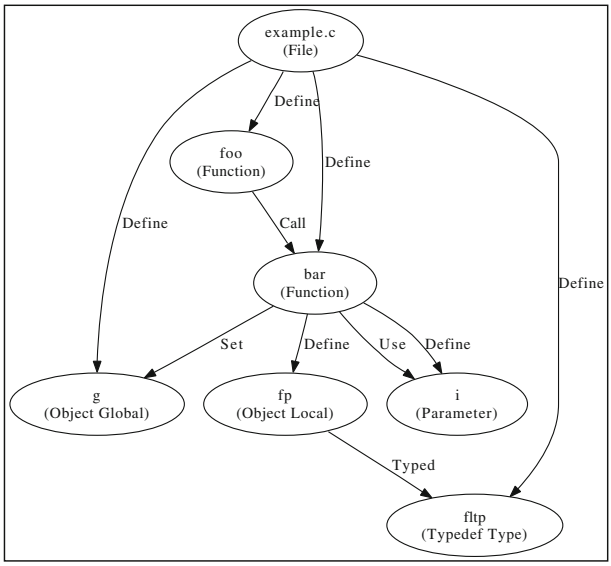

Fig. 10 Example of C source code and corresponding Understand graph model

Understand exposes its underlying database of the code structure with an API in C or Perl (SciTools 2004); we are using Perl. The database represents the code as a typed graph structure consisting of *entities* (i.e., nodes) and *references* (i.e., arcs). The schema of the graph is essentially a *middle-level model* (Lethbridge et al. 2004). It has *entity types* such as File, Function, Object (i.e., for local and global variables), Macro and Typedef; and *reference types* such as Include, Declare, Define, Use, and Set. Figure 10 gives an example of C code and the corresponding graph model. As can be seen, a File entity (here `example.c`) references all entities (directly or transitively) that are contained in the file. For each reference, there is a corresponding back-reference (not shown in Fig. 10).

Besides the graph model, Understand also provides a low-level API that gives access to the underlying token stream. There are different kinds of tokens (or lexemes): Keyword, Identifier, Literal, Punctuation, Whitespace, Comment, etc. A lexeme has a string containing the source code characters it represents and a set of attributes including precise location information (starting and ending line and column numbers). The token stream preserves all properties of the source text, allowing an identical pretty-printing of the original if desired. The low-level representation is connected to the graph model: If applicable, a lexeme refers to its corresponding entity or reference.

4.3.2 Rationale and experiences with Understand

In this section, we outline the reason for choosing Understand and first experiences in using it to implement system-specific analyses.

Our initial choice for Understand was mostly based on our existing expertise that we gained by implementing several sophisticated analyses with it.[12] One advantage is

[12] The most complex analysis that we realized with Understand is a program slicer implemented by the second author (Kraft 2010).

Understand's support for Perl scripting, which enables *rapid application development* (Ousterhout 1998). As mentioned before, this is an important concern in our setting because we expect to implement many different analyses that need to be iteratively refined based on feedback of the industrial partner. From our experiences, we also had a favorable impression of Understand in terms of robustness and scalability, which was confirmed when we used Understand on the ABB system. Understand turns out to be very robust, processing the ABB system out-of-the-box. It is fast as well, requiring only 118 seconds to process 1.1 MLOC contained in 1,699 source files. A subset of the system (183 KLOC, 416 files) is processed in 16 seconds. According to our experiences, the runtime is almost linear for systems that do not exceed 2.5 MLOC.

A key concern when implementing analyses with Understand is the API's usefulness and usability. In terms of usability, important attributes are how easy it is to learn, how efficiently it can be used to accomplish certain tasks, how easy it is to remember the usage, and so on (McLellan et al. 1998). Understand's APIs have an agreeable learning curve, also thanks to many sample scripts provided on SciTools' web site. Still, we also had to write smaller test cases to explore how certain constructs in the code are represented in the underlying data model. Understand's graph model (cf. Sect. 4.3.1) is easy to navigate and query, and sufficiently rich to accommodate many kinds of analyses. For example, it is possible to get the file includes for the Task Include Structure analysis (cf. Sect. 4.2.1), to get the composition of data structures for the Task IPC Data Structure analysis (cf. Sect. 4.2.2), to get functions calls for the Task Scheduling Priority analysis (cf. Sect. 4.2.4) and Visibility Constrains analysis (cf. Sect. 4.2.6), and to get set/use information for variables for the IPC Message In/Out Parameters analysis (cf. Sect. 4.2.5).

