Software Reverse Engineering in the Domain of Complex Embedded Systems

Holger M. Kienle[1], Johan Kraft[1] and Hausi A. Müller[2]
[1]*Mälardalen University*
[2]*University of Victoria*
[1]*Sweden*
[2]*Canada*

1. Introduction

This chapter focuses on tools and techniques for software reverse engineering in the domain of complex embedded systems. While there are many "generic" reverse engineering techniques that are applicable across a broad range of systems (e.g., slicing (Weiser, 1981)), complex embedded system have a set of characteristics that make it highly desirable to augment these "generic" techniques with more specialized ones. There are also characteristics of complex embedded systems that can require more sophisticated techniques compared to what is typically offered by mainstream tools (e.g., dedicated slicing techniques for embedded systems (Russell & Jacome, 2009; Sivagurunathan et al., 1997)). Graaf et al. (2003) state that "the many available software development technologies don't take into account the specific needs of embedded-systems development ... Existing development technologies don't address their specific impact on, or necessary customization for, the embedded domain. Nor do these technologies give developers any indication of how to apply them to specific areas in this domain." As we will see, this more general observations applies to reverse engineering as well.

Specifically, our chapter is motivated by the observation that the bulk of reverse engineering research targets software that is outside of the embedded domain (e.g., desktop and enterprise applications). This is reflected by a number of existing review/survey papers on software reverse engineering that have appeared over the years, which do not explicitly address the embedded domain (Canfora et al., 2011; Confora & Di Penta, 2007; Kienle & Müller, 2010; Müller & Kienle, 2010; Müller et al., 2000; van den Brand et al., 1997). Our chapter strives to help closing this gap in the literature. Conversely, the embedded systems community seems to be mostly oblivious of reverse engineering. This is surprising given that maintainability of software is an important concern in this domain according to a study in the vehicular domain (Hänninen et al., 2006). The study's authors "believe that facilitating maintainability of the applications will be a more important activity to consider due to the increasing complexity, long product life cycles and demand on upgradeability of the [embedded] applications."

Embedded systems are an important domain, which we opine should receive more attention of reverse engineering research. First, a significant part of software evolution is happening in this domain. Second, the reach and importance of embedded systems are growing with emerging trends such as ubiquitous computing and the Internet of Things. In this chapter we specifically focus on *complex embedded systems*, which are characterized by the following properties (Kienle et al., 2010; Kraft, 2010):

- large code bases, which can be millions of lines of code, that have been maintained over many years (i.e., "legacy")
- rapid growth of the code base driven by new features and the transition from purely mechanical parts to mechatronic ones
- operation in a context that makes them safety- and/or business-critical

The rest of the chapter is organized as follows. We first introduce the chapter's background in Section 2: reverse engineering and complex embedded systems. Specifically, we introduce key characteristics of complex embedded systems that need to be taken into account by reverse engineering techniques and tools. Section 3 presents a literature review of research in reverse engineering that targets embedded systems. The results of the review are twofold: it provides a better understanding of the research landscape and a starting point for researchers that are not familiar with this area, and it confirms that surprisingly little research can be found in this area. Section 4 focuses on timing analysis, arguably the most important domain-specific concern of complex embedded systems. We discuss three approaches how timing information can be extracted/synthesized to enable better understanding and reasoning about the system under study: executing time analysis, timing analysis based on timed automata and model checking, and simulation-based timing analysis. Section 5 provides a discussion of challenges and research opportunities for the reverse engineering of complex embedded systems, and Section 6 concludes the chapter with final thoughts.

2. Background

In this section we describe the background that is relevant for the subsequent discussion. We first give a brief introduction to reverse engineering and then characterize (complex) embedded systems.

2.1 Reverse engineering

Software reverse engineering is concerned with the analysis (not modification) of an existing (software) system (Müller & Kienle, 2010). The IEEE Standard for Software Maintenance (IEEE Std 1219-1993) defines reverse engineering as "the process of extracting software system information (including documentation) from source code." Generally speaking, the output of a reverse engineering activity is synthesized, higher-level information that enables the reverse engineer to better reason about the system and to evolve it in a effective manner. The process of reverse engineering typically starts with lower levels of information such as the system's source code, possibly also including the system's build environment. For embedded systems the properties of the underlying hardware and interactions between hardware and software may have to be considered as well.

When conducting a reverse engineering activity, the reverse engineer follows a certain process. The workflow of the reverse engineering process can be decomposed into three subtasks: extraction, analysis, and visualization (cf. Figure 1, middle). In practice, the reverse engineer has to iterate over the subtasks (i.e., each of these steps is repeated and refined several times) to arrive at the desired results. Thus, the reverse engineering process has elements that make it both ad hoc and creative.

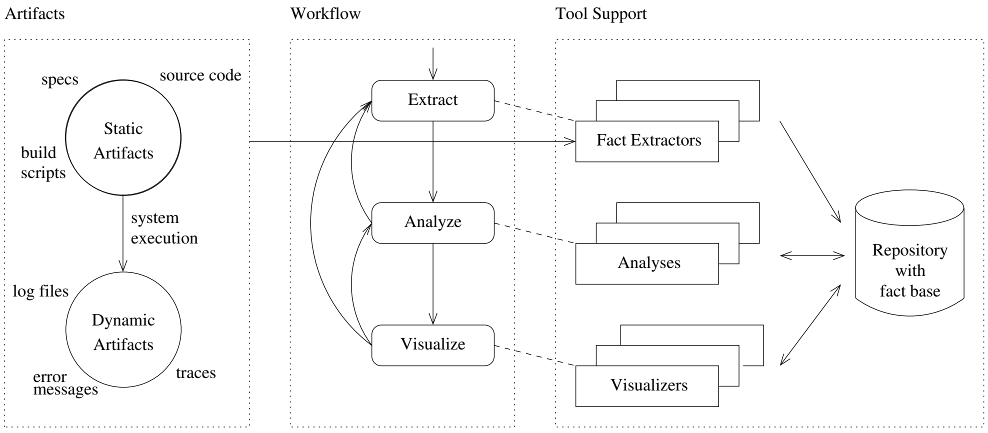

Fig. 1. High-level view of the reverse engineering process workflow, its inputs, and associated tool support.

For each of the subtasks tool support is available to assist the reverse engineer (cf. Figure 1, right). From the user's point of view, there may exist a single, integrated environment that encompasses all tool functionality in a seamless manner (tight coupling), or a number of dedicated stand-alone tools (weak coupling) (Kienle & Müller, 2010). Regardless of the tool architecture, usually there is some kind of a (central) repository that ties together the reverse engineering process. The repository stores information about the system under scrutiny. The information in the repository is structured according to a model, which is often represented as a data model, schema, meta-model or ontology.

When extracting information from the system, one can distinguish between static and dynamic approaches (cf. Figure 1, left). While static information can be obtained without executing the system, dynamic information collects information about the running system. (As a consequence, dynamic information describes properties of a single run or several runs, but these properties are not guaranteed to hold for all possible runs.) Examples of static information are source code, build scripts and specs about the systems. Examples of dynamic information are traces, but content in log files and error messages can be utilized as well. It is often desirable to have both static and dynamic information available because it gives a more holistic picture of the target system.

2.2 Complex embedded systems

The impact and tremendous growth of embedded systems is often not realized: they account for more than 98% of the produced microprocessors (Ebert & Jones, 2009; Zhao et al., 2003). There is a wide variety of embedded systems, ranging from RFID tags and household appliances over automotive components and medical equipment to the control of nuclear power plants. In the following we restrict our discussion mostly to complex embedded systems.

Complex embedded software systems are typically special-purpose systems developed for control of a physical process with the help of sensors and actuators. They are often mechatronic systems, requiring a combination of mechanical, electronic, control, and computer engineering skills for construction. These characteristics already make it apparent that complex embedded systems differ from desktop and business applications. Typical non-functional requirements in this domain are safety, maintainability, testability, reliability and robustness, safety, portability, and reusability (Ebert & Salecker, 2009; Hänninen et al., 2006). From a business perspective, driving factors are cost and time-to-market (Ebert & Jones, 2009; Graaf et al., 2003).

While users of desktop and web-based software are accustomed to software bugs, users of complex embedded systems are by far less tolerant of malfunction. Consequently, embedded systems often have to meet high quality standards. For embedded systems that are safety-critical, society expects software that is free of faults that can lead to (physical) harm (e.g., consumer reaction to cases of unintended acceleration of Toyota cars (Cusumano, 2011)). In fact, manufacturers of safety-critical devices have to deal with safety standards and consumer protection laws (Åkerholm et al., 2009). In case of (physical) injuries caused by omissions or negligence, the manufacturer may be found liable to monetarily compensate for an injury (Kaner, 1997). The Economist claims that "product-liability settlements have cost the motor industry billions" (The Economist, 2008), and Ackermann et al. (2010) say that for automotive companies and their suppliers such as Bosch "safety, warranty, recall and liability concerns ... require that software be of high quality and dependability."

A major challenge is the fact that complex embedded systems are becoming more complex and feature-rich, and that the growth rate of embedded software in general has accelerated as well (Ebert & Jones, 2009; Graaf et al., 2003; Hänninen et al., 2006). For the automotive industry, the increase in software has been exponential, starting from zero in 1976 to more than 10 million lines of code that can be found in a premium car 30 years later (Broy, 2006). Similar challenges in terms of increasing software are faced by the avionics domain (both commercial and military) as well; a fighter plane can have over 7 million lines of code (Parkinson, n.d.) and alone the flight management system of a commercial aircraft's cockpit is around 1 million lines of code (Avery, 2011). Software maintainers have to accommodate this trend without sacrificing key quality attributes. In order to increase confidence in complex embedded systems, verification techniques such as reviews, analyses, and testing can be applied. According to one study "testing is the main technique to verify functional requirements" (Hänninen et al., 2006). Ebert and Jones say that "embedded-software engineers must know and use a richer combination of defect prevention and removal activities than other software domains" Ebert & Jones (2009).

