System-specific Static Code Analyses for Complex Embedded Systems

Holger M. Kienle, Johan Kraft, and Thomas Nolte
Mälardalen University
Västerås, Sweden
{holger.kienle,johan.kraft,thomas.nolte}@mdh.se

***Abstract*—In this paper we are exploring the approach to utilize system-specific static analyses of code with the goal to improve software quality for specific software systems, especially targeting complex embedded systems. Specialized analyses, tailored for a particular system, make it possible to take advantage of system/domain knowledge that is not available to more general analyses. Furthermore, analyses can be selected and/or developed in order to best meet the challenges and specific issues of the system at hand. Such analyses are likely to have a high impact on software quality attributes and can be used as a complement to generic code analysis tools. We present a case study of a large, industrial embedded system, giving examples of what kinds of analyses could be realized. System-specific analyses for complex embedded systems can target quality attributes that are especially relevant for this domain, such as safety and reliability.**

***Keywords*-software; quality; embedded; code-checking;**

I. Introduction

In this paper we relate our position on the quality assurance of industrial embedded software. In many respects the quality attributes of embedded systems are especially stringent and demanding. Examples of key quality attributes in this domain include real-time, reliability, safety, security, limited resources, and heterogeneity [1].

Embedded software systems are typically special-purpose systems developed for control of a physical process with the help of sensors and actuators. There is a wide variety of embedded systems, ranging from RFID tags over automotive components to the control of nuclear power plants. An embedded system is often a mechatronic system, which means that its construction requires a combination of mechanical, electronic, control, and computer engineering. In this paper we address *complex embedded systems*, which are large software systems, safety- and/or business-critical, and typically with a long maintenance history causing significant legacy issues (cf. Section II).

Embedded systems developers in industry have to juggle several challenges:

- Embedded systems are becoming more complex and feature-rich, and the growth rate of embedded software has accelerated as well [2]. Developers have to accommodate this trend without sacrificing key quality attributes.
- Correctness of an embedded real-time system also depends on *timeliness*, i.e., that the latency between input and output does 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 behaviour (i.e., probing effect [3]).
- Quality control has to be a concern during the entire life cycle and needs to be supported by the software process. Especially for safety there is increasing pressure from the legislature to follow (safety) standards and to document standards compliance [4].

Developers use techniques such as static analyses, dynamic analyses, metrics, and testing to monitor, assess, and increase the quality of their systems. These techniques are embedded within a (domain-specific) process that provides guidance on how and when these techniques should be applied.

For static analyses, developers can rely on a number of general-purpose tools, ranging from compiler warnings to sophisticated checkers such as PC-lint (Gimpel Software) and Coverity Static Analysis (formerly Coverity Prevent). Developers can also use coding standards. Examples of standards are MISRA C (2004) and MISRA C++ (2008) developed by the automotive industry. MISRA C consists of over one hundred coding rules; most of these can be checked by static code analysis [5]. While these general approaches to static code analysis can be valuable (e.g., to decrease faults [6]), they should be augmented with more dedicated analyses that take into account specifically the target system's peculiarities. We call these kinds of analyses *system-specific analyses*.

In this paper, we are arguing for system-specific analyses that are tailored to a specific system because such analyses can take advantage of domain/system knowledge that is not available to more general analyses. System-specific analyses have the potential to specifically target certain characteristics of the system that in turn have a strong impact on a set of quality attributes. As a baseline for discussion, we look closer at a specific complex embedded system developed by industry (cf. Section II-A), a control system for industrial robots. It is a large and complex software system that has evolved for almost 20 years to date and therefore has significant legacy issues. For instance, many of the original architects, designers and developers are no longer available at the company, so a lot of system knowledge has been lost,

and the amount of code has increased significantly over the years due to a steady feature growth.

Introducing static code analyses into the development practice of a large, mature system has to deal with several requirements. On the one hand, the analysis should be as unintrusive as possible (e.g., adding low overhead to the existing development process, and producing no spurious results); on the other hand the analysis should show immediate pay-offs (e.g, improving the software's quality and reducing faults).