Analyses also combine the graph model and the token stream. For example, the IPC Message In/Out Parameters analysis can use the graph model to get `struct`/`union` members and then use the token stream to identify the corresponding code comments. Similarly, the Visibility Constrains analysis can use the graph model to get function declarations and then access the token stream to see if there is a macro with a visibility constraint before the function.

While the combination of both APIs can be used to good effect, it has its limitations. On the one hand, the graph model does not expose detailed information, especially the code structure in function bodies is missing. On the other hand, the token stream is too low level to identify concepts that can be typically found in abstract syntax trees such as statements and declarations. This "gap" in the APIs makes it difficult to implement more sophisticated analyses that require control-flow or data-flow information.

We have started to implement additional infrastructure that augments the existing APIs to mitigate their deficiencies. We leverage a Perl-based parser, Parse::RecDescent,[13] to specify patterns in the token stream. These patterns can be simple ones similar to regular expression or more complex ones using recursive structures. We wrote a generic *island grammar* (Klusener and Lämmel 2003) with Parse::RecDescent that can be instantiated with a concrete pattern specification. The island grammar finds all matching patterns (the "islands") with tokens that are of no interest in between (the "water").

For example, a island grammar is used to identify `case` labels such as needed for the Task IPC Dispatch Loop analysis (cf. Sect. 4.2.3). This grammar simply defines this pattern: a case label followed by an identifier (which corresponds to an `enum` constant in our case) followed by a colon. The pattern and parser instantiation is six lines of code. We

[13] http://www.search.cpan.org/dist/Parse-RecDescent/.

are also currently developing an island grammar that recovers statements and constructs a control-flow graph so that analyses requiring control-flow sensitivity can be realized. However, it is a concern that Parse::RecDescent is not optimized for speed and uses backtracking, which slows down the parse. It remains to be seen if this approach provides sufficient scalability for the ABB system.

5 Discussion: lessons learned and proposed hypotheses

In the following, we synthesize lessons learned based on our case study and provide recommendations. We state our observations and recommendations as working hypotheses that need to be further refined and strengthened (or refuted) with more case studies and experiences.

Even though we can rely on a single case study only, our confidence is increased by our believe that the kind of analyses that we have identified in our case study are representative of other complex industrial systems implemented in C. This belief is based on our knowledge of four other industrial systems in this domain (Kraft et al. 2010).

5.1 Analysis kinds and tools requirements

5.1.1 Hypothesis: System-specific analyses can be classified into structural, control-flow based, and data-flow based

Table 2 shows the case study's analyses structured by the kind of techniques—structural, control flow, or data flow—that are required to detect the rules. In the subsequent discussion, it will become apparent that this is a useful distinction.

For structural analyses, it is typically sufficient to identify properties that are associated with a single code entity. For example, a code entity could be a function call (e.g., analysis (4)), C identifier (e.g., analysis (2)), or file name (e.g., analysis (1)) and the associated property can be its name (e.g., analyses (1)/(2)/(4)) or type. We also identify an analysis as

Table 2 Case study analyses and their characteristics

Analysis	Structural	Control flow	Data flow
(1) Task include structure	File naming rules		
	Include file rules		
(2) Task IPC data structures	Identifier naming rules (enum/struct/union)		
	Data type structure rules (enum/struct/union)		
(3) Task IPC dispatch loop	Statement structure rules (switch)		
(4) Task scheduling priority	Function call rules	Guard statement rules (if)	
(5) IPC message in/out parameters	Comments as semantic rules		Variable set/use rules
(6) Visibility constraints	Macros as semantic rules		
(7) Task and semaphore dependencies	Semaphore calls		Shared variable set/use critical section

structural if code entities are constrained by their locations. For example, certain code entities that are contained in certain files (e.g., as in analysis (1)) or that follow consecutively (e.g., a field declaration followed by a comment as in analysis (5) or a macro name following a function declaration as in analysis (6)).

Analyses that require an analysis of the control flow typically look at nesting/containment relationship in the code. For example, analysis (4) checks for function calls that change the task's priority. The analysis is refined by checking if calls are enclosed by a conditional statement (e.g., to protect against a certain state, in this case an empty message queue). Analyses that require data flow typically check if a certain constraint holds through execution paths. For example, in analysis (7) an inter-procedural flow analysis [similar to the one by Ceccato and Tonella (2010)] is required to identify critical sections in the code.

