Migrating Legacy Control Software to Multi-core Hardware

Michael Wahler, Raphael Eidenbenz, Carsten Franke, Yvonne-Anne Pignolet
ABB Corporate Research
Baden-Daettwil, Switzerland
{firstname.lastname}@ch.abb.com

***Abstract*—This paper reports on a case study on analyzing, structuring and re-using real-time control algorithms which represent a significant amount of intellectual property. As a starting point, legacy code written in ADA together with a Windows-based testing framework is available. The goal is to migrate the code onto a real-time multi-core platform taking advantage of technological progress.**

We present a tool-supported three-step approach for such legacy control software: identifying and isolating the control algorithms, preparing these algorithms and their information exchange for execution within a modern execution framework for Linux written in C++, and validating the solution by a) performing regression testing to ensure partial correctness and b) validating its real-time properties.

I. Introduction

Legacy code is a valuable resource because it comprises knowledge and intellectual property (IP) which is often not available in other forms anymore. In domains such as industrial control and automation, this knowledge usually does not become obsolete and thus can be reused for several decades after it was written [1]. While algorithms and legacy code often do not change over time, the platforms on which the code can be executed steadily evolve and improve. Hardware and operating systems are enhanced continuously and new requirements may arise, e.g., connectivity to new kinds of devices and systems. To keep up with such changes and to exploit new opportunities, legacy software has to be maintained and adapted, e.g., to move to platforms with a better cost-benefit ratio or to profit from the progress of multi-core architectures properly [2].

Customers trust legacy code as it often has been in operation for many years. Hence the core algorithms of the code should not be changed during the evolution of legacy code. To this end the core algorithms first need to be identified and separated from platform-specific "glue code". This task is very challenging due to the fact that code typically grows over time. That is, functionality is not necessarily placed in one code unit, but it is distributed across the whole code base. Therefore, it is often difficult to understand which code fragments are used for a single functionality. To explore and understand a code base efficiently is hence a crucial pre-condition for re-use. To make things worse, specifications and documentation are often missing and the original programmers are often no longer available. Thus, (semi-)automatic systems and tools are of great value for the exploration and identification tasks.

In comparison to software in other domains, industrial control has to meet different requirements, e.g., regarding real-time properties, reliability, and constrained resources [3], [4]. General approaches for migration are thus often not directly applicable and have to be adapted to satisfy these requirements. In particular time-triggered behavior guarantees can be challenging and need special attention.

In this paper, we present an approach to analyzing, structuring and reusing legacy control software. It focuses on the identification and encapsulation of control algorithms, which represent core IP, and migrating these algorithms onto a next-generation execution framework. The contributions of this paper are

- tool-supported identification and isolation of the algorithms in the legacy code that comprise relevant IP;
- enhancement and environment development to execute the extracted legacy code on newer execution frameworks ensuring future code adaptation and more flexible program execution and maintenance. This includes moving existing data exchange mechanisms to more flexible concepts that can be maintained and replaced in the future; and
- validation of the proposed solution in two dimensions, functional correctness and timing behavior.

The chosen target execution framework is FASA (Future Automation System Architecture [5]), which offers real-time execution of control applications with advanced features such as multi-core support, dynamic software updates, distributed execution, and scalability from low-end to high-end platforms. Our approach preserves not only the know-how of many years of development, but also the corresponding testing and validation. We present a semi-automatic validation of the achieved solution by performing extensive functional testing and validating the code execution with respect to real-time properties.

A. Case Study

We will demonstrate the methods and tools on a case study based on an existing control application. This application consists of 342 kloc ADA code with a simulation and testing framework for Windows. This framework also provides us with validation criteria for our migration efforts in the form of integration test cases. The development of the code started in 1990 with a team of eight developers and has been actively maintained and extended ever since. The application has a large number of input and output streams, which need to be processed in real-time. Missing deadlines has critical consequences. To take advantage of modern multi-core architectures and to increase the maintainability this application is migrated to the real-time execution platform FASA written in C++ on RT-Linux (Linux with the Preempt-RT patch, which provides reliable preemption of kernel tasks based on thread priorities). At the same time no real-time guarantees must be violated and the functionality needs to stay exactly the same, without undesired side-effects introduced by the migration. Among the difficulties we face in this task, we first need to establish where the core control algorithms reside and to determine an efficient and safe transformation that allows these parts to expose their functionality to the FASA execution framework. In particular, dead-locks and race-conditions, which are often encountered when switching to a multi-threaded approach [6] are to be avoided. Due to the complex structure, the use of numerous global variables, and the lack of detailed documentation of the legacy code, it is not straightforward how to tackle this problem.

This paper is structured as follows. In Section II an overview of existing approaches is given, followed by a description of the methods applied to gain insights into the structure and functionality of the code base in Section III. Subsequently, Section IV presents how the legacy code is re-used and Section V explains the validation of the resulting artifacts. We summarize and discuss the methods and tools of this paper in Section VI.

II. RELATED WORK

The problem of re-using legacy code has been addressed from both the scientific and the practical perspective. This section provides an overview of existing methods and their differences to our approach.