Complex embedded systems are *real-time systems*, which are often designed and implemented as a set of tasks[1] that can communicate with each other via mechanisms such as message queues or shared memory. While there are off-line scheduling techniques that can guarantee the timeliness of a system if certain constraints are met, these constraints are too restrictive for many complex embedded systems. In practice, these systems are implemented on top of a real-time operating system that does online scheduling of tasks, typically using preemptive fixed priority scheduling (FPS).[2] In FPS scheduling, each task has a scheduling priority, which typically is determined at design time, but priorities may also change dynamically during

[1] 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.

[2] An FPS scheduler always executes the task of highest priority being ready to execute (i.e., which is not, e.g., blocked or waiting), and when preemptive scheduling is used, the executing task is immediately preempted when a higher priority task is in a ready state.

run-time. In the latter case, the details of the temporal behavior (i.e., the exact execution order) becomes an emerging property of the system at run-time. Worse, many complex embedded systems are *hard real-time systems*, meaning that a single missed deadline of a task is considered a failure. For instance, for Electronic Control Units (ECUs) in vehicles as much as 95% of the functionality is realized as hard real-time tasks (Hänninen et al., 2006). The deadline of tasks in an ECU has a broad spectrum: from milliseconds to several seconds.

The real-time nature of complex embedded systems means that maintainers and developers have to deal with the fact that the system's correctness also depends on *timeliness* in the sense that the latency between input and output should not exceed a specific limit (the deadline). This is a matter of timing predictability, not average performance, and therefore poses an additional burden on verification via code analyses and testing. For example, instrumenting the code may alter its temporal behavior (i.e., probing effect (McDowell & Helmbold, 1989)). Since timing analysis arguably is the foremost challenge in this domain, we address it in detail in Section 4.

The following example illustrates why it can be difficult or infeasible to automatically derive timing properties for complex embedded systems (Bohlin et al., 2009). Imagine a system that has a task that processes messages that arrive in a queue:

```
do {
  msg = receive_msg(my_msg_queue);
  process_msg(msg);
} while (msg != NO_MESSAGE);
```

The loop's execution time obviously depends on the messages in the queue. Thus, a timing analysis needs to know the maximum queue length. It also may have to consider that other tasks may preempt the execution of the loop and add messages to the queue.

Besides timing constraints there are other resource constraints such as limited memory (RAM and ROM), power consumption, communication bandwidth, and hardware costs (Graaf et al., 2003). The in-depth analysis of resource limitations if often dispensed with by over-dimensioning hardware (Hänninen et al., 2006). Possibly, this is the case because general software development technologies do not offer features to effectively deal with these constraints (Graaf et al., 2003).

Even though many complex embedded systems are safety-critical, or at least business-critical, they are often developed in traditional, relatively primitive and unsafe programming languages such as C/C++ or assembly.[3] As a general rule, the development practice for complex embedded systems in industry is not radically different from less critical software systems; formal verification techniques are rarely used. Such methods are typically only applied to truly safety-critical systems or components. (Even then, it is no panacea as formally proven software might still be unsafe (Liggesmeyer & Trapp, 2009).)

Complex embedded systems are often legacy systems because they contain millions of lines of code and are developed and maintained by dozens or hundreds of engineers over many years.

[3] According to Ebert & Jones (2009), C/C++ and assembly is used by more than 80 percent and 40 percent of companies, respectively. Another survey of 30 companies found 57% use of C/C++, 20% use of assembly, and 17% use of Java (Tihinen & Kuvaja, 2004).

Thus, challenges in this domain are not only related to software development per se (i.e., "green-field development"), but also in particular to software maintenance and evolution (i.e., "brown-field development"). Reverse engineering tools and techniques can be used—also in combination with other software development approaches—to tackle the challenging task of evolving such systems.

3. Literature review

As mentioned before, surprisingly little research in reverse engineering targets embedded systems. (Conversely, one may say that the scientific communities of embedded and real-time systems are not pursuing software reverse engineering research.) Indeed, Marburger & Herzberg (2001) did observe that "in the literature only little work on reverse engineering and re-engineering of embedded systems has been described." Before that, Bull et al. (1995) had made a similar observation: "little published work is available on the maintenance or reverse engineering specific to [safety-critical] systems."

Searching on IEEE Xplore for "software reverse engineering" and "embedded systems" yields 2,702 and 49,211 hits, respectively.[4] There are only 83 hits that match both search terms. Repeating this approach on Scopus showed roughly similar results:[5] 3,532 matches for software reverse engineering and 36,390 for embedded systems, and a union of 92 which match both. In summary, less than 4% of reverse engineering articles found in Xplore or Scopus are targeting embedded systems.

The annual IEEE Working Conference on Reverse Engineering (WCRE) is dedicated to software reverse engineering and arguably the main target for research of this kind. Of its 598 publication (1993–2010) only 4 address embedded or real-time systems in some form.[6] The annual IEEE International Conference on Software Maintenance (ICSM) and the annual IEEE European Conference on Software Maintenance and Reengineering (CSMR) are also targeted by reverse engineering researchers even though these venues are broader, encompassing software evolution research. Of ICSM's 1165 publications (1993–2010) there are 10 matches; of CSMR's 608 publications (1997-2010) there are 4 matches. In summary, less than 1% of reverse engineering articles of WCRE, ICSM and CSMR are targeting embedded systems.

The picture does not change when examining the other side of the coin. A first indication is that overview and trend articles of embedded systems' software (Ebert & Salecker, 2009; Graaf et al., 2003; Hänninen et al., 2006; Liggesmeyer & Trapp, 2009) do not mention reverse engineering. To better understand if the embedded systems research community publishes reverse engineering research in their own sphere, we selected a number of conferences and journals that attract papers on embedded systems (with an emphasis on software, rather than hardware): Journal of Systems Architecture – Embedded Systems Design

[4] We used the advanced search feature (`http://ieeexplore.ieee.org/search/advsearch.jsp`) on all available content, matching search terms in the metadata only. The search was performed September 2011.

[5] Using the query string `TITLE-ABS-KEY(reverse engineering) AND SUBJAREA(comp OR math)`, `TITLE-ABS-KEY(embedded systems) AND SUBJAREA(comp OR math)` and `TITLE-ABS-KEY(reverse engineering embedded systems) AND SUBJAREA(comp OR math)`. The search string is applied to title, abstract and keywords.

[6] We used FacetedDBLP (`http://dblp.l3s.de`), which is based on Michael Ley's DBLP, to obtain this data. We did match "embedded" and "real-time" in the title and keywords (where available) and manually verified the results.

(JSA); Languages, Compilers, and Tools for Embedded Systems (LCTES); ACM Transactions on Embedded Computing Systems (TECS); and International Conference / Workshop on Embedded Software (EMSOFT). These publications have a high number of articles with "embedded system(s)" in their metadata.[7] Manual inspection of these papers for matches of "reverse engineering" in their metadata did not yield a true hit.

In the following, we briefly survey reverse engineering research surrounding (complex) embedded systems. Publications can be roughly clustered into the following categories:

- summary/announcement of a research project:
  - Darwin (van de Laar et al., 2011; 2007)
  - PROGRESS (Kraft et al., 2011)
  - E-CARES (Marburger & Herzberg, 2001)
  - ARES (Obbink et al., 1998)
  - Bylands (Bull et al., 1995)
- an embedded system is used for
  - a comparison of (generic) reverse engineering tools and techniques (Bellay & Gall, 1997) (Quante & Begel, 2011)
  - an industrial experience report or case study involving reverse engineering for
    - design/architecture recovery (Kettu et al., 2008) (Eixelsberger et al., 1998) (Ornburn & Rugaber, 1992)
    - high-level language recovery (Ward, 2004) (Palsberg & Wallace, 2002)
    - dependency graphs (Yazdanshenas & Moonen, 2011)
    - idiom extraction (Bruntink, 2008; Bruntink et al., 2007)
- a (generic) reverse engineering method/process is applied to—or instantiated for—an embedded system as a case study (Arias et al., 2011) (Stoermer et al., 2003) (Riva, 2000; Riva et al., 2009) (Lewis & McConnell, 1996)
- a technique is proposed that is specifically targeted at—or "coincidentally" suitable for—(certain kinds of) embedded systems:
  - slicing (Kraft, 2010, chapters 5 and 6) (Russell & Jacome, 2009) (Sivagurunathan et al., 1997)
  - clustering (Choi & Jang, 2010) (Adnan et al., 2008)
  - object identification (Weidl & Gall, 1998)
  - architecture recovery (Marburger & Westfechtel, 2010) (Bellay & Gall, 1998) (Canfora et al., 1993)
  - execution views (Arias et al., 2008; 2009)
  - tracing (Kraft et al., 2010) (Marburger & Westfechtel, 2003) (Arts & Fredlund, 2002)
  - timing simulation models (Andersson et al., 2006) (Huselius et al., 2006) (Huselius & Andersson, 2005)
  - state machine reconstruction (Shahbaz & Eschbach, 2010) (Knor et al., 1998)

[7] According to FacetedDBLP, for EMSOFT 121 out of 345 articles (35%) match, and for TECS 125 out of 327 (38%) match. According to Scopus, for JSA 269 out of 1,002 (27%) and for LCTES 155 out of 230 (67%) match.

For the above list of publications we did not strive for completeness; they are rather meant to give a better understanding of the research landscape. The publications have been identified based on keyword searches of literature databases as described at the beginning of this section and then augmented with the authors' specialist knowledge.

In Section 5 we discuss selected research in more detail.