5.1.2 Hypothesis: System-specific analyses are often structural in nature

In our case study, the majority of analyses are structural. They either are purely structural or are a combination of structural and control/data. It seems that due to C's limitations in expressing constraints within the code (e.g., via the type system), often structural rules are introduced that developers and maintainers are supposed to follow. These rules involve naming, annotations with comments or macros, code templates, code couplings, etc. It is important to note that for the case study, we have first identified the analyses, then classified them and looked how to implement them in Understand. Thus, our proposed analyses are not constrained a priori by our chosen tool infrastructure.

This hypothesis is supported by work from Matsumura et al. (2002). In their study, they also first established system-specific analyses for an industrial C system based on failure reports. They then tried to encode the analyses based on a code-based pattern description language. The sample patterns that they give are all structural. For example, in one failure report the following rule was violated: Call of function F without setting a new value to global variable V first. This constraint is realized by checking that an assignment statement to V exists before F is called with arbitrary statements in between. Thus, control flow is not taken into account, only statement order. Also, the pattern only matches code within a function (i.e., no inter-procedural flow analysis).

The API-specific checks of Smalltalk code by Renggli et al. (2010) also suggest that structural analyses are often sufficient. They have implemented "30 rules working at the level of the abstract syntax tree." The vast majority of these rules seem quite simple in nature (e.g., that a certain message must be sent last in a cascade of message sends, or no invocations of certain messages are allowed because such an invocation constitutes a possible bug, deprecated API use, or a portability problem).

5.1.3 Hypothesis: System-specific analyses become increasingly costly (e.g., in terms of specification, verification, and maintenance) with structural, control-flow, and data-flow techniques. Thus, one should strive for structural analyses

Not surprisingly, analyses become increasingly sophisticated with structural, control-flow, and data-flow techniques. System-specific analyses strive for low overhead and consequently most of the case study's analyses are structural. With increasing sophistication, the specification effort increases. For example, structural constraints are easier to describe and more intuitive than data-flow equations. The simplicity of the specification is also important when the analysis detects a violation, because a system developer should be able to decide easily if the reported problem is a false positive or not. Similar considerations

hold for the verification and maintenance of the analyses. This hypothesis is supported by the developers of the Coverity tool: "At the most basic level, errors found with little analysis are often better than errors found with deeper tricks. A good error is probable, a true error, easy to diagnose; best is difficult to misdiagnose. As the number of analysis steps increases, so, too, does the chance of analysis mistake, user confusion, or the perceived improbability of event sequence."

If we assume that most analyses are structural then the expertise of the analysis developers should concentrate on this area. Also, the tools support should be most effective for structural analyses. Conversely, this means that analyses that use control/data flow can pose an additional burden on the analysis developers.

5.1.4 Hypothesis: Often system-specific analyses are incrementally evolved. A refined analysis often becomes more sophisticated, possibly changing the analysis kind

Once a candidate for a system-specific analysis has been identified, it should be implemented simply and cheaply. The results that the analysis produces will typically trigger a refinement to reduce the number of reported rule violations and/or the number of false positives. For example, imagine an initial analysis that simply looks for calls to a certain function (e.g., because these calls have a high overhead). The results may indicate that the analysis needs to be refined further. For example, calls in certain files are excluded or calls that have a certain number of arguments. A further refinement may look at the control flow, for instance, whether the call is enclosed by loops. Yet another refinement may use value range analysis to determine the upper bound of the loops' iterations. Since each refinement is costly, there needs to be a rational for it. Generally, the refinement of system-specific analyses is embedded within a process and controlled by stakeholders (see below).

5.1.5 Hypothesis: Ideally, the tool infrastructure for system-specific analyses should support all analysis kinds. In practice, there should be good tool support for structural analyses and some support to realize control/data flow analyses

The implementation effort is not much different for sophisticated analyses if the right tool infrastructure is used. For example, there are tools such as Metal that allow the specification of data-flow equations at a high abstraction level. It may be beneficial to have several tools that cover the different kinds of analyses. However, in practice it is probably rarely feasible to use more than one tool. If only a single tools is used, it should have (rudimentary) coverage for all analysis kinds. This is not always the case. For example, reverse engineering tools that target a middle-level or architectural meta-model (e.g., GReQL, Semantic Grep, and IntensiVE) provide no mechanism to access more fine-grained information. The kinds of analyses that we and Matsumura et al. (2002) have identified indicate that rudimentary tools such as `grep` and Perl are not a good choice because they reasonably cover only very simple structural analyses that look for a single entity.