Laguna and Crespo [7] performed a systematic literature-based study on different practice and research challenges for re-engineering of software product lines. The authors highlight tools and processes for feature extraction, feature analysis, feature-guided re-engineering, and meta model transformations. While we follow a similar general approach, none of the reviewed tools and use cases were applicable for our case study as ADA is not covered by them. In the last part of their article, Laguna and Crespo identify open issues and research challenges which include the analysis of the methodology and process of legacy code re-engineering as well as feature and model extraction from legacy code. We address these challenges by a migration case study for a real-time control application.

To support feature extraction and analysis, Anquetil and Laval use code metrics such as number of lines of code, cohesion, coupling and cyclomatic complexity to identify which code parts require attention for re-engineering and migration [8]. Anquetil and Laval analyze the relevance of existing metrics during re-engineering using the evolution of the Eclipse RPC platform as a test case. Their research concludes that metrics such as cohesion and coupling do not help the maintenance engineer. Similarly, the use of cyclic dependencies information did not improve the re-engineering results. A new metric called Bosch Maintainability Index (BMI) is introduced by Thums and Quante [4] to facilitate the task of understanding the code. The BMI uses questionnaire-based input as well as a combination of metrics like the ones described above for the first step of identifying relevant code segments, by describing the benefit that code maintenance would bring. This benefit is based on specific indicators. Another technique applied in their work focuses on code analysis to identify re-occuring segments and bad code smell patterns.

A different direction is proposed with a model-based validation method by Huselius et al. [9]. We cannot apply this approach as no model of the existing legacy software is available and the generation of a detailed model from scratch is too time consuming. However, future developments, especially for critical systems might benefit from model-based approaches.

Once the important parts of the legacy code have been identified and understood, there are different approaches to use them on new platforms. Petcu et al. [10] discuss a structured approach to determine which parts of the legacy software can be exposed as services and how to give access to these services. The authors distinguish three different approaches. The black box approach builds adapters to the existing legacy code. The white box approach analyzes the existing code and extracts the relevant parts to be re- used as services. The gray box approach combines wrapping and the white box approach and has been described in several case studies. E.g., by Colosimo et al. [11] and De Lucia et al. [12] on re-using legacy COBOL code with web-services or by Bernhart et al. [18], where an old airport operation system in COBOL is migrated using wrappers for intermediate systems running in parallel for maximum safety. Our case study also applies a gray box approach.

Razavian and Lago [13] state that industry practice often does not apply a black, white or gray box approach when moving to a service-oriented architecture. Instead the industry defines first the goal to be achieved and then follows a gradual migration path towards this goal. The authors define the sequence of work as understanding the current state, defining the goal, generating a gap analysis, architectural recovery, legacy system wrapping, and application wrapping. Additionally, Grimshaw et. al [14] address in general the wrapping problem of combining legacy code generated in various programming languages to form one system from the users perspective. They mention various problems for the resulting system without providing a specific methodology of the re-engineering task. De Schutter and Adams [15] address similar questions regarding aspect-oriented programming and logic meta-programming for legacy business code. To this end, four different use cases have been used, namely 1) reverse engineering, 2) recovery of business logic, 3) wrapping of business applications, and 4) maintenance of legacy code. Similarly to Grimshaw et al., De Schutter and Adams identified their solution as limited since legacy code often results from heterogeneous development (mixture of programming languages and environments).

Semi-automated code migration often uses abstract syntax trees [16], [17]. Chavaria-Miranda et al. [16] use such an approach for high performance computing applications written in C/Fortran/MPI. Kontogiannis et al. [17] concentrate on a comparison between manually migrated code and semi-automated code migration based on abstract syntax trees. The paper demonstrates the usefulness of abstract syntax trees and also highlights the difficulties to achieve efficient code which explains why this papers does not follow the described approach.

III. Understanding Legacy Code

Understanding the structure of the code base is an important prerequisite for effective code maintenance and migration. In particular, separating IP-relevant algorithms from platform-specific glue code should be done early to reduce the amount of code to be investigated in detail.

Legacy code often comes with little or no documentation of its architecture or design. In order to understand the structure of the code, developers must thus rely on the code itself. A useful starting point is to study how the code is organized in files and folders. Although this reveals partial information about the code structure, it hides many dependencies of the code and other parts of the system. In some cases, the names of files and folders are of limited help: in our case study, the code is organized in two folder levels with each folder having a three-letter name; in Fortran, folder names are limited to eight characters.

In particular, code written in languages that support static initializers, subtyping, or precompiled libraries can not be fully understood just by investigating the static call graph. As an example, reading the code starting from the *main* function does not reveal static initializations defined in other source files that could be executed before or during the *main* function.

Our solution to understanding the control flow of legacy software is twofold. First, we automatically extract the call graph while executing the software. Then, the extracted call graph is manually analyzed using a graphical representation.

A. Extracting the Call Graph

By extracting the call graph we want to find out which functions are called during normal execution and in which order. We do so by using a debugger. There are two kinds of debuggers: hardware debuggers and software debuggers. Hardware debuggers are typically used for embedded devices. They provide an active connection between the main board of the embedded device and the development PC. Through this connection, they can trace program execution down to individual CPU instructions. Software debuggers rely on symbols added to the executable file by the compiler. Using these symbols, software debuggers can step through programs by individual lines in the source code, view the values of variables, and manipulate control flow and data. Depending on the application and its current execution environment one or both options are viable. In our case study we use GDB (GNU Project debugger) for two reasons. First, because we compile the ADA sources with GCC which can create appropriate debug information on the binary for inspection by GDB. Another reason for using GDB is that our case study runs on a PC without any dedicated hardware debugging interface.