4. Timing analysis

A key concern for embedded systems is their timing behavior. In this section we describe static and dynamic timing analyses. We start with a summary of software development—i.e., forward engineering from this chapter's perspective—for real-time systems. For our discussion, forward engineering is relevant because software maintenance and evolution intertwine activities of forward and reverse engineering. From this perspective, forward engineering provides input for reverse engineering, which in turn produces input that helps to drive forward engineering.

Timing-related analyses during forward engineering are state-of-the-practice in industry. This is confirmed by a study, which found that "analysis of real-time properties such as response-times, jitter, and precedence relations, are commonly performed in development of the examined applications" (Hänninen et al., 2006). Forward engineering offers many methods, technique, and tools to specify and reason about timing properties. For example, there are dedicated methodologies for embedded systems to design, analyze, verify and synthesize systems (Åkerholm et al., 2007). These methodologies are often based on a component model (e.g., AUTOSAR, BlueArX, COMDES-II, Fractal, Koala, and ProCom) coupled with a modeling/specification language that allows to specify timing properties (Crnkovic et al., 2011). Some specification languages extend UML with a real-time profile (Gherbi & Khendek, 2006). The OMG has issued the UML Profile for Schedulability, Performance and Time (SPL) and the UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE).

In principle, reverse engineering approaches can target forward engineering's models. For example, synthesis of worst-case execution times could be used to populate properties in a component model, and synthesis of models based on timed automata could target a suitable UML Profile. In the following we discuss three approaches that enable the synthesis of timing information from code. We then compare the approaches and their applicability for complex embedded systems.

4.1 Execution time analysis

When modeling a real-time system for analysis of timing related properties, the model needs to contain execution time information, that is, the amount of CPU time needed by each task (when executing undisturbed). To verify safe execution for a system the *worst-case execution time* (WCET) for each task is desired. In practice, timing analysis strives to establish a tight upper bound of the WCET (Lv et al., 2009; Wilhelm et al., 2008).[8] The results of the WCET Tool Challenge (executed in 2006, 2008 and 2011) provide a good starting point for understanding the capabilites of industrial and academic tools (`www.mrtc.mdh.se/projects/WCC/`).

[8] For a non-trivial program and execution environment the true WCET is often unknown.

Static WCET analysis tools analyze the system's source or binary code, establishing timing properties with the help of a hardware model. The accuracy of the analysis greatly depends on the accuracy of the underlying hardware model. Since the hardware model cannot precisely model the real hardware, the analysis has to make conservative, worst case assumptions in order to report a save WCET estimate. Generally, the more complex the hardware, the less precise the analysis and the looser the upper bound. Consequently, on complex hardware architectures with cache memory, pipelines, branch prediction tables and out-of-order execution, tight WCET estimation is difficult or infeasible. Loops (or back edges in the control flow graph) are a problem if the number of iterations cannot be established by static analysis. For such case, users can provide annotations or assertions to guide the analyses. Of course, to obtain valid results it is the user's responsibility to provide valid annotations. Examples of industrial tools are AbsInt's aiT (`www.absint.com/ait/`) and Tidorum's Bound-T (`www.bound-t.com`); SWEET (`www.mrtc.mdh.se/projects/wcet`) and OTAWA (`www.otawa.fr`) are academic tools.

There are also *hybrid approaches* that combine static analysis with run-time measures. The motivation of this approach is to avoid (or minimize) the modeling of the various hardware. Probabilistic WCET (or pWCET), combines program analysis with execution-time measurements of basic-blocks in the control flow graph (Bernat et al., 2002; 2003). The execution time data is used to construct a probabilistic WCET for each basic block, i.e., an execution time with a specified probability of not being exceeded. Static analysis combines the blocks' pWCETs, producing a total pWCET for the specified code. This approach is commercially available as RapiTime (`www.rapitasystems.com/products/RapiTime`). AbsInt's TimeWeaver (`www.absint.com/timeweaver/`) is another commercial tool that uses a hybrid approach.

A common method in industry is to obtain timing information by performing measurements of the real system as it is executed under realistic conditions. The major problem with this approach is the coverage; it is very hard to select test cases which generate high execution times and it is not possible to know if the worst case execution time (WCET) has been observed. Some companies try to compensate this to some extent through a "brute force" approach, where they systematically collect statistics from deployed systems, over long periods of real operation. This is however very dependent on how the system has been used and is still an "optimistic" approach, as the real WCET might be higher than the highest value observed.

Static and dynamic approaches have different trade-offs. Static approaches have, in principle, the benefit that results can be obtained without test harnesses and environment simulations. On the other hand, the dependence on a hardware timing model is a major criticism against the static approach, as it is an abstraction of the real hardware behavior and might not describe all effects of the real hardware. In practice, tools support a limited number of processors (and may have further restrictions on the compiler that is used to produce the binary to be analyzed). Bernat et al. (2003) argues that static WCET analysis for real complex software, executing on complex hardware, is "extremely difficult to perform and results in unacceptable levels of pessimism." Hybrid approaches are not restricted by the hardware's complexity, but run-time measurements may be also difficult and costly to obtain.

WCET is a prerequisite for *schedulability or feasibility analysis* (Abdelzaher et al., 2004; Audsley et al., 1995). (Schedulability is the ability of a system to meet all of its timing constraints.)

While these analyses have been successively extended to handle more complex (scheduling) behavior (e.g., semaphores, deadlines longer than the periods, and variations (jitter) in the task periodicity), they still use a rather simplistic system model and make assumptions which makes them inapplicable or highly pessimistic for embedded software systems which have not been designed with such analysis in mind. Complex industrial systems often violate the assumptions of schedulability analyses by having tasks which

- trigger other tasks in complex, often undocumented, chains of task activations depending on input
- share data with other tasks (e.g., through global variables or inter-process communication)
- have radically different behavior and execution time depending on shared data and input
- change priorities dynamically (e.g., as on-the-fly solution to identified timing problems during operation)
- have timing requirements expressed in functional behavior rather than explicit task deadline, such as availability of data in input buffers at task activation

As a result, schedulability analyses are overly pessimistic for complex embedded systems since they do not take behavioral dependencies between tasks into account. (For this reason, we do not discuss them in more detail in this chapter.) Analyzing complex embedded systems requires a more detailed system model which includes relevant behavior as well as resource usage of tasks. Two approaches are presented in the following where more detailed behavior models are used: model checking and discrete event simulation.

4.2 Timing analysis with model checking

Model checking is a method for verifying that a model meets formally specified requirements. By describing the behavior of a system in a model where all constructs have formally defined semantics, it is possible to automatically verify properties of the modeled system by using a model checking tool. The model is described in a modeling language, often a variant of finite-state automata. A system is typically modeled using a network of automata, where the automata are connected by synchronization channels. When the model checking tool is to analyze the model, it performs a *parallel composition*, resulting in a single, much larger automaton describing the complete system. The properties that are to be checked against the model are usually specified in a temporal logic (e.g., CTL (Clarke & Emerson, 1982) or LTL (Pnueli, 1977)). Temporal logics allow specification of safety properties (i.e., ”something (bad) will never happen”), and liveness properties (i.e., ”something (good) must eventually happen”).

Model checking is a general approach, as it can be applied to many domains such as hardware verification, communication protocols and embedded systems. It has been proposed as a method for software verification, including verification of timeliness properties for real-time systems. Model checking has been shown to be usable in industrial settings for finding subtle errors that are hard to find using other methods and, according to Katoen (1998), case studies have shown that the use of model checking does not delay the design process more than using simulation and testing.

SPIN (Holzmann, 2003; 1997) is a well established tool for model checking and simulation of software. According to SPIN’s website (`wwww.spinroot.com`), it is designed to scale well and can perform exhaustive verification of very large state-space models. SPIN’s modeling language, Promela, is a guarded command language with a C-like syntax. A Promela model roughly consists of a set of sequential processes, local and global variables and communication channels. Promela processes may communicate using communication channels. A channel is a fixed-size FIFO buffer. The size of the buffer may be zero; in such a case it is a synchronization operation, which blocks until the send and receive operations can occur simultaneously. If the buffer size is one or greater, the communication becomes asynchronous, as a send operation may occur even though the receiver is not ready to receive. Formulas in linear temporal logic (LTL) are used to specify properties that are then checked against Promela models.[9] LTL is classic propositional logic extended with temporal operators (Pnueli, 1977). For example, the LTL formula `[] (l U e)` uses the temporal operators always (`[]`) and strong until (`U`). The logical propositions l and e could be electrical signals, e.g., in a washing machine, where l is true if the door is locked, and e is true if the machine is empty of water, and thereby safe to open. The LTL formula in the above example then means "the door must never open while there is still water in the machine."

Model checkers such as SPIN do not have a notion of quantitative time and can therefore not analyze requirements on timeliness, e.g., "if x, then y must occur within 10 ms". There are however tools for model checking of real-time systems that rely on timed automata for modeling and Computation Tree Logic (CTL) (Clarke & Emerson, 1982) for checking.

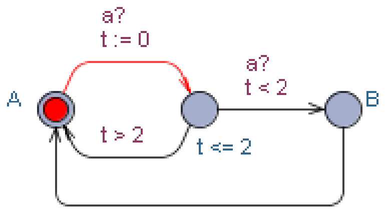

Fig. 2. Example of a timed automaton in UppAal.

A *timed automata* may contain an arbitrary number of clocks, which run at the same rate. (There are also extensions of timed automata where clocks can have different rates (Daws & Yovine, 1995).) The clocks may be reset to zero, independently of each other, and used in conditions on state transitions and state invariants. A simple yet illustrative example is presented in Figure 2, from the UppAal tool. The automaton changes state from A to B if event a occurs twice within 2 time units. There is a clock, t, which is reset after an initial occurrence of event a. If the clock reaches 2 time units before any additional event a arrives, the invariant on the middle state forces a state transition back to the initial state A.