Since system-specific analyses are typically structural (see above hypothesis), the selected tool should have good support for this analysis kind. A caveat here is that tools that primarily target structural constraints often have no or limited support for control and data flow. Understand is such an example. Structural constraints that are based on entity and reference types that are available in Understand's graph model are easy to specify. Structural constraints that are outside the graph model (e.g., comments and `case` labels) require more implementation effort but can still be realized with acceptable effort. Moving beyond structural constraints required us to implement functionality on top of Understand

that turned out to be a significant investment. We have also started with support for control-flow based analyses, but the effort is rather prohibitive. Still, while Understand has no or little support for control/flow-based analyses, it does not preclude it. From this perspective Understand is a reasonable choice but far from ideal.

5.2 Introduction and adoption in industry

5.2.1 Hypothesis: System-specific analyses have to be introduced incrementally into an existing industrial project (e.g., starting with a pilot study)

Introducing system-specific analyses to an existing system is both incremental and iterative. System-specific analyses should be introduced with a "chicken little" approach that slowly builds up expertise. Later on, more (sophisticated) analyses are gradually introduced over time as the need arises and as resources are available to do so.

At the beginning, an exploratory pilot study can identify a few analyses that are easy to realize. The task scheduling priority in our case study (cf. Sect. 4.2.4) is such an example because only the names of function calls have to be matched. At this stage, the focus is mostly on the integration of system-specific analysis into the existing maintenance process (e.g., when should they be run and by whom) and into the build process. The latter is not trivial if builds use custom-made solutions that are often not fully understood anymore (such as in our case study).

Another concern that should be explored early on is whether scalability requirements can be met by the chosen tooling infrastructure. For industrial systems, such as in our case study, the effective processing of a large code base is important. Depending on the development approach, it may be desirable to run an analysis in, say, less than an hour rather than a nightly batch run.

5.2.2 Hypothesis: System-specific analyses change the development/maintenance process and require new roles in the process

As already discussed, each analysis will have to be iteratively refined, starting out with a first prototype to assess its viability, then tuning it to meet the demands of the production environment, and finally maintaining it along with the evolving system. All these stages need to be articulated and interlocked with the system's development process.

At this point, we are developing the analyses of the case study ourselves. However, system-specific analyses can only reach their full potential when they are developed within the company. In practice, only a smaller subset of the company's developers that have been trained accordingly would write such analyses. Still, to accommodate this, it is necessary to introduce a new role in the process. This new (semi-formal) role shares characteristics with *local developers* (Gantt and Nardi 1992) in the sense that local developers are familiar with the target system (being active or former developers) and also have expertise in developing system-specific analyses.

5.2.3 Hypothesis: Each system-specific analysis needs to support a concrete problem or concern

At the inital pilot state, the focus is on the introduction of system-specific analyses and less on the effectiveness of each individual analysis. Once the pilot stage has been successfully

completed, the focus shifts to the identification of analyses that promise to have a high impact on "business" concerns (e.g., to improve upon a quality or to point out potential faults). Generally, the approaches and motivation of system-specific analyses are often similar to the ones found in more generic solutions. For example, the motivation to reduce faults is common to style/bug checkers and system-specific analyses.

For example, Matsumura et al. (2002) base the identification of system-specific rules on three types of failure reports (i.e., failure occurrence, failure solution, and file update reports). The maintainers of the system introduce a rule if "there is a possibility of injecting other faults caused by the same rules in the future." Thus, this approach strives to reduce the number of faults and failures. Another example are checks that identify non-portable code between environments or compilers—Renggli et al. (2010) have environment-specific checks for non-portable code between different Smalltalk platforms/versions. For our case study, one could also imagine such checks because the system is hosted by two different environments: one for testing (Windows) and another one in production (VxWorks). Examples of other promising candidates for analyses are:

Naming conventions: Matsumura et al. (2002) say that in their case, as in many other large industrial system, there is "a strict coding rule for deciding the names of the functions and variables." For instance, we know of one system in which the functions of a module should use the module name as prefix. Violations do not cause bugs directly, but indirectly impact quality due to lower maintainability.

Internal (coding) guidelines: Many companies have their own programming guidelines, which have emerged over the years. Typically, these are not enforced using automatic tools but are merely listed in some document, which the developers are supposed to know and follow. An interesting example that we know of is that every if-statement should have an explicit else-block, to avoid unhandled cases.

External requirements: For instance, for complex embedded systems there are typically safety standards that need to be met. One of our industrial partners has developed a software controlled vehicle braking system ("brake-by-wire"). To be able to certify this system, they needed to document several code characteristics, including the absence of "knots" (i.e., functions with multiple return points).