In order to extract the call graph during execution, two prerequisites are necessary. First, the source code must be compiled with flags that leave debugging information (such as variable names) in the compiled code and do not alter the original execution order. Second, it must be ensured that the execution covers a significant portion of the source code in order to get meaningful results. Code coverage is a research topic on its own in the context of software testing and thus, it is out of the scope of this paper. For obtaining insight into the control flow, it is usually sufficient to execute the code with one or several representative integration test case.

In our case study, we are interested in function-level granularity as we want to structure the source code on a coarse level only. To this end, it is sufficient to see which functions are only called during startup, which functions are regularly called during normal execution, and which functions are frequently called by a variety of other functions, thus indicating an auxiliary nature. Hence, we mainly use two features of GDB: a) setting a breakpoint at the entry point of each function and procedure that occurs in the source code and b) instructing GDB to print the function name, file name, and line number for each function (and, analogously, procedure). As mentioned earlier, the code to be executed must be compiled with special flags inhibiting optimization. For GDB, the GCC flags are `-O0` and `-ggdb`.

As a prerequisite to Step a), we use a shell script that locates the entry points of all functions and procedures in the source code and generates input commands for GDB. The whole script is shown in Appendix A. An example output of Step b) is shown in Listing 1.

```
#0 main ([...]) at main.adb:840
#0 adainit () at main.adb:463
#0 <dbs_file_name___elabb> () at bdy/
    dbs_file_name.adb:60
```

Listing 1. Example GDB output

It is a requirement for our approach that the program under analysis eventually terminates. Non-terminating programs can be terminated after a desired execution time or number of loop iterations.

B. Analyzing the Call Graph

The sequence of function invocations generated in the previous step can be analyzed in different ways. In our case study, we are interested in the call graph of our program. This representation allows us to visually identify clusters of related functions. For creating graphical representations, we use the

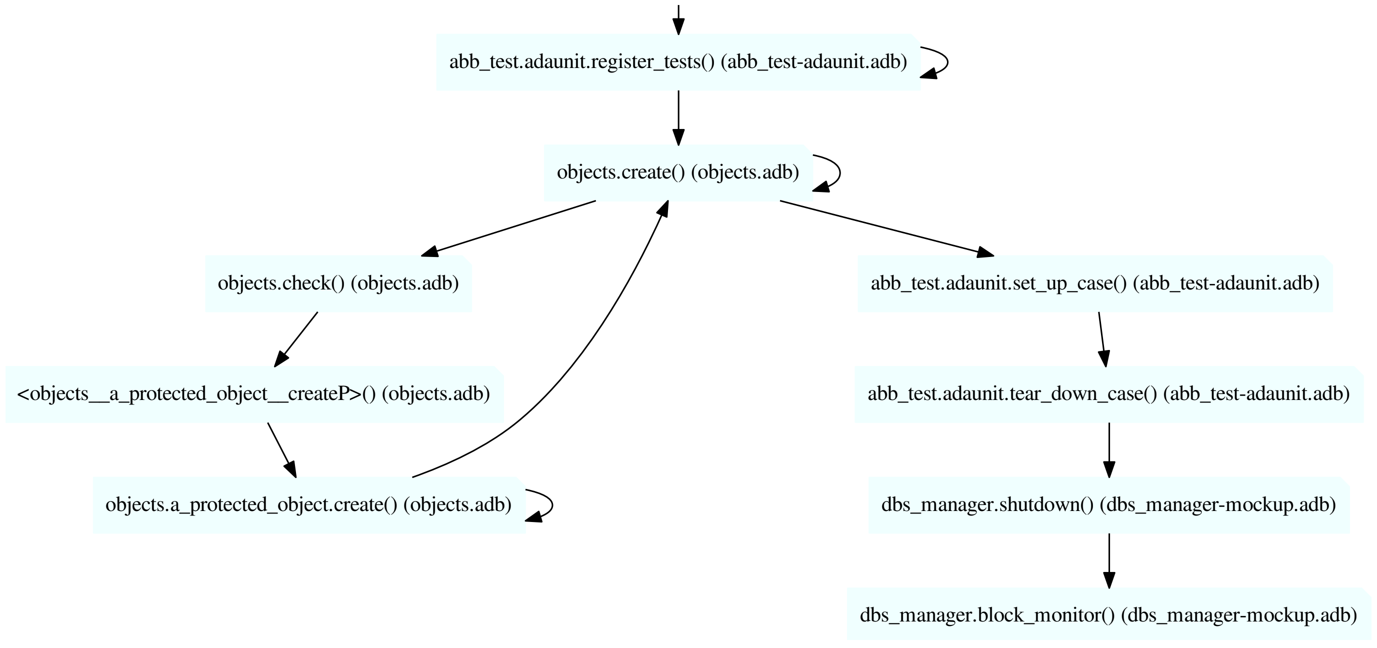
abb_test.adaunit.register_tests() (abb_test-adaunit.adb)
objects.create() (objects.adb)
objects.check() (objects.adb)
abb_test.adaunit.set_up_case() (abb_test-adaunit.adb)
<objects__a_protected_object__createP>() (objects.adb)
abb_test.adaunit.tear_down_case() (abb_test-adaunit.adb)
objects.a_protected_object.create() (objects.adb)
dbs_manager.shutdown() (dbs_manager-mockup.adb)
dbs_manager.block_monitor() (dbs_manager-mockup.adb)