The paper is organized as follows. Section II characterizes complex embedded systems and discusses an industrial example of such a system, namely the control system for an industrial robot developed at ABB Robotics. Section III discusses examples of system-specific analyses for the ABB system. We have started to implement these analyses with the help of Understand Analyst, a commercial, maintenance-oriented IDE that supports program understanding and reverse engineering activities. Understand's capabilites are described in Section IV. Section V briefly discusses related work, and Section VI concludes the paper.

II. Complex Embedded Systems

In our research we are mostly concerned with *complex embedded systems* [7]. These systems often contain millions of lines of code and are developed and maintained by dozens or hundreds of engineers over many years. These systems are mechatronic, having sensor and actuators to control functions in consumer cars, heavy industrial vehicles (e.g., for forestry and construction), industrial robots, etc. As a result, safety of complex embedded systems is a key quality attribute, because malfunction may result in death or serious injury of people. Furthermore, reliability and availability are important because these systems are often also business-critical.

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). In FPS scheduling, each task is has a scheduling priority, typically set by a system designer. This is typically not changed during runtime, although there are exceptions. An FPS scheduler always execute 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 becomes ready.

Online scheduling for real-time systems has the benefits that it is a flexible and efficient. There are also established methods for worst-case response time analysis [9]. Such methods are however often overly pessimistic for complex embedded systems, which typically has many intricate dependencies between the tasks. On the downside, for systems using FPS the details of the temporal behaviour (i.e., the exact execution order) is not known during design, but rather 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.

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.[2] As a general rule, the development practice for 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 such as nuclear reactor control. Even then, it is no panacea as formally proven software might still be unsafe [10].

In order to increase confidence in embedded systems, verification techniques such as reviews, analyses, and testing can be applied. 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" [2]. For embedded systems, Liggesmeyer and Trapp assess that "regarding quality assurance, dynamic testing is still a widespread approach for determining the correct functioning of software and systems" [10]. Processes and standards for safety-critical systems also mandate traceability of artifacts to facilitate verification activities.

A. Example: Industrial Robotics System

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

In essence, the robot controller has an object-oriented design, but is mainly implemented in C. It consists of approximately 2500 KLOC distributed in 400 to 500 so-called classes, which in turn are organized in a set of subsystems.

[1] A task is "the basic unit of work from the standpoint of a control program" [8]. It may be realized as an operating system process or thread.

[2] According to Ebert and Jones, C/C++ and assembly is used by more than 80 percent and 40 percent of companies, respectively [2].

The robot controller consists of three computers: the axis computer, a DSP which controls the motors of the robot, the I/O computer, and the main computer, which is the most complex part. The axis computer is truly safety-critical, but not very complex, while the main computer is more focused on performance; a deadline violation results in a failure (stop of the robot), but the axis computer keeps the system safe by stopping the motors and applying the brakes if control data is not received in time.

The main computer is a industry-PC solution using high-end Intel processors and the VxWorks[3] real-time operating system from Wind River Inc. (recently acquired by Intel). The software system in the main computer consists of more than 60 tasks, which are scheduled using preemptive fixed priority scheduling. The tasks communicate through message queues and shared data areas. Many tasks consist of several services, sometimes over 200, which are activated by messages from other tasks, or from a timer. A task typically spends most of the time blocked, waiting for incoming messages. When a message arrives, or a timer expires, the task executes the service corresponding to the event occurred. During the execution of a service, any incoming messages are buffered in the message queue for later processing.

Metric	Value
Total lines of code	1,082,934
Executable statements	448,853
Declarative statements	371,605
Functions	9,604
Source files	1,698