5.2.4 Hypothesis: System-specific analyses face significant adoption hurdles that need to be studied

Getting system-specific analyses integrated into an established process faces many potential adoption hurdles (Favre et al. 2003). There are established theories that help to understand the mechanisms of adoption and to influence it positively. For example, there is the model of technology transfer by Pfleeger (1999). The adoption of innovations can be promoted by so-called change agents or innovation champions (Rogers 1995, page 398). Lethbridge and Singer (unknowingly) did follow this strategy for getting one of their tools adopted: "We had significant difficulty introducing new tools. One technique that seems to hold promise is to train a single individual (in our case somebody new to the organization) and have him or her act as a consultant for others" (Lethbridge and Singer 1997). We plan to use such strategies for our case study—especially to identify a champion within the company—, but this is currently not a research focus of ours. The pilot stage mentioned above also helps to establish the buy-in of the stakeholders. Typically, stakeholders realize the need to improve the system's quality. To help adoption, system-specific analyses should support concrete development or business goals and provide evidence that introducing them into the

development process will have a positive impact on these goals. The Goal/Question/Metric (GQM) paradigm (Basili 1992) could be used for this purpose because it provides traceability from (business) goals down to code properties. Since companies in the industrial embedded domain have already invested in static analyses, there is already initial experiences and expertise which system-specific analyses can built upon.

6 Conclusions

System-specific analyses are the most specific kind in the spectrum of analyses because they target one particular system. The most prominent drawback of system-specific analyses—being highly specific—is also their greatest potential because they can be tailored to meet the requirements of the development project. In contrast to system-specific analyses, generic code and style checkers suffer much more from the problem that they may produce many false positives. The goal of our system-specific analyses is to have a positive impact on key concerns such as improving quality attributes of the system and reducing defects in the code that can lead to faults.

A key goal of our current research has been to establish the case for system-specific analyses. We have presented a case study based on an industrial system from ABB Robotics, which is a large and complex embedded system. The identified analyses in the case study illustrate that system-specific analyses can potentially tackle a wide range of coding and design concerns. There are analyses that leverage existing (design) annotations in the form of comments or macro definitions; there are coding constraints that can be contained within a file or function, or that span multiple files; there are mostly quite simple analyses (e.g., operating only on the names of function calls), but also more sophisticated ones that require a limited form of control-flow and/or data-flow information; and there are analyses that cover critical concerns of the domain such as scheduling and shared data for embedded systems.

We are hoping to deploy selected analyses at ABB Robotics and to empirically verify whether system-specific analyses indeed have a positive impact on the system's quality. In order to be effective, system-specific analyses have to be incorporated into the already established development process, which is a major adoption challenge we are facing. Developers in the embedded domain use a whole range of techniques such as static and dynamic analyses, inspections and reviews, metrics, and testing to monitor, assess, and increase the quality of their systems. Indeed, Ebert and Jones (2009) emphasize that "embedded-software engineers must know and use a richer combination of defect prevention and removal activities than other software domains." From this perspective, system-specific analysis should be attractive to the embedded systems domain because they promise to complement already established techniques.

Based on the results of the case study and our experiences with other systems in the complex embedded system domain, we propose a number of working hypotheses for system-specific analyses. These hypotheses address the characteristics of system-specific analyses and tooling for them as well as strategies for industrial adoption. These hypotheses can be seen as a challenges for our future research.

Acknowledgments Many thanks to the anonymous reviewers whose thorough comments greatly helped to improve the paper. This work is supported by the Swedish Foundation for Strategic Research through the PROGRESS Centre for Predictable Embedded Software Systems, which is a part of Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University in Västerås, Sweden.

References

Adams, E. N. (1984). Optimizing preventive service of software products. *IBM Journal of Research and Development, 28*(1), 2–14.

Åkerholm, M., Land, R., & Strzyz, C. (2009). Can you afford not to certify your control system? *iVTinternational* http://www.ivtinternational.com/legislative_focus_nov.ph.

Andersson, J. (2005). *Modeling the temporal behavior of complex embedded systems*. Licentiate thesis, Mälardalen University, Sweden.

Anderson, P. (2008). Detecting bugs in safety-critical code. *Dr Dobb's Journal* http://www.grammatech.com/products/codesonar/DrDobbsDetectingBugsInSafetyCriticalCode.pdf.

Antoniol, G., Penta, M. D., Masone, G., & Villano, U. (2003). XOgastan: XML-oriented gcc AST analysis and transformations. In *3rd IEEE international workshop on source code analysis and manipulation (SCAM'03)* (pp. 173–182).

Atkinson, D. C., & Griswold, W. G. (2006). Effective pattern matching of source code using abstract syntax patterns. *Software—Practice and Experience, 36*(4), 413–447

Basili, V. R. (1992). *Software modeling and measurement: The goal/question/metric paradigm*. Tech. Rep. CS-TR-2957, University of Maryland, http://www.cs.umd.edu/basili/publications/technical/T78.pdf.

