A Systematic Migration Methodology for Complex Real-time Software Systems

Shaik Mohammed Salman*†, Alessandro Vittorio Papadopoulos†, Saad Mubeen†, and Thomas Nolte†
*ABB AB, Västerås, Sweden, shaik.salman@se.abb.com
†Mälardalen University, Västerås, Sweden, {name.surname}@mdh.se

***Abstract*—This paper provides a systematic three-stage methodology for migrating complex real-time industrial software systems from single-core to multi-core computing platforms. Single-core platforms have limited computational capabilities that prevent integration of computationally demanding applications such as image processing within the existing system. Modern multi-core processors provide increased computing capacity and allow the parallel execution of different applications within the system. However, this transition is non-trivial and requires a systematic and well-defined migration process. This paper reviews some of the existing migration methods and provides a systematic multi-phase migration process with emphasis on software architecture recovery and transformation to explicitly address the timing and dependability attributes expected of industrial software systems.**

***Index Terms*—real-time systems, multi-core, software architecture, software migration, robotics.**

I. Introduction

Software evolution has been a continuous process in industrial real-time embedded software systems with new functionality, performance improvements and bug fixes introduced with each new version [1], [2]. Many of these industrial systems have been developed over the decades [3], undergoing major revisions due to technology shifts, changing customer requirements, improved development processes, etc. The long life-cycle of such systems results in creation of large assets that become critical for a business [4]. However, one constant factor with evolution of such systems is that many of the software components and their architectures are optimized for single-core computing platforms. Integrating new data-intensive and computationally demanding applications such as the augmented reality interfaces [5], [6] requires additional computational capacity. Moreover, with the decreasing availability of single-core processors, there is a need to migrate the existing software to the powerful multi-core computing platforms. Such migration, however, should ensure that the quality attributes such as performance and dependability [7], [8] match the current system quality and more optimistically, improved much further.

A major concern while migrating existing real-time software systems towards multi-core systems is the identification of technical solutions that can improve the performance in terms of resource usage and timing predictability [9]–[11]. Invariably, these solutions should also be complementary to other extra-functional attributes such as scalability, maintainability and portability of the software system. Furthermore, the new solutions should ensure maximum reuse of the existing software with minimal re-engineering efforts.

To address these multi-dimensional aspects of a complex real-time software system with strict timing and dependability requirements, we use a focus group discussion to formulate an open-ended Research Question (RQ),

RQ: How to migrate a complex real-time software from a single-core to a multi-core architecture with maximum software reuse and minimal re-engineering effort?

We further refine this into the following sub-questions,

RQ1: Which migration methodology addresses the concerns of software reuse, dependability and timing requirements?

RQ2: How to evaluate and analyse the applicability of different multi-core solutions for embedded control software?

RQ3: What are the tools that facilitate the migration process?

These questions were developed by considering a configurable robot controller software [7] developed at ABB Robotics[1], with functionality ranging from motion control to cloud connectivity. The system has close to 140 tasks and 71,128 methods. It integrates real-time and non real-time functionalities with varying Quality of Service (QoS) requirements on a single-core platform.

To address the discussed questions, we use a mixed research methodology utilizing discussions with a focus group and subject experts complemented with a review of the state-of-the-art literature to identify key concerns and provide a systematic methodology to migrate industrial software with real-time requirements from single-core to multi-core platforms. Concretely, the paper provides the following contributions:

- a review of systematic approaches to software migration;
- a systematic methodology for migrating complex embedded software from single-core to multi-core platforms; and
- a review of tools that facilitate the migration process.

We would like to point out that the paper does not include an evaluation of the methodology as the migration is currently in-progress and will take considerable time before completion, hence the evaluation is left for future work. Also, since the focus of the paper is the methodology, we have limited discussion on possible multi-core solutions.

The rest of the paper is organized as follows. Section II provides an overview of a robotic system and its controller software. Section III reviews the software migration methods.