Fig. 1. Example Call Graph

GRAPHVIZ tool, an open source graph visualization software that provides different layout options. Depending on the size of the project it can be useful to carry out additional processing steps before visualization. Graph analysis algorithms may be used to identify a hierarchical structure or closely coupled parts of the source code, which then can be visualized separately. In our case study this was not necessary.

Before invoking GRAPHVIZ, the GDB output that was previously generated must be converted into the input format of GRAPHVIZ, which defines the nodes and edges of the graph in text form. We perform this conversion in a shell script with frequent use of GAWK as shown in Listing 8.

There are several options for changing the layout of the generated graph. For control flow analysis, the option “strict digraph” is well suited because it combines multiple edges between the same pairs of nodes and formats the graph as a directed graph. An example call graph is shown in Figure 1. It can be seen that clusters of functions and files can be identified by analyzing the graph structure visually without further automation. After a linear sequence of startup activities (truncated from Figure 1), there are some independent sections in the bottom left and bottom right part of the figure. In our case study, analyzing the graph allowed us to separate code that belongs to the testing infrastructure (bottom right part) from code belonging to the actual algorithms (bottom left part).

IV. Re-using Legacy Code

Using the call graph analysis methods presented in the previous section, the algorithms that constitute the core IP of the legacy software can be identified. Then, these algorithms can be extracted and converted such that they can be used in a different runtime framework.

In our case study, the legacy code is written in ADA whereas the desired runtime framework is written in C++. Therefore, two options were considered:

1) Convert the selected ADA code into C++. Because of the large code size, this would need to be done automatically.
2) Compile the selected ADA code and provide wrappers for using it in a C++ project. The main disadvantage is that the number of ADA experts is likely to further decrease, which makes maintenance of the legacy algorithms more costly should the need arise in the future.

The second option was chosen for several reasons:

- The legacy implementation of the algorithms have been optimized, tested, debugged, and proven to work for many years.
- There is currently no need to change these algorithms.
- The code is platform independent.
- Automatic code conversion poses a risk that the generated code is not correct, maintainable, or meets its performance goals.

A. *C++ Wrappers for* ADA *Code*

As mentioned in Section I, we want to execute the legacy code on the FASA execution framework [5]. FASA provides a component-based architecture with a few simple concepts. As an example a *component* called Statistics, which is a container for blocks and state data, is depicted in Figure 2. *Blocks* (insert and average) are executable units that are sequential and must always terminate upon invocation, i.e., they must be non-blocking. Thus, developers must check the legacy code to be wrapped for the presence of concurrent or blocking code sections and fix them accordingly. Blocks communicate with each other through typed *ports* (in, out), which can be connected through *channels* (not shown in Figure 2) to form applications. This modular nature of components in FASA suggests a process for writing wrappers for the legacy code:

1) Combine any global variables used in the legacy code into a record (struct) and add this record as additional parameter to the corresponding legacy functions. The FASA components that wrap the legacy functions store this record as part of their state and pass it to the respective legacy functions. We assume here that the legacy code and its execution in FASA is sequential; concurrency would involve further investigations.
2) Compile the legacy code into a shared library, leaving out all unnecessary code, which has previously been identified as described in Section III.
3) For each function in the legacy code, create a new FASA wrapper block. The only action of the block is to read the input port(s), call the corresponding ADA function with the previously read inputs, and write the result of the ADA function to the output port(s). If the legacy function accesses global variables, create additional ports in order to pass this data to other blocks.
4) Aggregate blocks into components based on their coherence: closely related blocks should be in the same component, unrelated blocks should be in separate components.

In the legacy code, the names of the functions that should be migrated must be exported. In ADA this is done through an `Export` pragma in the header, as shown in Listing 2. The corresponding body is shown in Listing 3.

```
package Test is
  type Int32 is range -2**31..2**31-1;
  function f (x: Int32) return Int32;
  pragma Export (C, f, "_ada_f");
end Test;
```

Listing 2. Example ADA header

```
package body Test is
  function f (x: Int32) return Int32 is
  begin
    return x;
  end f;
end Test;
```

Listing 3. Example ADA body

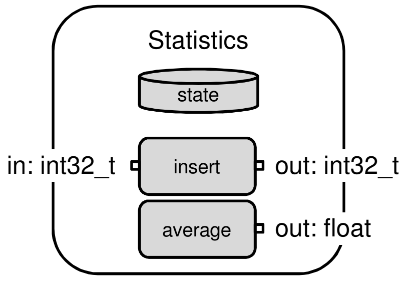

Fig. 2. Example FASA Component

The wrapper code in C++ remains minimal. As shown in Listing 4, the functions used from the legacy code must be declared using the keyword `extern` such that they can be called from C++ code.

```
extern "C"
{
  int32_t _ada_f (int32_t param1);
}

void QoS::operator() ()
{
  *out = _ada_f (*in);
}
```