Bessey, A., Block, K., Chelfs, B., Chou, A., Fulton, B., Hallem, S., et al. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM, 53*(2), 66–75.

Boogerd, C., & Moonen, L. (2008). Assessing the value of coding standards: An empirical study. In *24th IEEE international conference on software maintenance (ICSM'08)* (pp. 277–286).

Boogerd, C., & Moonen, L. (2009). Evaluating the relation between coding standard violations and faults within and across software versions. In *6th Working conference on mining software repositories (MSR'09)* (pp. 41–50).

Brichau, J., Kellens, A., Castro, S., & D'Hondt, T. (2010). Enforcing structural regularities in software using IntensiVE. *Science of Computer Programming, 75*(4), 232–246.

Bull, R. I., Trevors, A., Malton, A. J., & Godfrey, M. W. (2002). Semantic grep: Regular expressions + relational abstraction. In *9th IEEE working conference on reverse engineering (WCRE'02)* (pp. 267–276).

Ceccato, M., Dean, T. R., Tonella, P., & Marchignoli, D. (2008a). Data model reverse engineering in migrating a legacy system to Java. In *15th IEEE working conference on reverse engineering (WCRE'08)* (pp. 177–186).

Ceccato, M., Tonella, P. (2010). Static analysis for enforcing intra-thread consistent locks in the migration of a legacy system. In *26th IEEE international conference on software maintenance (ICSM'10)*.

Ceccato, M., Tonella, P., & Matteotti, C. (2008b). Goto elimination strategies in the migration of legacy code to Java. In *12th IEEE European conference on software maintenance and reengineering (CSMR'08)* (pp. 53–62).

Chelf, B., Ebert, C. (2009). Ensuring the integrity of embedded software with static code analysis. *IEEE Software, 26*(3), 96–99.

Dean, T. R., Malton, A. J., & Holt, R. (2001). Union schemas as a basis for a C++ extractor. In *8th IEEE working conference on reverse engineering (WCRE'01)* (pp. 59–67).

Eaton, C., & Memon, A. M. (2007). An empirical approach to testing web applications across diverse client platform configurations. *International Journal on Web Engineering and Technology, 3*(3), 227–253. http://www.cs.umd.edu/atif/papers/EatonIJWET2007.pdf.

Ebert, C., & Jones, C. (2009). Embedded software: Facts, figures and future. *IEEE Computer, 42*(4), 42–52.

Ebert, C., & Salecker, J. (2009). Embedded software—technologies and trends. *IEEE Software, 26*(3), 14–18.

Engler, D., & Ashcraft, K. (2003). RacerX: Effective, static detection of race conditions and deadlocks. In *19th ACM symposium on operating system principles (SOSP'03)* (pp. 237–252).

Favre, J., Estublier, J., & Sanlaville, R. (2003). Tool adoption issues in a very large software company. In *3rd International workshop on adoption-centric software engineering (ACSE'03)* (pp. 81–89).

Gantt, M., & Nardi, B. A. (1992). Gardeners and gurus: Patterns of cooperation among CAD users. In *ACM SIGCHI conference on human factors in computing systems (CHI'92)* (pp. 107–117).

Hallem, S., Chelf, B., Xie, Y., & Engler, D. (2002). A system and language for building system-specific static analyses. In *ACM SIGPLAN conference on programming language design and implementation (PLDI'02)* (pp. 69–83).

Hankin, C. (1998). Program analysis tools. *International Journal on Software Tools for Technology Transfer, 2*(1), 6–12.

Harman, M., & Hierons, R. M. (2001). An overview of program slicing. *Software Focus, 2*(3), 85–92.