[1]https://new.abb.com/products/robotics/controllers

Section IV discusses a systematic approach focusing on architecture migration, followed by implementation and verification of the migration in Section V and Section VI respectively. A review of the tools facilitating the migration process is discussed in Section VII. Finally, Section VIII concludes the paper.

II. System Overview

The systems corresponds to a typical robotic system consisting of a manipulator arm, a controller, and a graphical controller interface. This paper considers software functionality of the controller, which can be divided into functions concerning (i) configuration, (ii) communication, and (iii) control. The configuration functions include the robot programming interface that allows the user to configure and specify the runtime behavior of the manipulator. The user is also able to define the robot environment such as additional sensors and actuators. The real-time communication functions allow the controller to interact with devices such as Programmable Logic Controllers (PLCs) and other sensors interconnected using a real-time network. The non-real-time communication functions allow the controller to interact with enterprise network including PCs and the cloud. The control functions generate the path to be followed by the manipulator based on the user-defined configuration. The output of the control functions is provided to drive controllers that manage the low-level motor actuation. The controller software has different runtime modes and the available functions vary between modes. The main modes include the "Initialization mode", "Safe-init mode", "System update and configuration mode", "Normal operation mode", and "Fail-safe mode" [12]. The controller software is in the initialization mode during startup. It enters the safe-init mode if there are errors during the startup. The controller software can be modified in the system update and configuration mode. It executes the motion planning algorithms with real-time communication enabled during the normal operation mode. It transitions into a fail-safe mode from a normal operation mode if an unexpected error occurs. During normal operation, the user-defined instructions from the robot programming interface provide input to the motion generation components of the software, which in turn generate the path to be followed by the manipulator. Simultaneously, the sensor information and actuator commands are read and written by the communication components based on the user configuration as well as system configuration.

III. Software Migration Methodology

Software migration is usually carried out when adopting a different architectural paradigm than the existing one, such as changing the programming language [13] or when moving from native server deployments to cloud-based deployments [14], [15]. Sneed [16] proposed a five-step re-engineering planning process for legacy systems, covering *Project Justification, Portfolio Analysis, Cost estimation, Cost-benefit analysis* and *Contracting*. The author highlights the need for creating measurable metrics to justify the effort and the improvements achievable with the migration. Erraguntla et al. [17] discussed a three phase migration method consisting of analysis, synthesis and transformation phases to migrate single-core to multi-core parallel environments. During the analysis and synthesis phase, the design of the existing software is recovered while recommendations for the multi-core environment are made during the transformation phase of the migration method. They also provided a reverse engineering toolkit called *RETK* for the analysis and synthesis phases. Battaglia [18] presented the *RENAISSANCE* method for re-engineering a legacy system. The method focuses on planning and management of the evolution process.

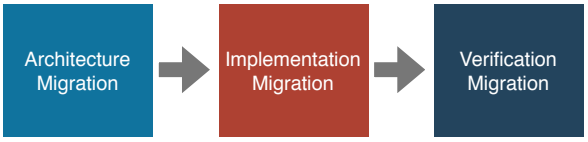

Fig. 1. Proposed migration workflow.

Menychtas et al. [19] presented a framework called *ARTIST*, a three-phase approach for software modernization focusing on migration towards the cloud. They categorized the migration into three main phases, *Pre-migration, Migration and Modernization* and *Post-migration*. During the pre-migration phase, they proposed a feasibility study to address the technical and economic points of view. During the migration and modernization phase, the actual migration is carried out and finally during the Post-migration phase, the system is deployed and validated. Forite et al. [20] proposed the *FASMM* approach to better manage the migration and to record and reuse the knowledge gained during the migration in other projects. More recently, Reussner et al. [2] and Wagner [21] proposed model-driven approaches to software migration. The focus in these approaches is to reverse engineer the system using automated tools and capture the information in modelling languages and then use the model-driven approach for further maintenance of the system.