CTL is a branching-time temporal logic, meaning that in each moment there may be several possible futures, in contrast to LTL. Therefore, CTL allows for expressing possibility properties such as "in the future, x may be true", which is not possible in LTL.[10] A CTL formula consists of a state formula and a path formula. The state formulae describe properties of individual states, whereas path formulae quantify over paths, i.e., potential executions of the model.

[9] Alternatively, one can insert "assert" commands in Promela models.

[10] On the other hand, CTL cannot express fairness properties, such as "if x is scheduled to run, it will eventually run". Neither of these logics fully includes the other, but there are extensions of CTL, such as CTL* (Emerson & Halpern, 1984), which subsume both LTL and CTL.

Both the UppAal and KRONOS model checkers are based on timed automata and CTL. **UppAal** (`www.uppaal.org` and `www.uppaal.com`) (David & Yi, 2000) is an integrated tool environment for the modeling, simulation and verification of real-time systems. UppAal is described as "appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables." In practice, typical application areas include real-time controllers and communication protocols where timing aspects are critical. UppAal extends timed automata with support for, e.g., automaton templates, bounded integer variables, arrays, and different variants of restricted synchronization channels and locations. The query language uses a simplified version of CTL, which allows for reachability properties, safety properties and liveness properties. Timeliness properties are expressed as conditions on clocks and state in the state formula part of the CTL formulae.

The **Kronos** tool[11] (`www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/`) (Bozga et al., 1998) has been developed with "the aim to verify complex real-time systems." It uses an extension of CTL, Timed Computation Tree Logic (TCTL) (Alur et al., 1993), allowing to express quantitative time for the purpose of specifying timeliness properties, i.e., liveness properties with a deadline.

For model checking of complex embedded systems, the *state-space explosion* problem is a limiting factor. This problem is caused by the effect that the number of possible states in the system easily becomes very large as it grows exponentially with the number of parallel processes. Model checking tools often need to search the state space exhaustively in order to verify or falsify the property to check. If the state space becomes too large, it is not possible to perform this search due to memory or run time constraints.

For complex embedded systems developed in a traditional code-oriented manner, no analyzable models are available and model checking therefore typically requires a significant modeling effort.[12] In the context of reverse engineering, the key challenge is the construction of an analysis model with sufficient detail to express the (timing) properties that are of interest to the reverse engineering effort. Such models can be only derived semi-automatically and may contain modeling errors. A practical hurdle is that different model checkers have different modeling languages with different expressiveness.

Modex/FeaVer/AX (Holzmann & Smith, 1999; 2001) is an example of a model extractor for the SPIN model checker. Modex takes C code and creates Promela models by processing all basic actions and conditions of the program with respect to a set of rules. A case study of Modex involving NASA legacy flight software is described by Glück & Holzmann (2002). Modex's approach effectively moves the effort from manual modeling to specifying patterns that match the C statements that should be included in the model (Promela allows for including C statements) and what to ignore. There are standard rules that can be used, but the user may add their own rules to improve the quality of the resulting model. However, as explained before, Promela is not a suitable target for real-time systems since it does not have a notion of quantitative time. Ulrich & Petrenko (2007) describe a method that synthesizes models from traces of a UMTS radio network. The traces are based on test case executions

[11] Kronos is not longer under active development.

[12] The model checking community tends to assume a model-driven development approach, where the model to analyze also is the system's specification, which is used to automatically generate the system's code (Liggesmeyer & Trapp, 2009).

and record the messages exchanged between network nodes. The desired properties are specified as UML2 diagrams. For model checking with SPIN, the traces are converted to Promela models and the UML2 diagrams are converted to Promela never-claims. Jensen (1998; 2001) proposed a solution for automatic generation of behavioral models from recordings of a real-time systems (i.e. model synthesis from traces). The resulting model is expressed as UppAal timed automata. The aim of the tool is verification of properties such as response time of an implemented system against implementation requirements. For the verification it is assumed that the requirements are available as UppAal timed automata which are then parallel composed with the synthesized model to allow model checking.

While model checking itself is now a mature technology, reverse engineering and checking of timing models for complex embedded system is still rather immature. Unless tools emerge that are industrial-strength and allow configurable model extraction, the modeling effort is too elaborate, error-prone and risky. After producing the model one may find that it cannot be analyzed with realistic memory and run time constraints. Lastly, the model must be kept in sync with the system's evolution.

4.3 Simulation-based timing analysis

Another method for analysis of response times of software systems, and for analysis of other timing-related properties, is the use of *discrete event simulation*,[13] or simulation for short. Simulation is the process of imitating key characteristics of a system or process. It can be performed on different levels of abstraction. At one end of the scale, simulators such as Wind River Simics (`www.windriver.com/products/simics/`) are found, which simulates software and hardware of a computer system in detail. Such simulators are used for low-level debugging or for hardware/software co-design when software is developed for hardware that does not physically exist yet. This type of simulation is considerably slower than normal execution, typically orders of magnitudes slower, but yields an exact analysis which takes every detail of the behavior and timing into account. At the other end of the scale we find scheduling simulators, who abstract from the actual behavior of the system and only analyzes the scheduling of the system's tasks, specified by key scheduling attributes and execution times. One example in this category is the approach by Samii et al. (2008). Such simulators are typically applicable for strictly periodic real-time systems only. Simulation for complex embedded systems can be found in the middle of this scale. In order to accurately simulate a complex embedded system, a suitable simulator must take relevant aspects of the task behavior into account such as aperiodic tasks, triggered by messages from other tasks or interrupts. Simulation models may contain non-deterministic or probabilistic selections, which enables to model task execution times as probability distributions.

Using simulation, rich modeling languages can be used to construct very realistic models. Often ordinary programming languages, such as C, are used in combination with a special simulation library. Indeed, the original system code can be treated as (initial) system model. However, the goal for a simulation models is to abstract from the original system. For example, atomic code blocks can be abstracted by replacing them with a "hold CPU" operation.

[13] Law & Kelton (1993) define discrete event simulation as "modeling of a system as it evolves over time by a representation in which the state variables change instantaneously at separate points in time." This definition naturally includes simulation of computer-based systems.

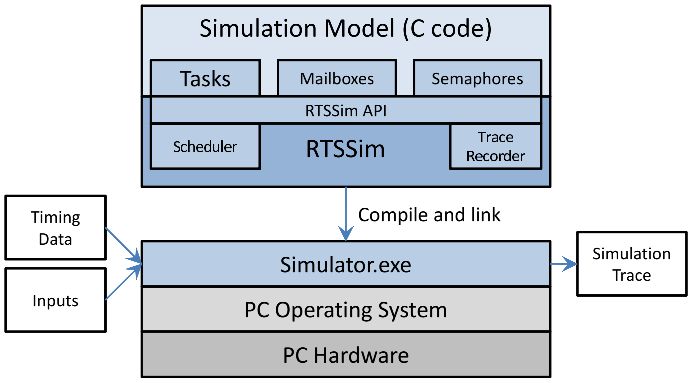

Fig. 3. Architecture of the RTSSim tool.

Examples of simulation tools are ARTISST (`www.irisa.fr/aces/software/artisst/`) (Decotigny & Puaut, 2002), DRTSS (Storch & Liu, 1996), RTSSim (Kraft, 2009), and VirtualTime (`www.rapitasystems.com/virtualtime`). Since these tools have similar capabilites we only describe RTSSim in more detail. **RTSSim** was developed for the purpose of simulation-based analysis of run-time properties related to timing, performance and resource usage, targeting complex embedded systems where such properties are otherwise hard to predict. RTSSim has been designed to provide a generic simulation environment which provides functionality similar to most real-time operating systems (cf. Figure 3). It offers support for tasks, mailboxes and semaphores. Tasks have attributes such as priority, periodicity, activation time and jitter, and are scheduled using preemptive fixed-priority scheduling. Task-switches can only occur within RTSSim API functions (e.g., during a "hold CPU"); other model code always executes in an atomic manner. The simulation can exhibit "stochastic" behavior via random variations in task release time specified by the jitter attribute, in the increment of the simulation clock, etc.

To obtain timing properties and traces, the simulation has to be driven by suitable input. A typical goal is to determine the highest observed response time for a certain task. Thus, the result of the simulation greatly depends on the chosen sets of input. Generally, a random search (traditional Monte Carlo simulation) is not suitable for worst-case timing analysis, since a random subset of the possible scenarios is a poor predictor for the worst-case execution time. *Simulation optimization* allows for efficient identification of extreme scenarios with respect to a specified measurable run-time property of the system. MABERA and HCRR are two heuristic search methods for RTSSim. MABERA (Kraft et al., 2008) is a genetic algorithm that treats RTSSim as a black-box function, which, given a set of simulation parameters, outputs the highest response-time found during the specified simulation. The genetic algorithm determines how the simulation parameters are changed for the next search iteration. HCRR (Bohlin et al., 2009), in contrast, uses a hill climbing algorithm. It is based on the idea of starting at a random point and then repeatedly taking small steps pointing "upwards", i.e., to nearby input combinations giving higher response times. Random restarts are used to avoid getting stuck in local maxima. In a study that involved a subset of an industrial complex embedded system, HCRR performed substantially better than both Monte Carlo simulation and the MABERA (Bohlin et al., 2009).

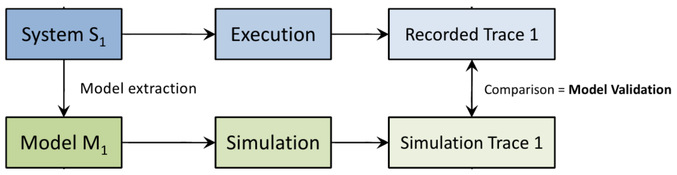

Fig. 4. Conceptual view of model validation with tracing data.