Table I
METRICS OF AVAILABLE SUBSYSTEM CODE

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

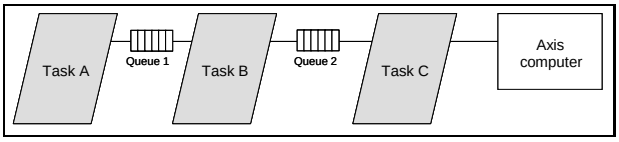

Figure 1. The Main Tasks of the Motion Control Subsystem

The axis computer sends requests to the main computer with a fixed, high rate, over 200 Hz. The axis computer expects a reply in the form of motor references within a certain time. This mainly depends on three large tasks in the motion control subsystem, as depicted in Figure 1. There are also other tasks in this subsystem, but these three are the largest, by far, as each of these consists of 1500-2500 functions and 100-200 KLOC each, even though some code is shared. We refer to these large tasks as A, B and C. The tasks B and C have high priority and run frequently with a fixed period. Task A executes mostly in the beginning of each robot movement and has lower priority, but produces data required by B. The B task processes the data and forwards the result in smaller parts to C, which makes the final processing and sends motor references to the axis computer. The data is sent through message queues. The interface of the motion control subsystem in A receives orders to move the robot and other commands from a client application (not shown in Figure 1), which decides how to move the robot.

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

III. SYSTEM-SPECIFIC STATIC ANALYSES

In this section we motivate the need for system-specific static analyses and give examples of such analyses that can be applied to ABB's system and perhaps to other similar systems. Based on previous work with this industrial partner, we believe that these analyses have the potential to improve upon the current development practice, resulting in an increase in the system's maintainability, robustness, and safety.

In contrast to system-specific static analyses, generic code checkers suffer much more from the problem that they may produce many false positives (i.e., warnings of non-conforming code that can be ignored). Boogerd and Moonen report on a study where 30% of the lines of code in their system triggered non-conformance warnings with respect to MISRA C rules [6]. Also, checkers that look for particular coding errors (e.g., buffer overruns or race conditions) can report 30% or more false positives [11]. The developers of the Coverity tool made the experience that more than 30% false positives reported by a tool causes problems in the sense that users ignore the tool or have little trust in it [12]. Code checking tools usually allow for suppressing warnings found to be irrelevant by annotating the corresponding code locations. However, since many developers work under significant time pressure and thereby are eager to solve any code checker warnings quickly, sometimes valid warnings are suppressed by mistake. Thus, increasing the credibility of code checking results, by increasing the accuracy, is therefore of major importance.

System-specific analyses can provide accurate analyses targeting particular concerns. If violations of a coding rule are observed, a dedicated analysis can be employed to identify all such non-conforming code in the system. This can be a valuable complement to using generic code checkers.

A. Task Interface

A simple example of a coding rule in the ABB system is the naming conventions used when implementing tasks.

The implementation of a task X is split into a set of public interface methods, corresponding to the available task services, and the task proper. The public interface methods are executed by other tasks and act as proxies, sending service requests to the task proper via a message queue by calling ipc_send(). The task proper handles these requests in its main loop with ipc_receive().

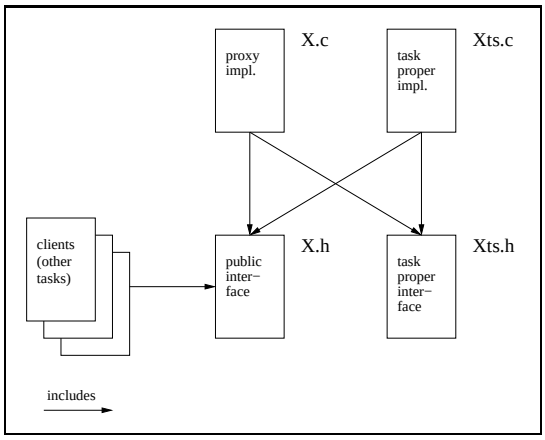

Figure 2. Task File Structure

The public interface methods are implemented as C functions in the files X.c and X.h. The task proper consists of the task's main loop and is implemented in the files Xts.c and Xts.h. The main loop of a task can be identified because it uses a standard coding pattern (see below). A system-specific analysis can check that the naming conventions and include relationships are not violated (cf. Figure 2). Importantly, X.h should not include Xts.h and vice versa. Also, Xts.h should no be included by clients of the task.

B. IPC Structure

An important constraint in the implementation of a task is that its public services (the interface methods) have to match the defined IPC message data-types and message codes. Public services are declared in X.h of the form $X\_s()$ where s is the name of a particular service (see top of Figure 3). For each service, there needs to be

- a corresponding enum member X_cmd_s in X_ipc_cmd that identifies the IPC message (i.e., service identifier),
- a struct typedef X_MSG_s that defines the message format, and
- a union member s of type X_MSG_s in a union that contains all task messages.