Hatton, L. (2004). Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology, 46*(7), 465–472.
Hatton, L. (2005). EC—a measurement based safer subset of ISO C suitable for embedded system development. *Information and Software Technology, 47*(7), 181–187.
Hills, C. (2005). C-change for safety critical systems. *IEEE Electronics Systems and Software, 3*(1), 28–31.
Holzmann, G. J. (2006). The power of 10: Rules for developing safety-critical code. *IEEE Computer, 39*(6), 95–97.
Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. In *Companion to the 19th ACM object-oriented programming, systems, languages, and applications (OOPSLA'04)* (pp. 92–106).
Jackson, D., & Rinard, M. (2000). Software analysis: A roadmap. In *Conference on the future of software engineering* (pp. 135–145).
Kienle, H. M., & Müller, H. A. (2001). Leveraging program analysis for web site reverse engineering. In *3rd IEEE international workshop on web site evolution (WSE'01)* (pp. 117–125).
Kienle, H. M., & Müller, H. A. (2010). The tools perspective on software reverse engineering: Requirements, construction and evaluation. *Advances in Computers, 79*, 189–290.
Klusener, S., & Lämmel, R. (2003). Deriving tolerant grammars from a base-line grammar. In *19th IEEE international conference on software maintenance (ICSM'03)* (pp. 179–188).
Kraft, J. (2010). Enabling timing analysis of complex embedded systems. PhD thesis no. 84, Mälardalen University, Sweden, http://www.mdh.diva-portal.org/smash/get/diva2:312516/FULLTEXT0.
Kraft, J., Wall, A., & Kienle, H. (2010). *1st International conference on runtime verification (RV 2010)*, Lecture Notes in Computer Science (Vol. 6418, pp. 315–329), Springer, chap Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects.
Kremenek, T., Ashcraft, K., Yang, J., & Engler, D. (2004). Correlation exploitation in error ranking. In *12th ACM SIGSOFT international symposium on foundations of software engineering (FSE-12)* (pp. 83–93).
Kullbach, B., & Winter, A. (1999). Querying as an enabling technology in software reengineering. In *3rd IEEE European conference on software maintenance and reengineering (CSMR'99)* (pp. 42–50).
Lethbridge, T. C., & Singer, J. (1997). Understanding software maintenance tools: Some empirical research. In *3rd Workshop on empirical studies of software maintenance (WESS'97)* (pp. 157–162).
Lethbridge, T. C., Tichelaar, S., & Ploedereder, E. (2004). The Dagstuhl middle metamodel: A schema for reverse engineering. In J. M. Favre, M. Godfrey, & A. Winter (Eds.), *International workshop on meta-models and schemas for reverse engineering (ateM'03)*, Electronic Notes in Theoretical Computer Science (Vol. 94, pp. 7–18). Amsterdam: Elsevier.
Lv, M., Guan, N., Zhang, Y., Deng, Q., Yu, G., & Zhang, J. (2009). A survey of WCET analysis of real-time operating systems. In *2009 IEEE international conference on embedded software and systems* (pp. 65–72).
Marinescu, C., & Jurca, I. (2006). A meta-model for enterprise applications. In *8th IEEE international symposium on symbolic and numeric algorithms for scientific computing (SYNASC'06)* (pp. 187–194).
Martin, F. (1998). PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer, 2*(1), 46–67.
Matsumura, T., Monden, A., & Matsumoto, K. (2002). The detection of faulty code violating implicit coding rules. In *IEEE international symposium on empirical software engineering (ISESE'02)* (pp. 173–182).
McDowell, C. E., & Helmbold, D. P. (1989). Debugging concurrent programs. *ACM Computing Surveys, 21*(4), 593–622.
McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998). Building more usable APIs. *IEEE Software, 15*(3), 78–86
Murphy, G. C., Notkin, D., Griswold, W. G., & Lan, E. S. (1998). An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology, 7*(2), 158–191.
Nagy, C., Pantos, J., Gergely, T., & Beszedes, A. (2010). Towards a safe method for computing dependencies in database-intensive systems. In *14th IEEE European conference on software maintenance and reengineering (CSMR'10)* (pp. 169–178).
Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *IEEE Computer, 31*(3), 23–30.
Perin, F., Girba, T., & Nierstrasz, O. (2010). Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *26th IEEE international conference on software maintenance (ICSM'10)*.
Pfleeger, S. L. (1999). Understanding and improving technology transfer in software engineering. *Journal of Systems and Software, 47*(2–3), 111–124.
Pinzger, M., Fischer, M., Gall, H., & Jazayeri, M. (2002). Revealer: A lexical pattern matcher for architecture recovery. In *9th IEEE working conference on reverse engineering (WCRE'02)* (pp. 170–178).

Pinzger, M., Oberleitner, J., & Gall, H. (2003). Analyzing and understanding architectural characteristics of COM+ components. In *11th IEEE international workshop on program comprehension (IWPC'03)* (pp. 54–63).

Power, J. F., & Malloy, B. A. (2002). Program annotation in XML: A parse-tree based approach. In *9th IEEE working conference on reverse engineering (WCRE'02)* (pp. 190–198).

Renggli, L., Ducasse, S., Gîrba, T., & Nierstrasz, O. (2010). Domain-specific program checking. In *48th International conference on objects, models, components and patterns (TOOLS 2010)* (pp. 213–232).