Most of the works discussed so far focused on reverse engineering the existing system to get an understanding of the system, and then to use this information to model and transform the system based on the technical requirements. However, an important aspect we found lacking was emphasis on verification and validation of the reverse engineering processes. Additionally, while many of these works focused on architecture transformation and implementation changes, emphasis on migration of the testing methods was negligible. During our discussions in the focus group, testing was identified as an important domain which required investigation as multi-core architectures are more prone to concurrency issues, e.g., livelock, deadlock, race-conditions and data corruption.

Based on the reviewed methods and the extra-functional requirements, we create a migration workflow as depicted in Fig. 1 and apply the *A*nalyze, *V*erify, *T*ransform and *Va*lidate approach to this workflow. Essentially, during analysis, the requirements for the migration process are established and the

existing system behaviour is recovered. Then the results of the analysis are verified by the subject experts. New solutions are identified and evaluated during the transformation phase. Finally, the applicability of these solutions, along with the migration process, is validated during the validation phase. Additionally, we consider the migration process to be iterative in the sense that each stage can be revisited and decisions can be roll-backed or modified to address unexpected issues that may have been missed or if they do not meet the objective of the migration. A brief overview of the different stages of the proposed workflow is as follows.

1) During the first stage, we focus on the migration of software architecture. In this stage, the goal is to synthesize an abstract system model, validate its accuracy and transform the model for the multi-core environment.
2) In the second stage, the implementation and verification migration, the goal is to analyse the system source code to identify potential concurrency issues within the code and transform the code according to the new multi-core architecture model. Additionally. the existing verification techniques are augmented with methods relevant for a multi-core architecture.
3) In the third stage, we validate the migration process by identifying the validation parameters and measuring these parameters and then comparing them with the values obtained before migration.

IV. Software Architecture Migration

Real-time embedded systems have a strong focus on timing, safety and dependability requirements and therefore, should have a well-defined software architecture to support such requirements. As there are significant differences in the single-core and multi-core platforms, the software architecture should be modified such that it can make the best use of the resources and address implicit constraints. To approach this transformation systematically, the software architecture migration stage is divided into five well-defined phases shown in the Fig. 2.

1) architecture requirements specification;
2) architecture abstraction and representation;
3) architecture recovery;
4) architecture transformation; and
5) architecture verification.

A. Architecture Requirements Specification

As is common in any software development process, we consider the requirements specification as a necessary step in the migration process. In our case, the requirements are essentially high-level, focusing primarily on the extra-functional properties of scalability, performance and timing guarantees as the guiding principles for the complete migration process in itself and that the actual requirements are derived in the architecture recovery phase of the migration process. During this phase, we also identify a requirements specification and management process to better manage the requirements during the implementation stages of the migration process.

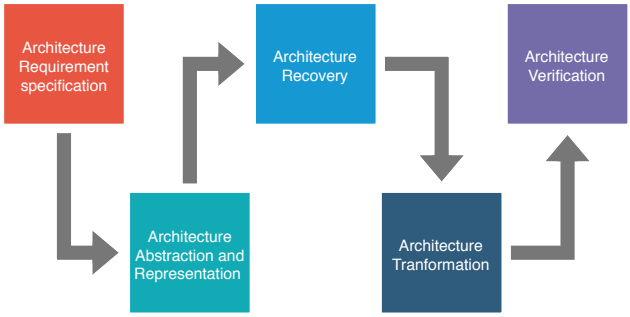

Fig. 2. Various phases in the software architecture migration.

B. Architecture Abstraction and Representation

In this phase, we seek to identify an abstraction level that accurately represents the system behaviour. An abstraction level close to the implementation may be too detailed, while a higher abstraction level can miss critical information that may be necessary for assuring correct system behaviour. Therefore, to identify the right abstraction, we need to identify the system properties that can be affected when moving to the multi-core architectures. A representation model that can sufficiently represent these properties is necessary. Such a representation should also be easy to comprehend for further analysis and for communication with stakeholders such as system architects and developers. To address these issues, we conduct interviews with experts and review state-of-the-art literature related to multi-core in the real-time systems domain and the model-driven engineering domain to guide the selection of the abstraction level and for the identification of the representation tools. We highlight the motivation for our choices and discuss the outcome in the respective sections.