The above structures are defined in Xts.h and their constraints are illustrated in Figure 3. This coding pattern enables service s to send an IPC message (identified with X_cmd_s and having the message format X_MSG_s) to the task proper. A system-specific analysis can assure that for each service there is a unique, corresponding service

X.h

```
/* declaration of services */
...
int X_s(...);
...
```

Xts.h

```
/* service identifiers     */
enum X_ipc_cmd
{
  X_cmd_s,
  ...
}

/* service messages */
typedef struct {
  ...
} X_MSG_s;
...

typedef union {
  X_MSG_s s;
  ...
} X_MSG;
```

Figure 3. Task IPC Structure

identifier and message structure. This check is especially relevant in the following scenarios. When a new message is introduced, existing message structures should not be reused (even though the message may have the same fields).[4] More importantly, when a service is removed then all corresponding code should be eliminated as well in order to avoid dead code and deterioration of the code base. Having such a check is of practical importance for maintenance because the number of services can be large. For example, task C has more than 100 services.

Figure 4 gives a stylized example of a task main loop in the ABB system. When a message is received properly, it is passed to a message dispatcher function, which contains a (often large) "switch" statement, with one label (i.e., service identifier) for each service. An interesting analysis for this design pattern is to compare the labels of the switch statement (in the message dispatcher) with the declared service identifiers (in the X_ipc_cmd enum), in order to detect unused/obsolete message codes (i.e., for which no switch label exist). Any corresponding service function — which should be unused, dead code — can thereby be identified and removed. A similar analysis could also find service identifiers which are handled in the task message dispatcher,

[4]This is a form of defensive programming. If the message structure for one service is changed, this change will not affect other services if the message structure is not shared.

```
void Xts()
{
  int status, timeout;
  short cmd;
  X_MSG msg;

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

/* message dispatcher */
void ipc_disp (enum X_ipc_cmd cmd,
               X_MSG *msg )
{
  switch (cmd)
  {
    case X_s1:
      X_MSG_s1 *m= &msg->s1;
      ...
      break;
    case X_s2:
      ...
  }
}
```

Figure 4. Code Structure of a Task's Main Dispatch Loop (Xts.c)

but which have no corresponding service function, maybe due to an incomplete earlier removal of the service. The switch case for this message code can thereby be safely removed in order to make the message dispatcher's code shorter and more understandable.

C. Task Scheduling Priority

For systems using fixed-priority scheduling, response-time analysis typically requires that the task priorities are static, i.e., not changed during run-time, even though this is typically possible in most real-time operating systems. However, there are exceptions to this rule; in rare cases the priority of tasks is changed at run-time, in order to optimize the temporal behaviour (e.g., to prevent starvation due to empty message queues). For example, in the ABB system, task B boosts A's priority if a critical message queue drops below a certain length. In order to change priorities, tasks can call a wrapper function that abstracts away from the actual operating system call (VxWorks or Windows).

A simple system-specific analysis that looks for certain function calls can assure that code changes do not introduce new dynamic priority changes. Such a check is important because such changes may change the system's timing behaviour drastically. Thus, these changes should always be done in a controlled manner, e.g., in coordination with an in-depth impact analysis involving senior developers, followed by a thorough dynamic analysis of the new system behaviour.

D. IPC Message In/Out Parameters

As mentioned before, each service that a task offers is associated with an IPC message format defined in a struct (cf. Section III-B). The implementation of the public interface sets the relevant fields of the message which is then sent to the task proper.