It is desirable to have a model that is substantially smaller than the real system. A smaller model can be more effectively simulated and reasoned about. It is also easier to evolve. Since the simulator can run on high-performance hardware and the model contains only the characteristics that are relevant for the properties in focus, a simulation run can be much faster than execution of the real system. Coupled with simulator optimization, significantly more (diverse) scenarios can be explored. A simulation model also allows to explore scenarios which are difficult to generate with the real system, and allows impact analyses of hypothetical system changes.

Reverse engineering is used to construct such simulation models (semi-)automatically. The **MASS** tool (Andersson et al., 2006) supports the semi-automatic extraction of simulation models from C code. Starting from the entry function of a task, the tool uses dependency analysis to guide the inclusion of relevant code for that task. For so-called model-relevant functions the tool generates a code skeleton by removing irrelevant statements. This skeleton is then interactively refined by the user. Program slicing is another approach to synthesize models. The Model eXtraction Tool for C (MTXC) (Kraft, 2010, chapter 6) takes as input a set of model focus functions of the real system and automatically produces a model via slicing. However, the tool is not able to produce an executable slice that can be directly used as simulation input. In a smaller case study involving a subset of an embedded system, a reduction of code from 3994 to 1967 (49%) lines was achieved. Huselius & Andersson (2005) describe a dynamic approach to obtain a simulation model based on tracing data containing interprocess communications from the real system. The raw tracing data is used to synthesize a probabilistic state-machine model in the ART-ML modeling language, which can be then run with a simulator.

For each model that abstracts from the real system, there is the concern whether the model's behavior is a reasonable approximation of the real behavior (Huselius et al., 2006). This concern is addressed by *model validation*, which can be defined as the "substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model" (Schlesinger et al., 1979). Software simulation models can be validated by comparing trace data of the real system versus the model (cf. Figure 4). There are many possible approaches, including statistical and subjective validation techniques (Balci, 1990). Kraft describes a five-step validation process that combines both subjective and statistical comparisons of tracing data (Kraft, 2010, chapter 8).

4.4 Comparison of approaches

In the following we summarize and compare the different approaches to timing analysis with respect to three criteria: soundness, scalability and applicability to industrial complex embedded systems. An important concern is *soundness* (i.e., whether the obtained timing results are guaranteed to generalize to all system executions). Timing analysis via executing the actual system or a model thereof cannot give guarantees (i.e., the approach is unsound), but heuristics to effectively guide the runs can be used to improve the confidence into the obtained results.[14] A sound approach operates under the assumptions that the underlying model is valid. For WCET the tool is trusted to provide a valid hardware model; for model checking the timing automata are (semi-automatically) synthesized from the system and thus model validation is highly desirable.[15]

Since both model synthesis and validation involve manual effort, scalability to large systems is a major concern for both model checking and simulation. However, at this time simulation offers better tool support and less manual effort. Another scalability concern for model checking is the state-space explosion problem. One can argue that improvements in model checking techniques and faster hardware alleviate this concern, but this is at least partially countered by the increasing complexity of embedded systems. Simulation, in contrast, avoids the state-space explosion problem by sacrificing the guaranteed safety of the result. In a simulation, the state space of the model is sampled rather than searched exhaustively.

With respect to applicability, execution time analysis (both static and hybrid) are not suitable for complex embedded systems and it appears this will be the case for the foreseeable future. The static approach is restricted to smaller systems with simple hardware; the hybrid approach does overcome the problem to model the hardware, but is still prohibitive for systems with nontrivial scheduling regimes and data/control dependencies between tasks. Model checking is increasingly viable for model-driven approaches, but mature tool support is lacking to synthesize models from source code. Thus, model checking may be applicable in principle, but costs are significant and as a result a more favorable cost-to-benefit ratio likely can be obtained by redirection effort elsewhere. Simulation arguably is the most attractive approach for industry, but because it is unsound a key concern is quality assurance. Since industry is very familiar with another unsound technique, testing, expertise from testing can be relatively easily transferred to simulation. Also, synthesis of models seems feasible with reasonable effort even though mature tool support is still lacking.

5. Discussion

Based on the review of reverse engineering literature (cf. Section 3) and our own expertise in the domain of complex embedded system we try to establish the current state-of-the-art/practice and identify research challenges.

It appears that industry is starting to realize that approaches are needed that enable them to maintain and evolve their complex embedded "legacy" systems in a more effective and predictable manner. There is also the realization that reverse engineering techniques are one important enabling factor to reach this goal. An indication of this trend is the Darwin

[14] This is similar to the problems of general software testing; the method can only be used to show the presence of errors, not to prove the absence of errors. Nonetheless, a simulation-based analysis can identify extreme scenarios, e.g., very high response-times which may violate the system requirements, even though worst case scenarios are not identified.

[15] The simulation community has long recognized the need for model validation, while the model checking community has mostly neglected this issue.

project (van de Laar et al., 2011), which was supported by Philips and has developed reverse engineering tools and techniques for complex embedded systems using a Philips MRI scanner (8 million lines of code) as a real-world case study. Another example is the E-CARES project, which was conducted in cooperation with Ericsson Eurolab and looked at the AXE10 telecommunications system (approximately 10 millions of lines of PLEX code developed over about 40 years) (Marburger & Herzberg, 2001; Marburger & Westfechtel, 2010).

In the following we structure the discussion into static/dynamic fact extraction, followed by static and dynamic analyses.

5.1 Fact extraction

Obtaining facts from the source code or the running system is the first step for each reverse engineering effort. Extracting static facts from complex embedded systems is challenging because they often use C/C++, which is difficult to parse and analyze. While C is already challenging to parse (e.g., due to the C preprocessor) , C++ poses additional hurdles (e.g., due to templates and namespaces). Edison Design Group (EDG) offers a full front-end for C/C++, which is very mature and able to handle a number of different standards and dialects. Maintaining such a front end is complex; according to EDG it has more than half a million lines of C code of which one-third are comments. EDG's front end is used by many compiler vendors and static analysis tools (e.g., Coverity, CodeSurfer, Axivion's Bauhaus Suite, and the ROSE compiler infrastructure). Coverity's developers believe that the EDG front-end "probably resides near the limit of what a profitable company can do in terms of front-end gyrations," but also that it "still regularly meets defeat when trying to parse real-world large code bases" (Bessey et al., 2010). Other languages that one can encounter in the embedded systems domain—ranging from assembly to PLEX and Erlang—all have their own idiosyncratic challenges. For instance, the Erlang language has many dynamic features that make it difficult to obtain precise and meaningful static information.

Extractors have to be robust and scalable. For C there are now a number of tools available with fact extractors that are suitable for complex embedded system. Examples of tools with fine-grained fact bases are Coverity, CodeSurfer, Columbus (`www.frontendart.com`), Bauhaus (`www.axivion.com/`), and the Clang Static Analyzer (`clang-analyzer.llvm.org/`); an example of a commercial tool with a course-grained fact base is Understand (`www.scitools.com`). For fine-grained extractors, scalability is still a concern for larger systems of more than half a million of lines of code; coarse-grained extractors can be quite fast while handling very large systems. For example, in a case study the Understand tool extracted facts from a system with more than one million of lines of C code in less than 2 minutes (Kraft, 2010, page 144). In another case study, it took CodeSurfer about 132 seconds to process about 100,000 lines of C code (Yazdanshenas & Moonen, 2011).

Fact extractors typically focus on a certain programming language per se, neglecting the (heterogeneous) environment that the code interacts with. Especially, fact extractors do not accommodate the underlying hardware (e.g., ports and interrupts), which is mapped to programming constructs or idioms in some form. Consequently, it is difficult or impossible for down-stream analyses to realize domain-specific analyses. In C code for embedded systems one can often find embedded assembly. Depending on the C dialect, different constructs are used.[16] Robust extractors can recognize embedded assembly, but analyzing it is beyond their capabilites (Balakrishnan & Reps, 2010).

Extracting facts from the running system has the advantage that generic monitoring functionality is typically provided by the hardware and the real-time operating system. However, obtaining finer-grained facts of the system's behavior is often prohibitive because of the monitoring overhead and the probing effect. The amount of tracing data is restricted by the hardware resources. For instance, for ABB robots around 10 seconds (100,000 events) of history are available, which are kept in a ring buffer (Kraft et al., 2010). For the Darwin project, Arias et al. (2011) say "we observed that practitioners developing large and complex software systems desire minimal changes in the source code [and] minimal overhead in the system response time." In the E-CARES project, tracing data could be collected within an emulator (using a virtual time mode); since tracing jobs have highest priority, in the real environment the system could experience timing problems (Marburger & Herzberg, 2001).

For finer-grained tracing data, strategic decisions on what information needs to be traced have to be made. Thus, data extraction and data use (analysis and visualization) have to be coordinated. Also, to obtain certain events the source code may have to be selectively instrumented in some form. As a result, tracing solutions cannot exclusively rely on generic approaches, but need to be tailored to fit a particular goal. The Darwin project proposes a tailorable architecture reconstruction approach based on logging and run-time information. The approach makes "opportunistic" use of existing logging information based on the assumption that "logging is a feature often implemented as part of large software systems to record and store information of their specific activities into dedicated files" (Arias et al., 2011).

After many years of research on scalable and robust static fact extractors, mature tools have finally emerged for C, but they are still challenged by the idiosyncracies of complex embedded systems. For C++ we are not aware of solutions that have reached a level of maturity that matches C, especially considering the latest iteration of the standard, C++11. Extraction of dynamic information is also more challenging for complex embedded systems compared to desktop applications, but they are attractive because for many systems they are relatively easy to realize while providing valuable information to better understand and evolve the system.