Listing 4. Example C++ wrapper

In our case study, the biggest hurdle in the process was compiling and linking. It took a significant amount of effort to collect all required settings for the compiler and the linker such that we could call ADA functions from the C++ component code, which itself is compiled as a shared library in FASA. Some useful settings for the ADA project file (.gpr) are shown in Listing 5.

```
project My_Library is
  for Library_Name use "My_Library";
  for Library_Kind use "relocatable";
  for Library_Dir use "lib";
  for Library_Interface use ("my_main");
  for Library_Auto_Init use "true";
  for Object_Dir use ".tmp";

  package Compiler is
    for Driver ("Ada") use "g++";

    for Default_Switches ("ada") use
      ("-fstack-check", "-gnat12",
       "-Wall","-gnatwM", "-gnatwR",
       "-gnatQ", "-gnatp", "-gnatn",
       "-O2");
  end Compiler;

  package Binder is
   for Default_Switches ("ada") use
     ("-d32m", "-n");
  end Binder;
end My_Library;
```

Listing 5. ADA compiler/linker settings

When building the C++ code, the shared library containing the legacy code can then be linked (using the "-l" option for GCC). Also, the standard ADA library must be linked with the C++ code, as shown in Listing 6.

```
gcc component.cc -lgnat-2014
  -lMy_Library -L/usr/local/lib
```

Listing 6. GCC settings for linking a shared library in ADA

The result of our wrapping efforts is a component that executes legacy code on the FASA platform. This allows the algorithms in the legacy code to communicate transparently to other components, which can be executed on the same CPU core, on a different CPU core, or even on a different computer on the network [5]. Thus, we have moved existing data exchange mechanisms to a flexible concept that can be maintained and exchanged in the future.

B. Integration into the Execution Framework

Having wrapped the legacy code as components we can now integrate them into the execution framework FASA. As a prerequisite, an *application* must be created from the desired blocks. In FASA, an application is a directed graph whose nodes are blocks; this graph is implied by the data flow between the blocks and their ports.

FASA uses offline scheduling, i.e., the schedule is static and computed before the system goes into operation. The PASA tool (Partitioning And Scheduling Algorithms [19]) is used for calculating such static schedules. To this end, PASA maps an application to a set of execution resources, which can be locally distributed (i.e., different cores in one CPU) or physically distributed (i.e., nodes in a network), taking into account the respective communication latencies between blocks.

Figure 3 shows a cascaded PID control application as an example. It can be seen from the graph that the application contains concurrent execution branches. If two execution resources with low communication latency between them are available, our offline scheduler may compute the schedule shown in Figure 4. This schedule exploits the parallelism in the application, the sequential behavior of individual blocks (cf. Section IV-A), and caters for synchronization at the M1:Offset block (which needs the inputs of I1:Pressure) and at P2:PidCC (which needs the inputs of P1:PidCC, M1:Offset, and I2:Track).

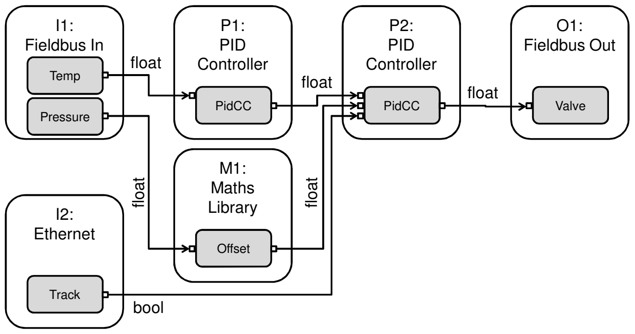

Fig. 3. Example Application

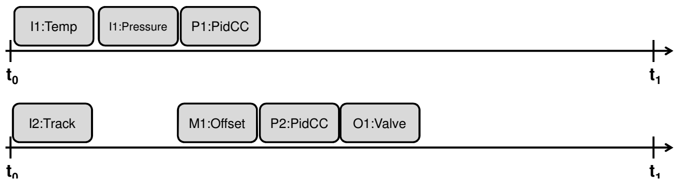

Fig. 4. Example Schedule for the Application in Figure 3

Using FASA, legacy applications can be transparently executed on either one or more execution resources. The runtime of applications can be decreased by exploiting concurrent branches in the control flow and consequently executing blocks on different execution resources in parallel. Alternatively, FASA allows developers to execute larger applications or multiple instances of the same application on a parallel system.

When scheduling legacy applications on FASA, it must be ensured that no race conditions or deadlocks can occur. Race conditions can be avoided by ensuring that blocks of the same component are not executed in parallel because they could share state data. Deadlocks are not possible in FASA, provided that the code in each block is non-blocking, which is usually the case for control algorithms.

V. VALIDATION

To show that the migration of the legacy code to the FASA platform according to our approach is successful, we validated the functional correctness of a reference implementation of our new solution as well as correct timely behavior, according to the application-specific real-time requirements.