If the sender expects a reply from the task proper a special IPC call is used that waits for a reply, ipc_sendwait(). For such cases, some of the message field are used as return values. Thus, from the perspective of the sender, fields have "in" or "out" parameter passing semantics. Naturally, the proxy implementation and the task proper have to agree on the fields' semantics. The importance of the in/out semantics is reflected in the code by providing comments next to most message fields (see Figure 5 for an example). These comments consistently read "In" or "Out" in C commenting style.

```
typedef struct
{
  IPC_HEADER header;
  param1 p1;
  param1 p2;  /* In */
  param1 p3;    /* Out  */
  ...
} X_MSG_s;
```

Figure 5. IPC Message In/Out Parameters (Xts.h)

A system-specific analysis can check that in-parameters are indeed set by the proxy and that out-parameters are indeed set by the task proper. Also, inconsistencies between the commenting and the actual implementation can be pointed out. Since parameters are manipulated within the same functions that perform the IPC calls an inter-procedural analysis is not needed. This nicely illustrates that certain properties of the system can reduce the complexity of system-specific analysis because simplifying assumptions can be made without decreasing the usefulness or the accuracy of the analysis.

E. Visibility Constraints

The system uses visibility constraints to describe the intended clients of functions. Functions are declared and defined by prepending PUBLIC, INTERNAL, and PRIVATE macros, where PRIVATE is translated to "static," which in C/C++ means "file internal." PUBLIC means that all code is allowed to call the function. In contrast, INTERNAL means that the function should be used only within the subsystem where it is defined.[5]

In the ABB system, all functions except PRIVATE are however globally visible, as there is only a single global namespace for non-private functions. The PUBLIC and INTERNAL labels have no effect on the actual visibility of the functions, but are rather a design-time annotation.

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

F. Task and Semaphore Dependencies

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

When tasks share data in this manner, they should use semaphores in order to protect critical sections where multiple related variables are updated in order to achieve thread-safety. An interesting analysis would be to check for functions which modify shared data without first acquiring the correct semaphore for a critical section. Such an analysis helps to identify transient faults, which otherwise are very hard to replicate and understand, as they depend on random execution-time variations which alter the relative timing between tasks.

For response time analysis, since a semaphore may block the execution of a task, it is necessary to know which tasks are using which semaphores. For large systems, this analysis would be very time consuming and error prone if performed manually. As an example, the more than 60 tasks in the ABB system use altogether close to 200 semaphores. Therefore, another interesting system-specific analysis would be to identify what semaphores are used by each task in order to facilitate response-time analysis. This can be further extended by also identifying the sections of code protected by semaphores (i.e., the program locations in which the semaphore's lock function is called, and the corresponding locations where the semaphore is released). This could allow for focused WCET-analysis[6] in order to estimate the worst case blocking time of the task, which is necessary input for response time analysis. Apart from response-time analysis, this analysis could also be used to check for missing semaphore release operations, at least obvious cases.

Another interesting analysis, related to semaphores but actually more general, is to check whether the return values of specific functions are actually handled by the caller. The semaphore lock operation is a typical example of this issue, as it is often performed with a time-out. If the time-out occurs, the wait is aborted and an error code returned to indicate that the semaphore has not been obtained. In the ABB system, this is treated as an error, resulting in a controlled stop of the robot in order meet safety requirements. However, if this check is forgotten when adding or modifying a critical section, a potential concurrency error is introduced which may lead to an erroneous, inconsistent system state, which in the worst case is detrimental to system's safety.

IV. Implementing the Analyses

We have started to implement system-specific analyses for the ABB system leveraging Understand Analyst 2.0 [13]. Understand is a GUI-based tool that can be used as an IDE for software development, but its primary strength are views for program understanding. For example, there is an Information Browser that gives detailed information about code entities. Understand can process several different languages, but we are using it for C/C++ only.

Understand builds a database model of the source code, describing what symbols (i.e., variables, functions, data types, etc.) exist and where they are used. Any preprocessor directives in the code are resolved with respect to specified preprocessor definitions and properly taken into account. Code that would be excluded by the preprocessing are included in the model, but marked as inactive.[7]

Understand exposes its underlying database of the code structure with an API [14]. The database represents the code as a typed graph structure consisting of *entities* (i.e., nodes) and *references* (i.e., arcs). The schema of the graph is essentially a *middle-level model* [15]. It has entity types such as File, Function, Object (i.e., for local and global variables), Macro and Typedef; and reference types such as Include, Declare, Define, Use, and Set. Understand resolves all symbols except calls via function pointers. Function pointer calls are detected, as well as their creation, but Understand does not have any support for the data-flow analysis required to connect these two.