5.2 Static analyses

Industry is using static analysis tools for the evolution of embedded systems and there is a broad range of them. Examples of common static checks include stack space analysis, memory leakage, race conditions, and data/control coupling. Examples of tools are PC-lint (Gimpel Software), CodeSurfer, and Coverity Static Analysis. While these checkers are not strictly reverse engineering analyses, they can aid program understanding.

Static checkers for complex embedded systems face several adoption hurdles. Introducing them for an existing large system produces a huge amount of diagnostic messages, many of which are false positives. Processing these messages requires manual effort and is often prohibitively expensive. (For instance, Boogerd & Moonen (2009) report on a study where

[16] The developers of the Coverity tool say (Bessey et al., 2010): "Assembly is the most consistently troublesome construct. It's already non-portable, so compilers seem to almost deliberately use weird syntax, making it difficult to handle in a general way."

30% of the lines of code in an industrial system triggered non-conformance warnings with respect to MISRA C rules.) For complex embedded systems, analyses for concurrency bugs are most desirable. Unfortunately, Ornburn & Rugaber (1992) "have observed that because of the flexibility multiprocessing affords, there is an especially strong temptation to use ad hoc solutions to design problems when developing real-time systems." Analyses have a high rate of false positives and it is difficult to produce succinct diagnostic messages that can be easily confirmed or refuted by programmers. In fact, Coverity's developers says that "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).

Generally, compared to Java and C#, the features and complexity of C—and even more so of C++—make it very difficult or impossible to realize robust and precise static analyses that are applicable across all kinds of code bases. For example, analysis of pointer arithmetic in C/C++ is a prerequisite to obtain precise static information, but in practice pointer analysis is a difficult problem and consequently there are many approaches that exhibit different trade-offs depending on context-sensitivity, heap modeling, aggregate modeling, etc. (Hind, 2001). For C++ there are additional challenges such as dynamic dispatch and template metaprogramming. In summary, while these general approaches to static code analysis can be valuable, we believe that they should be augmented with more dedicated (reverse engineering) analyses that take into account specifically the target system's peculiarities (Kienle et al., 2011).

Architecture and design recovery is a promising reverse engineering approach for system understanding and evolution (Koschke, 2009; Pollet et al., 2007). While there are many tools and techniques very few are targeted at, or applied to, complex embedded systems. Choi & Jang (2010) describe a method to recursively synthesize components from embedded software. At the lowest level components have to be identified manually. The resulting component model can then be validated using model simulation or model checking techniques. Marburger & Westfechtel (2010) present a tool to analyze PLEX code, recovering architectural information. The static analysis identifies blocks and signaling between blocks, both being key concepts of PLEX. Based on this PLEX-specific model, a higher-level description is synthesized, which is described in the ROOM modeling language. The authors state that Ericssons' "experts were more interested in the coarse-grained structure of the system under study rather than in detailed code analysis." Research has identified the need to construct architectural viewpoints that address communication protocols and concurrency as well as timing properties such as deadlines and throughput of tasks (e.g., (Eixelsberger et al., 1998; Stoermer et al., 2003)), but concrete techniques to recover them are missing.

Static analyses are often geared towards a single programming language. However, complex embedded system can be heterogenous. The Philips MRI scanner uses many languages, among them C, C++/STL, C#, VisualBasic and Perl (Arias et al., 2011); the AXE10 system's PLEX code is augmented with C++ code (Marburger & Westfechtel, 2010); Kettu et al. (2008) talk about a complex embedded system that "is based on C/C++/Microsoft COM technology and has started to move towards C#/.NET technology, with still the major and core parts of the codebase remaining in old technologies." The reverse engineering community has neglected (in general) multi-language analyses, but they would be desirable—or are often necessary—for complex embedded systems (e.g., recovery of communication among tasks implemented in different languages). One approach to accommodate heterogenous systems with less tooling effort could be to focus on binaries and intermediate representations rather than source code (Kettu et al., 2008). This approach is most promising if source code is transformed to an underlying intermediate representation or virtual machine (e.g., Java bytecode or .NET CIL code) because in this case higher-level information is often preserved. In contrast, if source code is translated to machine-executable binaries, which is typically the case for C/C++, then most of the higher-level information is lost. For example, for C++ the binaries often do not allow to reconstruct all classes and their inheritance relationships (Fokin et al., 2010).

Many complex embedded systems have features of a product line (because the software supports a portfolio of different devices). Reverse engineering different configurations and variablity points would be highly desirable. A challenge is that often ad hoc techniques are used to realize product lines. For instance, Kettu et al. (2008) describe a C/C++ system that uses a number different techniques such as conditional compilation, different source files and linkages for different configurations, and scripting. Generally, there is research addressing product lines (e.g., (Alonso et al., 1998; Obbink et al., 1998; Stoermer et al., 2003)), but there are no mature techniques or tools of broader applicability.

5.3 Dynamic analyses

Research into dynamic analyses have increasingly received more attention in the reverse engineering community. There are also increasingly hybrid approaches that combine both static and dynamic techniques. Dynamic approaches typically provide information about a single execution of the system, but can also accumulate information of multiple runs.

Generally, since dynamic analyses naturally produce (time-stamped) event sequences, they are attractive for understanding of timing properties in complex embedded systems. The Tracealyzer is an example of a visualization tool for embedded systems focusing on high-level runtime behavior, such as scheduling, resource usage and operating system calls (Kraft et al., 2010). It displays task traces using a novel visualization technique that focuses on the task preemption nesting and only shows active tasks at a given point in time. The Tracealyzer is used systematically at ABB Robotics and its approach to visualization has proven useful for troubleshooting and performance analysis. The E-CARES project found that "structural [i.e., static] analysis ... is not sufficient to understand telecommunication systems" because they are highly dynamic, flexible and reactive (Marburger & Westfechtel, 2003). E-CARES uses tracing that is configurable and records events that relate to signals and assignments to selected state variables. Based on this information UML collaboration and sequence diagrams are constructed that can be shown and animated in a visualizer. The Darwin project relies on dynamic analyses and visualization for reverse engineering of MRI scanners. Customizable mapping rules are used to extract events from logging and run-time measurements to construct so-called execution viewpoints. For example, there are visualizations that show with different granularity the system's resource usage and start-up behavior in terms of execution times of various tasks or components in the system (Arias et al., 2009; 2011).

Cornelissen et al. (2009) provide a detailed review of existing research in dynamic analyses for program comprehension. They found that most research focuses on object-oriented software and that there is little research that targets distributed and multi-threaded applications. Refocusing research more towards these neglected areas would greatly benefit complex embedded systems. We also believe that research into hybrid analyses that augment static information with dynamic timing properties is needed.

Runtime verification and monitoring is a domain that to our knowledge has not been explored for complex embedded systems yet. While most work in this area addresses Java, Havelund (2008) presents the RMOR framework for monitoring of C systems. The idea of runtime verification is to specify dynamic system behavior in a modeling language, which can then be checked against the running system. (Thus, the approach is not sound because conformance is always established with respect to a single run.) In RMOR, expected behavior is described as state machines (which can express safety and liveness properties). RMOR then instruments the system and links it with the synthesized monitor. The development of RMOR has been driven in the context of NASA embedded systems, and two case studies are briefly presented, one of them showing "the need for augmenting RMOR with the ability to express time constraints."

6. Conclusion

This chapter has reviewed reverse engineering techniques and tools that are applicable for complex embedded systems. From a research perspective, it is unfortunate that the research communities of reverse engineering and embedded and real-time systems are practically disconnected. As we have argued before, embedded systems are an important target for reverse engineering, offering unique challenges compared to desktop and business applications.

Since industry is dealing with *complex* embedded systems, reverse engineering tools and techniques have to scale to larger code bases, handle the idiosyncracies of industrial code (e.g., C dialects with embedded assembly), and provide domain-specific solutions (e.g., synthesis of timing properties). For industrial practitioners, adoption of research techniques and tools has many hurdles because it is very difficult to assess the applicability and suitability of proposed techniques and the quality of existing tools. There are huge differences in quality of both commercial and research tools and different tools often fail in satisfying different industrial requirements so that no tool meets all of the minimum requirements. Previously, we have argued that the reverse engineering community should elevate adoptability of their tools as a key requirement for success (Kienle & Müller, 2010). However, this needs to go hand in hand with a change in research methodology towards more academic-industrial collaboration as well as a change in the academic rewards structure.

Just as in other domains, reverse engineering for complex embedded systems is facing adoption hurdles because tools have to show results in a short time-frame and have to integrate smoothly into the existing development process. Ebert & Salecker (2009) observe that for embedded systems "research today is fragmented and divided into technology, application, and process domains. It must provide a consistent, systems-driven framework for systematic modeling, analysis, development, test, and maintenance of embedded software in line with embedded systems engineering." Along with other software engineering areas, reverse engineering research should take up this challenge.

Reverse engineering may be able to profit from, and contribute to, research that recognizes the growing need to analyze systems with multi-threading and multi-core. Static analyses and model checking techniques for such systems may be applicable to complex embedded systems as well. Similarly, research in runtime-monitoring/verification and in the visualization of streaming applications may be applicable to certain kinds of complex embedded systems.

Lastly, reverse engineering for complex embedded systems is facing an expansion of system boundaries. For instance, medical equipment is no longer a stand-alone system, but a node in the hospital network, which in turn is connected to the Internet. Car navigation and driver assistance can be expected to be increasingly networked. Similar developments are underway for other application areas. Thus, research will have to broaden its view towards software-intensive systems and even towards systems of systems.

7. References

Abdelzaher, L. S. T., Arzen, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J. & Mok, A. K. (2004). Real time scheduling theory: A historical perspective, *Real-Time Systems* 28(2–3): 101–155.

Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C. & Latronico, E. (2010). *1st International Conference on Runtime Verification (RV 2010)*, Vol. 6418 of *Lecture Notes in Computer Science*, Springer-Verlag, chapter Automatic Requirements Extraction from Test Cases, pp. 1–15.

Adnan, R., Graaf, B., van Deursen, A. & Zonneveld, J. (2008). Using cluster analysis to improve the design of component interfaces, *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)* pp. 383–386.

Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P. & Tivoli, M. (2007). The SAVE approach to component-based development of vehicular systems, *Journal of Systems and Software* 80(5): 655–667.

Åkerholm, M., Land, R. & Strzyz, C. (2009). Can you afford not to certify your control system?, *iVTinternational* p. 16. http://www.ivtinternational.com/legislative_focus_nov.php.

Alonso, A., Garcia-Valls, M. & de la Puente, J. A. (1998). Assessment of timing properties of family products, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Vol. 1429 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 161–169.

Alur, R., Courcoubetis, C. & Dill, D. L. (1993). Model-checking in dense real-time, *Information and Computation* 104(1): 2–34. http://citeseer.ist.psu.edu/viewdoc/versions?doi=10.1.1.26.7610.

Andersson, J., Huselius, J., Norström, C. & Wall, A. (2006). Extracting simulation models from complex embedded real-time systems, *1st International Conference on Software Engineering Advances (ICSEA 2006)*.

Arias, T. B. C., Avgeriou, P. & America, P. (2008). Analyzing the actual execution of a large software-intensive system for determining dependencies, *15th IEEE Working Conference on Reverse Engineering (WCRE'08)* pp. 49–58.

Arias, T. B. C., Avgeriou, P. & America, P. (2009). Constructing a resource usage view of a large and complex software-intensive system, *16th IEEE Working Conference on Reverse Engineering (WCRE'09)* pp. 247–255.

Arias, T. B. C., Avgeriou, P., America, P., Blom, K. & Bachynskyyc, S. (2011). A top-down strategy to reverse architecting execution views for a large and complex software-intensive system: An experience report, *Science of Computer Programming* 76(12): 1098–1112.

Arts, T. & Fredlund, L.-A. (2002). Trace analysis of Erlang programs, *ACM SIGPLAN Erlang Workshop (ERLANG'02)*.

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W. & Wellings, A. J. (1995). Fixed priority pre-emptive scheduling: An historical perspective, *Real-Time Systems* 8(2–3): 173–198.

Avery, D. (2011). The evolution of flight management systems, *IEEE Software* 28(1): 11–13.

Balakrishnan, G. & Reps, T. (2010). WYSINWYX: What you see is not what you eXecute, *ACM Transactions on Programming Languages and Systems* 32(6): 23:1–23:84.

Balci, O. (1990). Guidelines for Successful Simulation Studies, *Proceedings of the 1990 Winter Simulation Conference*, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 2061-0106, USA.

Bellay, B. & Gall, H. (1997). A comparison of four reverse engineering tools, *4th IEEE Working Conference on Reverse Engineering (WCRE'97)* pp. 2–11.

Bellay, B. & Gall, H. (1998). Reverse engineering to recover and describe a system's architecture, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Vol. 1429 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 115–122.

Bernat, G., Colin, A. & Petters, S. (2002). WCET Analysis of Probabilistic Hard Real-Time Systems, *Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS'02), Austin, TX, USA*.

Bernat, G., Colin, A. & Petters, S. (2003). pWCET: a Tool for Probabilistic Worst Case Execution Time Analysis of Real-Time Systems, *Technical Report YCS353*, University of York, Department of Computer Science, United Kingdom.

Bessey, A., Block, K., Chelfs, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S. & Engler, D. (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.

Bohlin, M., Lu, Y., Kraft, J., Kreuger, P. & Nolte, T. (2009). Simulation-Based Timing Analysis of Complex Real-Time Systems, *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pp. 321–328.

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

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S. & Yovine, S. (1998). Kronos: A Model-Checking Tool for Real-Time Systems, *in* A. J. Hu & M. Y. Vardi (eds), *Proceedings of the 10th International Conference on Computer Aided Verification, Vancouver, Canada*, Vol. 1427, Springer-Verlag, pp. 546–550.

Broy, M. (2006). Challenges in automotive software engineering, *28th ACM/IEEE International Conference on Software Engineering (ICSE'06)* pp. 33–42.

Bruntink, M. (2008). Reengineering idiomatic exception handling in legacy C code, *12th IEEE European Conference on Software Maintenance and Reengineering (CSMR'08)* pp. 133–142.

Bruntink, M., van Deursen, A., D'Hondt, M. & Tourwe, T. (2007). Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations, *6th International Conference on Aspect-Oriented Software Development (AOSD'06)* pp. 199–211.

Bull, T. M., Younger, E. J., Bennett, K. H. & Luo, Z. (1995). Bylands: reverse engineering safety-critical systems, *International Conference on Software Maintenance (ICSM'95)* pp. 358–366.

Canfora, G., Cimitile, A. & De Carlini, U. (1993). A reverse engineering process for design level document production from ada code, *Information and Software Technology* 35(1): 23–34.

Canfora, G., Di Penta, M. & Cerulo, L. (2011). Achievements and challenges in software reverse engineering, *Communications of the ACM* 54(4): 142–151.

Choi, Y. & Jang, H. (2010). Reverse engineering abstract components for model-based development and verification of embedded software, *12th IEEE International Symposium on High-Assurance Systems Engineering (HASE'10)* pp. 122–131.

Clarke, E. M. & Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic, *Logic of Programs, Workshop*, Springer-Verlag, London, UK, pp. 52–71.

Confora, G. & Di Penta, M. (2007). New frontiers of reverse engineering, *Future of Software Engineering (FOSE'07)* pp. 326–341.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L. & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis, *IEEE Transactions on Software Engineering* 35(5): 684–702.

Crnkovic, I., Sentilles, S., Vulgarakis, A. & Chaudron, M. R. V. (2011). A classification framework for software component models, *IEEE Transactions on Software Engineering* 37(5): 593–615.

Cusumano, M. A. (2011). Reflections on the Toyota debacle, *Communications of the ACM* 54(1): 33–35.

David, A. & Yi, W. (2000). Modelling and analysis of a commercial field bus protocol, *Proceedings of 12th Euromicro Conference on Real-Time Systems*, IEEE Computer Society Press, pp. 165–172.

Daws, C. & Yovine, S. (1995). Two examples of verification of multirate timed automata with kronos, *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, IEEE Computer Society, Washington, DC, USA, p. 66.

Decotigny, D. & Puaut, I. (2002). ARTISST: An extensible and modular simulation tool for real-time systems, *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)* pp. 365–372.

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.

Eixelsberger, W., Kalan, M., Ogris, M., Beckman, H., Bellay, B. & Gall, H. (1998). Recovery of architectural structure: A case study, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Vol. 1429 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 89–96.

Emerson, E. A. & Halpern, J. Y. (1984). Sometimes and Not Never Revisited: on Branching Versus Linear Time, *Technical report*, University of Texas at Austin, Austin, TX, USA.

Fokin, A., Troshina, K. & Chernov, A. (2010). Reconstruction of class hierarchies for decompilation of C++ programs, *14th IEEE European Conference on Software Maintenance and Reengineering (CSMR'10)* pp. 240–243.

Gherbi, A. & Khendek, F. (2006). UML profiles for real-time systems and their applications, *Journal of Object Technology* 5(4). `http://www.jot.fm/issues/issue_2006_05/article5`.

Glück, P. R. & Holzmann, G. J. (2002). Using SPIN model checking for flight software verification, *IEEE Aerospace Conference (AERO'02)* pp. 1–105–1–113.

Graaf, B., Lormans, M. & Toetenel, H. (2003). Embedded software engineering: The state of the practice, *IEEE Software* 20(6): 61–69.

Hänninen, K., Mäki-Turja, J. & Nolin, M. (2006). Present and future requirements in developing industrial embedded real-time systems – interviews with designers in the vehicle domain, *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)* pp. 139–147.

Havelund, K. (2008). *Runtime Verification of C Programs*, Vol. 5047 of *Lecture Notes in Computer Science*, Springer-Verlag, chapter Testing of Software and Communicating Systems (TestCom/FATES'08), pp. 7–22.

Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet?, *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)* pp. 54–61.

Holzmann, G. (2003). *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley.

Holzmann, G. J. (1997). The Model Checker SPIN, *IEEE Trans. Softw. Eng.* 23(5): 279–295.

Holzmann, G. J. & Smith, M. H. (1999). A practical method for verifying event-driven software, *Proceedings of the 21st international conference on Software engineering (ICSE'99)*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 597–607.

Holzmann, G. J. & Smith, M. H. (2001). Software model checking: extracting verification models from source code, *Software Testing, Verification and Reliability* 11(2): 65–79.

Huselius, J. & Andersson, J. (2005). Model synthesis for real-time systems, *9th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pp. 52–60.

Huselius, J., Andersson, J., Hansson, H. & Punnekkat, S. (2006). Automatic generation and validation of models of legacy software, *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)*, pp. 342–349.

Jensen, P. K. (1998). Automated Modeling of Real-Time Implementation, *Technical Report BRICS RS-98-51*, University of Aalborg.

Jensen, P. K. (2001). *Reliable Real-Time Applications. And How to Use Tests to Model and Understand*, PhD thesis, Aalborg University.

Kaner, C. (1997). Software liability. `http://www.kaner.com/pdfs/theories.pdf`.

Katoen, J. (1998). Concepts, algorithms and tools for model checking, lecture notes of the course Mechanised Validation of Parallel Systems, Friedrich-Alexander University at Erlangen-Nurnberg.