A. Reference Implementation

We implemented a reference solution of the migrated application on an Aaeon industrial PC, which has an Intel i7 2.3 GHz Quad core processor and 2 GB RAM. The used real-time operating system is RT-Linux (kernel 3.4.12).

In contrast to our starting point, in which the input for the legacy ADA code was generated by a non-real-time simulation framework in software, the input in our target solution is provided as data streams sent from an external testing tool to the reference implementation via the network. Such a setup is necessary to test real-time properties of the whole system, including the processing of network traffic.

To this end, network traffic has to be received and delivered in real-time to the FASA application that wraps the original code. Since FASA does not provide a concept for handling event-driven tasks, such as processing network traffic, we use an approach that has been proposed by Eidenbenz et al. [20]. The approach relies on a non-FASA process that handles the network interrupts and communicates with the FASA application through inter-process communication, i.e., it puts the data into shared memory, where FASA blocks of a specific type, so-called *proxies*, fetch the data periodically and deliver it to the consuming FASA blocks. In our case, the proxies deliver the input data to the ADA code wrappers, which prepare the data and call the original functions with the data. Whereas the output of the legacy code was processed by the simulation and testing framework in the original solution, the output in the reference implementation was delivered to the network according to the delivery of input to the ADA code, namely via ADA wrappers, FASA proxies, and an external transmission process. Refer to the right-hand side of Figure 5 for an illustration of the reference implementation and testing architecture.

B. Functional Validation

For functional validation, we employed *regression testing*: we applied the same set of test cases that were used for functional validation in the original simulation framework to

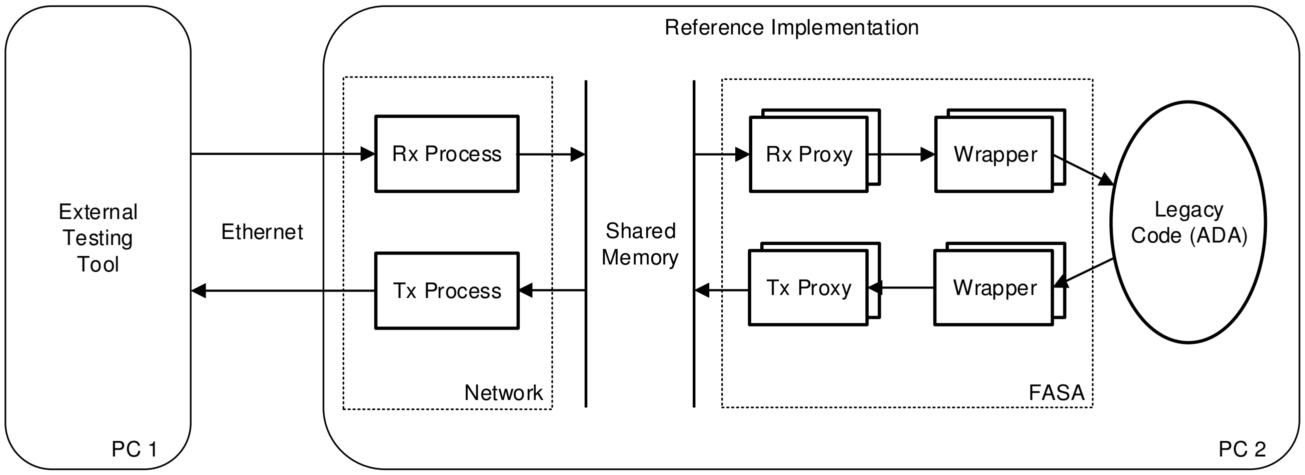

Fig. 5. Test setup for validation of reference implementation containing two RT-Linux PCs, one hosting the testing tool and one running the reference implementation. The arrows indicate the direction of the data flow.

the new solution. Doing so enabled us to compare the behavior of the new solution to the behavior of the original solution and to make sure there are no regressions, i.e., no new software bugs, introduced during the migration process. In our case, the simulation framework included the test-cases as *AUnit* tests[1]. Each of these tests defines input sequences to the ADA code and checks whether the monitored output corresponds to the expected values. In order to establish the same test cases on the new solution, it was necessary to migrate, i.e., re-implement these tests, so that they can be run from an external testing machine. As the input and output data of the new solution is mapped to Ethernet streams according to an industry standard, we could harness an existing testing tool for the purpose of the validation. The original test cases were thus migrated to this external testing tool. The testing tool generates Ethernet streams according to the test cases, transmits them across the network as input for the reference implementation; simultaneously, the tool monitors and verifies the output streams that are emitted by our device under test.

We executed several types of tests corresponding to relevant use cases of the controller. Each type comprises of testing the system with k input streams, producing n outputs, where n is a function of k that is determined by the tested use case. For a use case, we tested the behavior of the reference implementation with typical values of k and n as well as the highest values of k and n that the controller should be able to handle.

The results of the validation show that the migration to a FASA environment was successful: in all functional test cases, the monitored output of our new solution corresponded to the expected output. The new FASA version of our solution behaves like the legacy solution in all tested cases.