[5]The system is organized into logical units called subsystems, where each subsystem consists of a set of classes (modules).

[6]Tools for Worst-Case Execution Time (WCET) analysis exists, e.g. AiT (AbsInt) and RapiTime (Rapita Systems). WCET analysis is however quite challenging and may, apart from automated static analysis, also require manual code annotations or runtime measurements.

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

Understand offers a C and Perl API; we are using Perl. The database entries can be directly queried with a lookup function based on name matching and entity type. For example, to get all header files contained in a database

```
$db->lookup("*.h", "File")
```

can be used. (In this example, `$db` is an object of the `Understand::Db` class that provides methods to access the database.)

Navigation of the graph structure from a certain source entity is accomplished by specifying the type of outgoing references and the type of the target entities. For example, given a file entity from the lookup operation in the previous example, then

```
$file_entity->refs("Define",
                   "Function")
```

gives a list of references that identify the functions defined in the specific header-file.[8]

We have chosen Understand for the following reasons. First, we have existing expertise in implementing sophisticated analyses with it. Second, the parser is robust, processing our system out-of-the-box. Third, the middle-level schema is easy to navigate and query, and sufficiently rich to accommodate many kinds of analyses. Fourth, the Perl API enables rapid prototyping of functionality, which is important in our setting because we expect to implement many different analyses that need to be iterative refined based on feedback of the industrial partner. On the downside, Understand's API makes it difficult to implement more sophisticated analyses that require control-flow or data-flow information. Also, Understand offers only very basic support to incorporate analyses into its GUI. Thus, visualization of results have to use external tools.

V. Related Work

There is a vast body of related work in the area of static code analyses and tools. In the following we briefly highlight some relevant work.

Static code analyses (e.g., static program slicing [17]) heavily draw from compiler design [18]. Such analyses are typically *sound* in the sense that they do not produce false positives and results are guaranteed to hold for all program runs [19]. There are dedicated analyses for real-time systems (e.g., worst case execution time analysis [20]) and concurrent/multithreaded systems (e.g., detection of race conditions and deadlocks [21]).

Static code checkers analyze programs for certain properties, typically with the idea to find faults (or design defects that may cause faults). For real-time systems there are checks that address memory, performance/timeliness, and security [2]. Examples of commercial tools are Fortify 360 (Fortify) and CodeSonar (Grammatech). For such a tool "the real measure of the effectiveness … is how well it simultaneously balances the false-positive rate with the false-negative rate" [22]. The same is true for analyses in the reverse engineering and program understanding domains, which are often unsound. The rationale for this is that even if the result of an analysis is unsound, it may give valuable information to a reverse engineer. Our system-specific analyses can be unsound, but since they are targeting a particular system the number of false positives should be small. In fact, one can argue that if a certain system-specific analysis cannot be realized with few false positives it has missed a key requirement and should not be deployed in the first place.

There are many approaches to building system-specific analyses on top of existing tools. The key questions are: how to obtain a suitable fact base of the source code, and how to write analyses using the fact base? C/C++ fact bases can be obtained from compiler front-ends (e.g., EDG) and reverse engineering tools developed by academia (e.g., CPPX [23], gccXfront [24], and XOgastan [25]) as well as industry (e.g., Bauhaus, SourceAudit, SolidFX, and Understand). Approaches exhibit different trade-offs in terms of robustness, scalability, completeness, and soundness. Our choice, Understand, offers a robust, scalable parser that produces a sound fact base (besides pointer analysis). The schema is situated at the middle-level and hence not complete; the low-level lexeme-based API is complete but awkward to query.

There are approaches that simplify the writing of analyses with dedicated querying and manipulation languages. Similar to Understand, many tools offer an API that provides access to the fact base with a high-level programming or scripting language. There are also approaches that allow a declarative specification of the query. The GUPRO tool parses source into an graph structure, which is then queried via a domain-specific graph query language called GReQL. GReQL queries looks similar to SQL and have predicates based on first order logic [26]. Semantic Grep is based on binary relations calculus and pattern matching [27]. It allows queries such as "*show all functions in parser.c*" or, more advanced, "*show all function calls from parser.c to scanner.c*." The tool is based on the established tools grok and grep, transforming its queries into commands for these tools. Revealer is a tool for architectural recovery, based on syntactic analysis [28]. It allows searching for complex patterns in source code — corresponding to "hotspots" — of a specific architectural view. For instance, the tool can be instructed to extract the relevant program statements of socket communication. The patterns are expressed in XML.