Rogers, E. M. (1995). *Diffusion of innovations* (4th ed.). New York: The Free Press.

Roy, C. K., & Cordy, J. R. (2008). Scenario-based comparison of clone detection techniques. In *16th IEEE international conference on program comprehension (ICPC'08)* (pp. 153–162).

RTCA. (1992). *Software considerations in airborne systems and equipment certification*. Standard RTCA/DO-17B, RTCA.

SciTools. (2004). *Perl and C application program interface for Understand: User guide and reference manual*. http://www.getunderstand.com/documents/manuals/pdf/understand_api.pdf.

Xie, Y., Naik, M., Hackett, B., & Aiken, A. (2005). Soundness and its role in bug detection systems. In *Workshop on the evaluation of software defect detection tools at PLDI 2005* http://www.cs.umd.edu/pugh/BugWorkshop05/papers/12-xie.pdf.

Author Biographies

Holger M. Kienle holds a PhD degree from the University of Victoria, Canada (2006), a Diploma in Computer Science from the University of Stuttgart, Germany (1999), and a Master of Science degree in Computer Science from the University of Massachusetts Dartmouth (1995). He received a two year Post Graduate Research Fellowship (1997–1998) from the University of California Santa Barbara to work as a member of Professor Hölzle's Object-Oriented Compilers group. He is currently a postdoctoral researcher in Computer Science at Mälardalen University (MDH), Västerås, Sweden, where he is a member of the PROGRESS National Strategic Research Centre; and the University of Victoria, Canada, where he is a member of Professor Müller's research group. His interests include software reverse engineering, domain-specific languages, virtual worlds, and legal issues that impact information technology. He is program co-chair for WSE 2010 and co-organizer of the WASDeTT workshop series. He has served on the program committees of CSMR, ICSM, WCRE and WSE.

Johan Kraft is a post-doctoral researcher in Computer Science at Mälardalen University, Sweden. Johan's doctoral work focused on approximate timing analysis of embedded software systems using discrete event simulation. His work has included simulation techniques, static analysis for generation of simulation models from complex embedded software systems, program slicing techniques, runtime tracing of embedded systems as well as static and dynamic software visualization. Johan has previously worked as embedded software developer at ABB Robotics. Johan is the founding director of Percepio AB, a university spinoff focusing on software visualization.

Thomas Nolte was awarded a B.Eng., M.Sc., Licentiate, and Ph.D. degree in Computer Engineering from Mälardalen University (MDH), Västerås, Sweden, in 2001, 2002, 2003, and 2006, respectively. He was a Visiting Researcher at University of California, Irvine (UCI), Los Angeles, USA, in 2002, and a Visiting Researcher at University of Catania, Italy, in 2005. He was a Postdoctoral Researcher at University of Catania in 2006, and at MDH in 2006–2007. Dr. Nolte became an Assistant Professor at MDH in 2008. He is now an Associate Professor in Computer Science at MDH since 2009. His research interests include predictable execution of embedded systems, design, modeling and analysis of real-time systems, multicore systems, distributed embedded real-time systems. Dr. Nolte is Program Leader of the PROGRESS National Strategic Research Centre, President of The Swedish National Real-Time Association (SNART), Vice-Chair of IEEE IES Technical Committee on Factory Automation (TCFA), and Co-Chair of IEEE IES TCFA Real-Time Fault Tolerant Systems Subcommittee. He is a Member of the Editorial Board of Elsevier's Journal of Systems Architecture: Embedded Software Design, since 2009; Guest Editor of the IEEE Transactions on Industrial Informatics, three times since 2008; General Co-Chair of IEEE Intl. Symposium on Industrial Embedded Systems (SIES) 2011; Co-Organizer of intl. events including the Intl. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS), 2008, 2009 and 2010 (Co-founder with Dr. Insik Shin, co-located with the IEEE Real-Time Systems Symposium (RTSS)); Program Co-Chair of IEEE Intl. Workshop on Factory Communication Systems (WFCS), 2012, IEEE Intl. Symposium on Industrial Embedded Systems (SIES), 2012, IEEE Intl. Conference on Emerging Technologies and Factory Automation (ETFA), the Real-Time and (Networked) Embedded Systems track, 2008, 2009, 2010 and 2011; PC-member of more than 50 technical program committees for intl. conferences and workshops, including top events in real-time systems: RTSS, ECRTS, RTAS and RTCSA, and factory automation: ETFA and WFCS. Dr. Nolte has been reviewer of hundreds of scientific papers submitted to 13 different intl. journals, and a total of 95 editions of 39 different intl. conferences and workshops.