1) Expert Interviews: Since the system we are considering is a complex system providing different functionalities ranging from embedded control to cloud connectivity, any ad-hoc or intuition-based selection of the abstraction level could result in potentially avoidable complexity issues and costly re-work. Therefore, we relied on informal and open-ended interviews with the system software architects and domain architects to identify possible abstraction levels. From these discussions, we identify the task-level abstraction to be sufficient for the migration process. We further identify dependencies, precedence constraints, timing properties, and inter-task communication as key parameters that can sufficiently capture the system's behaviour at the task level abstraction.

2) State-of-the-art in Real-time Systems: Since the literature on real-time systems is extensive, we skip further discussion but we conclude that much of the literature in this domain uses the task-level abstraction for the system representation [22] and provides results based on this abstractions [10], [23]. With this, we convincingly conclude that the task-level abstraction can be relied upon for the migration process.

3) State-of-the-art in Model-driven Engineering: Many modelling languages are available for representing the architecture of the system at various abstraction levels. Since

the system considered in this work is a real-time system, we identified the UML MARTE[2] profile [24], Rubus [25], [26], UPPAAL [27], MechatronicUML[3] [28], AUTOSAR [29], ART-ML Framework [30], just to name a few, as the possible modelling languages and frameworks that can be used to represent the system under discussion.

Although many of these languages, frameworks and supporting tools offer detailed semantics for capturing multiple viewpoints which is essential for managing embedded systems, the learning curve with many of these tools is rather steep, especially when being used for representing task-level abstraction of existing systems. We choose the UPPAAL tool for our purposes primarily for its ability to allow expressive modelling of the task-level abstraction. Once we have identified this, we move on to the architecture recovery phase of transformation.

C. Architecture Recovery

We need to have a better understanding of existing architecture to be able to modify and adapt it to new platforms. However, in many cases, the documented architecture or the intended architecture does not represent the actual implementation. Such deviations can be attributed to multiple reasons. For example, many of the software systems are developed using a top-down development approach. As a result, implementation level changes are not propagated back to the architectural documents resulting in inconsistencies. Recovering the architecture, therefore, is an essential step for the migration. While many useful architecture visualisation tools such as CodeSonar[4] and Imagix[5] analyse the source code to provide architecture visualisation, they only provide information on the logical structure of the software and additionally, they may not be able to detect faulty architectural patterns within the recovered architecture.

In this phase, we focus primarily on extracting the temporal properties of the system, which can manifest themselves in different forms such as deadlines or message buffer sizes. We consider the system to be modelled with cause-effect task chains [31], which implicitly consider maintaining the causality in the underlying communication. These chains are constrained by the timing constraints similar to that of the AUTOSAR standard. At the task-level abstraction, each task can be represented in terms of its period, worst-case execution time and various types of timing requirements such as deadline, data age, and data reaction constraints [32]. Note that the tasks and their corresponding software components at the software architecture abstraction have the read-execute-write semantics, which allow them to be adapted to comply with the Logical Execution Time (LET) model. In addition to these, there can be indirect temporal requirements such as the number of messages in a message queue should not be less than a specific value during a certain operating mode, which then requires that the task producing the messages for the queue

[2]https://www.omg.org/omgmarte/
[3]http://www.mechatronicuml.org/en/index.html
[4]https://www.grammatech.com/products/code-visualization
[5]https://www.imagix.com/index.html

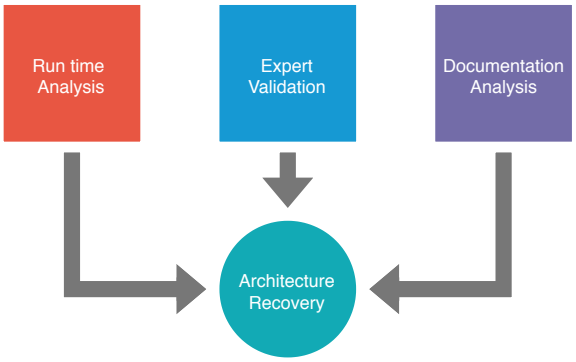

Fig. 3. Architecture analysis.