Note that full code coverage was not a focus of our migration effort, and we did not measure the code coverage achieved by our validation. In terms of the ADA code, however, we can expect a level of code coverage that is very close to the level achieved in the original validation of the legacy code due to the fact that we conducted the same tests, albeit migrated.

[1]http://libre.adacore.com/tools/aunit/

C. Real-time Properties

For validation of correct timing behavior, we added time measurements to the relevant test cases and measured the reaction times of the controller solution. The start time of each measurement was taken in the sender thread when issuing the call to send the packet to the network card; the end time was set to the time when the network card received the corresponding packet containing the expected signal. For this purpose, we adapted the testing tool and harnessed the hardware timestamping feature of the network card, which provides nanosecond accuracy.

For validating the real-time properties, we ran the tests again, augmented with time measurements. Each test case was executed several hundred times. The analysis of the time measurements showed that the reaction times of the new controller solution were within the ranges stated in the respective requirements specifications in all tested use cases and for all tested k and n.

VI. CONCLUSIONS

In this paper, we have described the techniques and methods applied for the migration of existing control applications to a new software and hardware platform. We have shown in our case study that it is possible to effectively migrate legacy code to a modern multi-core execution environment, even if the legacy code is 25 years old and the original deployment and runtime environment of the legacy program differs substantially from nowaday's environments. Clearly, not every legacy application and target environment calls for the same migration method; however, the strategic three-steps approach presented in this paper promises a successful migration in most cases. The three generic steps are:

1) identification and isolation of control algorithms with tools such as GDB and GRAPHVIZ;
2) wrapping or adapting of extracted code to embed it in a modern, future-proof deployment and runtime environment, such as FASA;
3) validation of new solution by re-using original, potentially migrated validation tests.

A. Lessons Learned

Our case study has provided us with some valuable insights. To start with, understanding how legacy code is structured and what the IP-relevant parts are requires substantial effort. Tool support is helpful because it can provide important clues, but it cannot replace manual code analysis in the general case. Developing the call graph extraction scripts and extracting the business logic consumed around 20 % of the project time.

Wrapping legacy code seems like a straight forward activity. The devil is, however, in the details. Building ADA code into a library that can be linked with C++ code was the most time-consuming activity in our case study (around 50 %). Little documentation for compiling and linking is available on the Internet in the form of Q&A discussions or tutorials for slightly related cases. The lack of comprehensive documentation motivated us to present detailed linker and compiler settings in this paper.

Integrating the wrapped code into the FASA runtime required around 10 % of the project time. The combination of GCC and GDB is very helpful for the integration of legacy code because these tools can be used for a variety of languages. In particular, GDB provides a uniform debugging interface to both the ADA and the C++ parts of the system.

The case study has been concluded by showing the feasibility of the presented approach to the stakeholders of our research project. They have provided us with positive feedback, in particular on the speed (ca. 8 person weeks of effort) and quality with which we managed to migrate the legacy software onto a modern platform. In particular, the possibility of utilizing multi-core CPUs and thus, to make the software scalable, was appreciated. The extensive validation, which consumed around 20 % of the project time, helped to convince the stakeholders of our solution.

B. Future Work

There are still many significant challenges in the area of migration of legacy code to other programming languages, operating systems, and hardware setups. In particular, more work is needed to make these processes easier and more cost efficient. To this end the identification of reachable code in legacy systems can benefit from systematic parametrization engines which can adapt to given scenarios and code segments. Additionally, a mechanism is required to generate representative test cases that help to derive the relevant code segments for translation or wrapping purposes. Better tool support for the approach discussed in this paper would include automatic generation of breakpoints for all functions and procedures and the fully automatic extraction of directed call graphs using the described underlying techniques.

Also the validation of migrated legacy code needs further research. Here, the generation of test cases (as for the identification of relevant code segments) need to be automated as well as the comparison of the results of the original and the new systems. Furthermore, code coverage for each test case should be investigated to assess the quality of the overall validation.

Beside the functional aspects, also non-functional aspects like run-time requirements need to be integrated into an automatic procedure. Thus, for the code analysis as well as for the validation work, precise and detailed time analysis is required to ensure the real-time behavior of the migrated solution.

REFERENCES

[1] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 36–47.

[2] A. Meade, J. Buckley, and J. Collins, "Challenges of evolving sequential to parallel code: an exploratory review," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. ACM, 2011, pp. 1–5.

[3] V. Schulte-Coerne, A. Thums, and J. Quante, "Automotive software: Characteristics and reengineering challenges," *Corporate Sector Research and Advance Engineering Software PO Box*, vol. 30, pp. 02–40, 2009.

[4] A. Thums and J. Quante, "Reengineering embedded automotive software," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 493–502.

[5] M. Wahler, T. Gamer, A. Kumar, and M. Oriol, "FASA: A Software Architecture and Runtime Framework for Flexible Distributed Automation Systems," *Journal of Systems Architecture*, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2015.01.002

[6] R. Craig and P. N. Leroux, "Case study-making a successful transition to multi-core processors," *QNX Software Systems GmbH & Co*, 2006.

[7] M. A. Laguna and Y. Crespo, "A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring," *Science of Computer Programming*, vol. 78, no. 8, pp. 1010–1034, 2013.