Kettu, T., Kruse, E., Larsson, M. & Mustapic, G. (2008). *Architecting Dependable Systems V*, Vol. 5135 of *Lecture Notes in Computer Science*, Springer-Verlag, chapter Using Architecture Analysis to Evolve Complex Industrial Systems, pp. 326–341.

Kienle, H. M., Kraft, J. & Nolte, T. (2010). System-specific static code analyses for complex embedded systems, *4th International Workshop on Software Quality and Maintainability (SQM 2010), sattelite event of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*. `http://holgerkienle.wikispaces.com/file/view/KKN-SQM-10.pdf`.

Kienle, H. M., Kraft, J. & Nolte, T. (2011). System-specific static code analyses: A case study in the complex embedded systems domain, *Software Quality Journal*. Forthcoming, `http://dx.doi.org/10.1007/s11219-011-9138-7`.

Kienle, H. M. & Müller, H. A. (2010). The tools perspective on software reverse engineering: Requirements, construction and evaluation, *Advances in Computers* 79: 189–290.

Knor, R., Trausmuth, G. & Weidl, J. (1998). Reengineering C/C++ source code by transforming state machines, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Vol. 1429 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 97–105.

Koschke, R. (2009). Architecture reconstruction: Tutorial on reverse engineering to the architectural level, *in* A. De Lucia & F. Ferrucci (eds), *ISSSE 2006–2008*, Vol. 5413 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 140–173.

Kraft, J. (2009). RTSSim – a simulation framework for complex embedded systems, *Technical Report*, Mälardalen University. `http://www.mrtc.mdh.se/index.php?choice=publications&id=1629`.

Kraft, J. (2010). *Enabling Timing Analysis of Complex Embedded Systems*, PhD thesis no. 84, Mälardalen University, Sweden. `http://mdh.diva-portal.org/smash/get/diva2:312516/FULLTEXT01`.

Kraft, J., Kienle, H. M., Nolte, T., Crnkovic, I. & Hansson, H. (2011). Software maintenance research in the PROGRESS project for predictable embedded software systems, *15th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2011)* pp. 335–338.

Kraft, J., Lu, Y., Norström, C. & Wall, A. (2008). A Metaheuristic Approach for Best Effort Timing Analysis targeting Complex Legacy Real-Time Systems, *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*.

Kraft, J., Wall, A. & Kienle, H. (2010). *1st International Conference on Runtime Verification (RV 2010)*, Vol. 6418 of *Lecture Notes in Computer Science*, Springer-Verlag, chapter Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects, pp. 315–329.

Law, A. M. & Kelton, W. D. (1993). *Simulation, Modeling and Analysis*, ISBN: 0-07-116537-1, McGraw-Hill.

Lewis, B. & McConnell, D. J. (1996). Reengineering real-time embedded software onto a parallel processing platform, *3rd IEEE Working Conference on Reverse Engineering (WCRE'96)* pp. 11–19.

Liggesmeyer, P. & Trapp, M. (2009). Trends in embedded software engineering, *IEEE Software* 26(3): 19–25.

Lv, M., Guan, N., Zhang, Y., Deng, Q., Yu, G. & Zhang, J. (2009). A survey of WCET analysis of real-time operating systems, *2009 IEEE International Conference on Embedded Software and Systems* pp. 65–72.

Marburger, A. & Herzberg, D. (2001). E-CARES research project: Understanding complex legacy telecommunication systems, *5th IEEE European Conference on Software Maintenance and Reengineering (CSMR'01)* pp. 139–147.

Marburger, A. & Westfechtel, B. (2003). Tools for understanding the behavior of telecommunication systems, *25th Internatinal Conference on Software Engineering (ICSE'03)* pp. 430–441.

Marburger, A. & Westfechtel, B. (2010). Graph-based structural analysis for telecommunication systems, *Graph transformations and model-driven engineering*, Vol. 5765 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 363–392.

McDowell, C. E. & Helmbold, D. P. (1989). Debugging concurrent programs, *ACM Computing Surveys* 21(4): 593–622.

Müller, H. A. & Kienle, H. M. (2010). *Encyclopedia of Software Engineering*, Taylor & Francis, chapter Reverse Engineering, pp. 1016–1030. http://www.tandfonline.com/doi/abs/10.1081/E-ESE-120044308.

Müller, H., Jahnke, J., Smith, D., Storey, M., Tilley, S. & Wong, K. (2000). Reverse engineering: A roadmap, *Conference on The Future of Software Engineering* pp. 49–60.

Obbink, H., Clements, P. C. & van der Linden, F. (1998). Introduction, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Vol. 1429 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–3.

Ornburn, S. B. & Rugaber, S. (1992). Reverse engineering: resolving conflicts between expected and actual software designs, *8th IEEE International Conference on Software Maintenance (ICSM'92)* pp. 32–40.

Palsberg, J. & Wallace, M. (2002). Reverse engineering of real-time assembly code. http://www.cs.ucla.edu/~palsberg/draft/palsberg-wallace02.pdf.

Parkinson, P. J. (n.d.). The challenges and advances in COTS software for avionics systems. http://blogs.windriver.com/parkinson/files/IET_COTSaviation_PAUL_PARKINSON_paper.pdf.

Pnueli, A. (1977). The temporal logic of programs, *18th IEEE Annual IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pp. 46–57.

Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cimpan, S. & Verjus, H. (2007). Towards a process-oriented software architecture reconstruction taxonomy, *11th IEEE European Conference on Software Maintenance and Reengineering (CSMR'07)* pp. 137–148.

Quante, J. & Begel, A. (2011). ICPC 2011 industrial challenge. http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html.

Riva, C. (2000). Reverse architecting: an industrial experience report, *7th IEEE Working Conference on Reverse Engineering (WCRE'00)* pp. 42–50.

Riva, C., Selonen, P., Systä, T. & Xu, J. (2009). A profile-based approach for maintaining software architecture: an industrial experience report, *Journal of Software Maintenance and Evolution: Research and Practice* 23(1): 3–20.

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

Russell, J. T. & Jacome, M. F. (2009). Program slicing across the hardware-software boundary for embedded systems, *International Journal of Embedded Systems* 4(1): 66–82.

Samii, S., Rafiliu, S., Eles, P. & Peng, Z. (2008). A Simulation Methodology for Worst-Case Response Time Estimation of Distributed Real-Time Systems, *Proceedings of Design, Automation, and Test in Europe (DATE'08)*, pp. 556–561.

Schlesinger, S., Crosbie, R. E., Gagne, R. E., Innis, G. S., Lalwani, C. S. & Loch, J. (1979). Terminology for Model Credibility, *Simulation* 32(3): 103–104.

Shahbaz, M. & Eschbach, R. (2010). Reverse engineering ECUs of automotive components, *First International Workshop on Model Inference In Testing (MIIT'10)* pp. 21–22.

Sivagurunathan, Y., Harman, M. & Danicic, S. (1997). Slicing, I/O and the implicit state, *3rd International Workshop on Automatic Debugging (AADEBUG'97)* pp. 59–67. http://www.ep.liu.se/ea/cis/1997/009/06/.

Stoermer, C., O'Brien, L. & Verhoef, C. (2003). Moving towards quality attribute driven software architecture reconstruction, *10th IEEE Working Conference on Reverse Engineering (WCRE'03)* pp. 46–56.

Storch, M. & Liu, J.-S. (1996). DRTSS: A Simulation Framework for Complex Real-Time Systems, *2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pp. 160–169.

The Economist (2008). Driven to distraction: Why autonomous cars are still a pipe-dream. April 25, `http://www.economist.com/node/11113185`.

Tihinen, M. & Kuvaja, P. (2004). Embedded software development: State of the practice. `http://www.vtt.fi/moose/docs/oulu/embedded_sw_development_tihinen_kuvaja.pdf`.

Ulrich, A. & Petrenko, A. (2007). *3rd European conference on Model driven architecture-foundations and applications (ECMDA-FA'07)*, Vol. 4530 of *Lecture Notes in Computer Science*, Springer-Verlag, chapter Reverse Engineering Models from Traces to Validate Distributed Systems – An Industrial Case Study, pp. 184–193.

van de Laar, P., Douglas, A. U. & America, P. (2011). *Views on the Evolvability of Embedded Systems*, Springer-Verlag, chapter Researching Evolvability, pp. 1–20.

van de Laar, P., van Loo, S., Muller, G., Punter, T., Watts, D., America, P. & Rutgers, J. (2007). The Darwin project: Evolvability of software-intensive systems, *3rd IEEE Workshop on Software Evolvability (EVOL'07)* pp. 48–53.

van den Brand, M. G. J., Klint, P. & Verhoef, C. (1997). Reverse engineering and system renovation–an annotated bibliography, *SIGSOFT Software Engineering Notes* 22(1): 57–68.

Ward, M. P. (2004). Pigs from sausages? reengineering from assembler to C via FermaT transformations, *Science of Computer Programming* 52(1–3): 213–255.

Weidl, J. & Gall, H. (1998). Binding object models to source code: An approach to object-oriented re-architecting, *22nd IEEE International Computer Software and Applications Conference (COMPSAC'98)* pp. 26–31.

Weiser, M. (1981). Program Slicing, *5th International Conference on Software Engineering (ICSE'81)*, pp. 439–449.

Wilhelm, R., Engblom, J., Ermedahl, A., Holst, N., Thesing, S. et al. (2008). The worst-case execution-time problem—overview of methods and survey of tools, *Transactions on Embedded Computing Systems* 7(3): 36:1–36:50.

Yazdanshenas, A. R. & Moonen, L. (2011). Crossing the boundaries while analyzing heterogeneous component-based software systems, *27th IEEE International Conference on Software Maintenance (ICSM'11)* pp. 193–202.

Zhao, M., Childers, B. & Soffa, M. L. (2003). Predicting the impact of optimizations for embedded systems, *ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES'03)* pp. 1–11.