can be blocked only for a duration that does not violate this requirement. These kinds of properties are difficult to obtain from static analysis tools. Therefore, we need a comprehensive multi-dimensional software comprehension and reverse engineering approach to extract such information from the existing software architecture, specifically, the timing properties and constraints, which are crucial in verifying timing predictability of the system [32].

To extract the necessary timing-related software properties, we relied on multiple data sources. Essentially, the architecture documentation, run-time execution logs and domain experts. In the following, we justify the need for analysing each of these sources and identify the information that can be obtained from them, also shown in Fig. 3.

*a) **Documentation Analysis**:* The architecture of large software intensive systems is normally documented according to the "4+1" architectural view model [33] or an enhanced variant. The format for architecture documentation can vary depending on the internal process and industry-relevant certification requirements. SysML [34] and UML models are some of the formal description formats for documentation used in the industry. Complementing such formal description formats are the textual documents explaining the architecture in natural language as a part of the documentation. These high-level architectural models and documents identify the different components of the system and the interaction between components, summarise the design patterns and technologies employed in the implementation and provide a concise overview of the functions of these components. By analysing these documentation artefacts, it should be possible to identify chains of dependent components, the tasks associated with these components and the expected timing behaviours. However, during our analysis, we found that the existing documentation did not contain any information mapping different tasks to their respective components and information on expected timing behaviours was either unavailable or was incomplete in the architecture documents.

*b) **Run-time Analysis**:* While the high-level documents are good sources of information, the information provided by such documentation may either be incomplete or may not reflect the actual implementation. One reason for such an

inconsistency is due to the structure of the development process, where the information flow is usually top-down, and the changes made at the implementation level are not propagated back to the architecture documents [35]. Additionally, these industrial software systems have been incrementally developed over many years with the addition of new functionality, bug fixing, and other optimisations in each increment. Therefore, due to the accumulation of undocumented changes made during implementation over the years, relying solely on high-level documentation as the only source of information for modelling the system can result in an inaccurate representation of the expected system behaviour. This makes it necessary to consider the run-time logs as complementary sources of the system information. One approach to understanding the run-time behaviour of the system is the tracing and measurement-based approach [36]. Using this approach, information such as number of context switches, response times, execution times, number of task instances, periodicity of the tasks, among others can be collected. By using dynamic analysis and visualisation tools such as Tracealyzer [36], additional information such as the communication flow between different tasks, identification of shared resources, task chains and precedence constraints between the tasks can be obtained. The information gained from the run-time analysis can be used to refine and enhance the model.

The run-time analysis comes with its own set of conundrums. As the system under consideration is configurable, i.e., the user can configure and specify the runtime behavior, it is difficult to identify a configuration that can be single representative of possible configurations for run-time analysis. One possible approach to address this issue is to use the "maximum load" approach. We consider the system to be in "maximum load" state, if under normal operation mode, all system tasks are active and that each task is executing its most computationally heavy or memory intensive jobs. Relying on a single configuration, however, is not sufficient to make any statistically reliable conclusions about the measurements. Therefore, another argument would be to gather run-time behaviour from as many possible configurations as feasible. Again, identifying this "feasible" number is not straight forward. This is made even more complicated by the continuous development process, where code is modified and new builds generated daily. Identifying a fixed version of the software for analysis becomes non-trivial for such cases. To simplify the process and move forward, we use a maximum load configuration and ensure that all system software components are active during the trace period. Note that we rely on the latest released version of the software.

During the run-time analysis of our system, we found that there were inconsistencies between the expected and observed behaviours. A few of the inconsistencies were a result of incorrect configuration of the instrumented code, while others were actual deviations from the expected behaviour. This observation highlights the fact that relying on a single source for information is not only ineffective but also error-prone. This necessitates the need for expert validation of the collected information to create a sufficiently accurate system model.