VI. Discussion, Future Work and Conclusions

We have discussed the idea of system-specific analyses and gave examples of such analyses for an industrial sample

[8] Understand also provides a low-level API that give access to the parser's lexeme stream, including comments. This API does not resolve symbols and matching of lexemes to entities have to be done with location information. The second author is developing a program slicer based on the low-level API [16].

system from ABB Robotics. It is a large and complex embedded system, which has to meet safety and dependability requirements.

The identified analyses for our sample system illustrate that system-specific analyses can potentially tackle a wide range of coding and design concerns. There are analyses that leverage existing (design) annotations in the form of comments or macro definitions; there are coding constraints that can be contained within a file or function, or that span multiple files; there are quite simple analyses (e.g., operating only on the names of function calls), and more sophisticated ones that require control-flow and/or data-flow information; and there are analyses that cover critical concerns of the domain such as scheduling and shared data for embedded systems.

In as sense, system-specific analyses are the most specific kind in the spectrum of analyses because they target one particular system. At the other end of the spectrum are language-agnostic analyses (e.g., simple metrics); their approach makes them general, but at the same time less targeted. Then there are static code checkers such as PC-lint that are language-specific but do not take the environment (i.e., operating system, libraries, or frameworks) into account. An example of domain-specific analyses are the ones targeting Web site reverse engineering [29]. An example of an environment-specific approach is a reverse engineering analysis by Pinzger et al. that targets systems based on COM/COM+ [30]. For instance, the analyses uses environment-specific knowledge such as framework calls and meta-data information stored in so-called type libraries. Similarly, static analyses for real-time and concurrent systems are domain-specific, targeting properties of the program that are especially relevant for their respective domains.

The goal of our system-specific analyses is to have a positive impact on key quality attributes of the system and to reduce defects in the code that can lead to faults. In order to be effective, system-specific analyses have to be part of the development process. For a large existing system this is difficult to accomplish. In order to get buy-in from the developers, system-specific analyses have to be low overhead (i.e., little or no false positives) and show almost immediate value. System-specific analyses need to be rapidly prototyped to test their effectiveness and to iteratively improve them. We believe that we can accomplish this by basing our analyses on Understand Analyst.

We are currently implementing the analyses for the industrial partner to demonstrate the feasibility of our approach. In the future, we envision to develop a specification tool that enables the system developers to write analyses themselves.[9] The specification tool should allow to specify the analyses at a higher-level of abstraction with the help of a (declarative) domain-specific language (DSL). For this approach to be affective, the specification needs to be rather concise and easy to learn. As a first step in this direction, we are planning to develop a higher-level API on top of the Understand API that provides an infrastructure that can be leveraged by several analyses. For example, the API can provide an abstraction for fix-point iteration of data-flow problems [18]. The second author is working on a static program slicer that we are hoping to integrate into the API [16]. Figure 6 depicts the envisioned infrastructure that we just discussed.

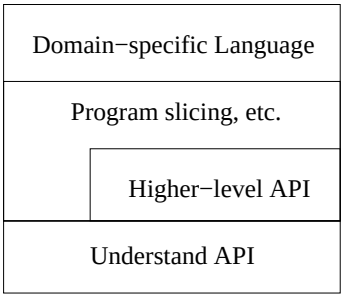

Figure 6. Infrastructure for System-specific Analyses

We are also hoping to deploy selected analyses at ABB Robotics and to study whether system-specific analyses indeed have a positive impact on the system's quality. Such a validation is important because the impact of code checks on fault detection can vary drastically as Boogerd and Moonen's study suggests [6].

[9] In practice, only a smaller subset of the developers that have been trained accordingly would write such analyses.

ACKNOWLEDGMENTS

This work is supported by the Swedish Foundation for Strategic Research (www.stratresearch.se), through the PROGRESS Centre for Predictable Embedded Software Systems, which is a part of Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University in Västerås, Sweden.