[8] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 279–286.

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

[10] D. Petcu, A. Eckstein, and C. Giurgiu, "Adapting a legacy code for ordinary differential equations to novel software and hardware architectures," *Transactions on Computers*, vol. 7, no. 5, pp. 463–472, 2008.

[11] M. Colosimo, A. De Lucia, G. Scanniello, and G. Tortora, "Evaluating legacy system migration technologies through empirical studies," *Information and Software Technology*, vol. 51, no. 2, pp. 433–447, 2009.

[12] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Developing legacy system migration methods and tools for technology transfer," *Software - Practice and Experience*, vol. 38, no. 13, pp. 1333–1364, 2008.

[13] M. Razavian and P. Lago, "A lean and mean strategy for migration to services," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*. ACM, 2012, pp. 61–68.

[14] A. S. Grimshaw, W. A. Wulf *et al.*, "The legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[15] K. De Schutter and B. Adams, "Aspect-orientation for revitalising legacy business software," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 63–80, 2007.

[16] D. Chavarría-Miranda, A. Panyala, W. Ma, A. Prantl, and S. Krishnamoorthy, “Global transformations for legacy parallel applications via structural analysis and rewriting,” *Parallel Computing*, vol. 43, March 2015.

[17] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos, “Code migration through transformations: An experience report,” in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 201–213.

[18] M. Bernhart, A. Mauczka, M. Fiedler, S. Strobl, and T. Grechenig, “Incremental reengineering and migration of a 40 year old airport operations system,” in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 503–510.

[19] A. Monot, A. Vulgarakis, and M. Benham, “PASA: Framework for Partitioning and Scheduling Automation Applications on Multicore Controllers,” in *Proceedings of the 19th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA 2014), Barcelona, Spain*, 2014.

[20] R. Eidenbenz, T. Sivanthi, A. Monot, and J. Liu, “Real-time Network Traffic Handling in FASA,” in *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems (SIES 2015), Siegen, Germany*, 2015.

APPENDIX
BASH SCRIPTS

These scripts show by an example how to extract the call graph from an ADA program and plot a visualization of it as explained in Section III.

A. Call Graph Extraction

```
#!/bin/bash
FASA_PATH=$(readlink -f ..)
WORKING_DIR=`pwd`

# preparing GDB commands file
echo "set␣breakpoint␣pending␣on" > gdb-
    commands.txt
echo "" >> gdb-commands.txt

# creating list of ada files; ignore file
    dbs_text_io and others because they are
    irrelevant
cd ..
find icssys -name *.adb | grep -v dbs_text_io
    | grep -v gnl_ | grep -v pdf_ >
    $WORKING_DIR/tmp-adafiles
find subsystem_test -name *.adb >>
    $WORKING_DIR/tmp-adafiles

# iterate through list of files
while read l; do
# find procedures, ignore lines starting with
    a comment

# file names with path
  grep -v '^--' $l | grep -n "procedure" | gawk
       -v filename=$l -v path=$FASA_PATH 'BEGIN
       { FS=":"} { print "b␣" path "/" filename
       ":" $1; print "commands"; print "silent"
      ; print "bt␣1"; print "c"; print "end";
      print ""; }' >> $WORKING_DIR/gdb-commands
      .txt

  #repeat line 24, replacing "procedure" with "
      function"
done < $WORKING_DIR/tmp-adafiles

cd $WORKING_DIR
rm tmp-adafiles

echo "run" >> gdb-commands.txt

gdb -batch -x gdb-commands.txt ../
    subsystem_test/obj/CU_native/Debug/ada/
    icstst_main_native > gdb-output.txt
```

Listing 7. Call Graph Extraction

B. Call Graph Visualization

```
#!/bin/bash
DOT_FILE=graph.dot
OUTPUT_FORMAT=pdf

# extract function names
cat gdb-output.txt \
| grep at | grep "#0" | gawk 'BEGIN { FS=":" }
     { print $1 }' | gawk 'BEGIN {FS="#0"} {
    print $2 }' | gawk 'BEGIN {FS="/"} {print
    $1 $NF }' | gawk '{print $1 "()␣(" $NF ")"
     } '> tmp.txt # | gawk '{print $2}'

# create dot file
echo "strict␣digraph␣ABBSS␣{" > $DOT_FILE
echo "graph␣[␣fontcolor=black,␣fontsize=14,␣
    ratio=0.7071␣];" >> $DOT_FILE
echo "node␣[color=azure,␣style=filled,␣shape=
    note];␣\"main() (b__cl_main_native.adb)\"␣
    [color="cornsilk"];" >> $DOT_FILE
cat tmp.txt | gawk -v q="\x22" '{ if (tmp!="")
     {print q $0 q ";";} printf q $0 q "->";
    if ((getline tmp) > 0) { print q tmp q ";\
    n" q tmp q "->"; } else { print "end"; }}'
     | head -n -1 >> $DOT_FILE
echo "}" >> $DOT_FILE

dot -T$OUTPUT_FORMAT $DOT_FILE >graph.
    $OUTPUT_FORMAT

rm $DOT_FILE
rm tmp.txt
```

Listing 8. Call Graph Visualization