REFERENCES

[1] C. Ebert and J. Salecker, "Embedded software—technologies and trends," *IEEE Software*, vol. 26, no. 3, pp. 14–18, May/Jun. 2009.

[2] C. Ebert and C. Jones, "Embedded software: Facts, figures and future," *IEEE Computer*, vol. 42, no. 4, pp. 42–52, Apr. 2009.

[3] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Computing Surveys*, vol. 21, no. 4, p. 1989, Dec. 1989.

[4] M. Åkerholm, R. Land, and C. Strzyz, "Can you afford not to certify your control system?" *iVTinternational*, Nov. 2009, http://www.ivtinternational.com/legislative_focus_nov.php.

[5] C. Hills, "C-change for safety critical systems," *IEE Electronics Systems and Software*, vol. 3, no. 1, pp. 28–31, Feb./Mar. 2005.

[6] C. Boogerd and L. Moonen, "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, May 2009.

[7] J. Andersson, "Modeling the temporal behavior of complex embedded systems," Licentiate Thesis, Mälardalen University, Sweden, Jun. 2005.

[8] RTCA, "Software considerations in airborne systems and equipment certification," RTCA, Standard RTCA/DO-17B, Dec. 1992.

[9] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Systems*, vol. 8, no. 2–3, pp. 173–198, Mar. 1995.

[10] P. Liggesmeyer and M. Trapp, "Trends in embedded software engineering," *IEEE Software*, vol. 26, no. 3, pp. 19–25, May/ Jun. 2009.

[11] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pp. 83–93, Oct. 2004.

[12] A. Bessey, K. Block, B. Chelfs, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[13] SciTools, "Understand 2.0," http://www.scitools.com/ products/understand/.

[14] ——, *Perl and C Application Program Interface for Understand: User Guide and Reference Manual*, May 2004, http://getunderstand.com/documents/manuals/pdf/ understand_api.pdf.

[15] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder, "The Dagstuhl middle metamodel: A schema for reverse engineering," in *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, ser. Electronic Notes in Theoretical Computer Science, J.-M. Favre, M. Godfrey, and A. Winter, Eds. Elsevier, May 2004, vol. 94, pp. 7–18.

[16] J. Kraft, "Enabling timing analysis of complex embedded systems," Ph.D. dissertation, Mälardalen University, Sweden, 2010, forthcoming.

[17] M. Harman and R. M. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, Autumn 2001.

[18] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[19] D. Jackson and M. Rinard, "Software analysis: A roadmap," *Conference on The Future of Software Engineering*, pp. 135–145, Jun. 2000.

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

[21] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *19th ACM Symposium on Operating System Principles (SOSP'03)*, pp. 237–252, Oct. 2003.

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

[23] T. R. Dean, A. J. Malton, and R. Holt, "Union schemas as a basis for a C++ extractor," *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pp. 59–67, Oct. 2001.

[24] J. F. Power and B. A. Malloy, "Program annotation in XML: a parse-tree based approach," *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pp. 190–198, Nov. 2002.

[25] G. Antoniol, M. D. Penta, G. Masone, and U. Villano, "XOgastan: XML-oriented gcc AST analysis and transformations," *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pp. 173–182, Sep. 2003.

[26] B. Kullbach and A. Winter, "Querying as an enabling technology in software reengineering," *3rd IEEE European Conference on Software Maintenance and Reengineering (CSMR'99)*, pp. 42–50, Mar. 1999.

[27] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey, "Semantic grep: Regular expressions + relational abstraction," *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pp. 267–276, Oct. 2002.

[28] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri, "Revealer: A lexical pattern matcher for architecture recovery," *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pp. 170–178, Nov. 2002.

[29] H. M. Kienle and H. A. Müller, "Leveraging program analysis for web site reverse engineering," *3rd IEEE International Workshop on Web Site Evolution (WSE'01)*, pp. 117–125, Nov. 2001.

[30] M. Pinzger, J. Oberleitner, and H. Gall, "Analyzing and understanding architectural characteristics of COM+ components," *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 54–63, May 2003.