*c) **Expert Validation**:* Architectural design decisions are made by analysing multiple factors such as domain requirements, dependencies on services provided by the operating systems and the underlying hardware platform, among others. However, the high-level architectural models and documents do not describe the rationale behind the design decisions and even if they do, such information is limited. Moreover, in legacy systems, such documents do not completely reflect the implementation [35]. Furthermore, as the information from the run-time analysis is quantitative and statistical in nature, it is possible to misinterpret any deviation from a commonly occurring pattern as an inconsistency whereas this could have been a design decision. To avoid such misinterpretations and improve system model accuracy, discussions with domain experts are mandatory during the architecture analysis. These discussions will be used to understand the rationale behind the design decisions, and to validate the observations of the documentation and the run-time analysis phases. In our work, we were able to validate the inconsistencies such as the deviation from a commonly occurring pattern as a design decision and also mark some of the observed results as an outcome of incorrect code instrumentation configuration.

D. Architecture Transformation

As discussed earlier, the architecture transformation phase focuses primarily on evaluating potential solutions and identifying the most appropriate ones for the final implementation. Before we evaluate any solution, we need to identify the system requirements that need to be considered to identify, evaluate and qualitatively rank possible solutions. Since in our case, the migration to multi-core will primarily affect the runtime behaviour, we focus on the explicit temporal requirements, implicit requirements such as the number of messages in a queue and assigned QoS levels to different functional domains. An important requirement here is to ensure that this transformation results in improved system predictability, performance and that the architecture is scalable in terms of the number of cores and new functionality that needs to be integrated into future versions of the software. Since the terms predictability, performance, and scalability are generic in nature, we need to ensure that we have measurable definitions for these terms. Once we define the evaluation criteria, we then move towards the evaluation process itself. The evaluation can be carried out in various ways depending on the evaluation metric and the solution being considered, such as simulation, model-checking and analytical calculations. Once the evaluation of possible solutions is complete, we rank these solutions based on an agreed evaluation metric and based on these rankings, we select the solutions for the final implementation phase. To ensure that this transformation is systematic, we divide the transformation phase into the following steps:

1) identification of potential solutions;
2) evaluation of the solutions;
3) ranking of the solutions;
4) selection of the solutions.

*a) **Identification of potential solutions**:* Identification of potential solutions can be done in many different ways. Although we don't make any specific recommendations, we would like to point out that the number of potential solutions could be infinitely many and we hypothesize that evaluating each solution will be impossible. Especially in the case of real-time systems, where the search space in terms of near-optimal solutions is large [10], [11], [37], [38]. Therefore, a good starting point in this stage could be the domain experts. Also, the information from the architecture abstraction and recovery phases can be a useful guide in reducing the search space. In our case, we use expert interviews and review the state-of-art in the real-time systems domain to identify potential solutions. Another important consideration is that since application developers are focused primarily on the application functionality, they rely on the operating systems to provide support for real-time properties. This implies that in many cases, only those mechanisms supported by an operating system can be considered as part of the potential solution set.

As highlighted earlier, the purpose of an abstract system model is to capture all the relevant properties of the system but without the functional complexity. This enables creation of synthetic tasks for simulation and verification of new design solutions. These abstract task sets can be modified and verified in short time spans when compared to modification of the actual implementation of the system. Many of the real-time workload models such as those reviewed in [22] have been successfully used to represent practical systems such as in the avionics domain as well as in the automotive domain. While many of these workload models consider the tasks to be independent, we found that the system under study violates this assumption and that new jobs of tasks are triggered by jobs of other tasks. Also, the presence of event triggered components within the system along with multi-rate task chains implementing a single functionality, requires that the precedence constraints as well as task chains be considered when considering potential solutions [31].

Some of the relevant issues that should be addressed by the potential solutions for transitioning from single core to multi-core platforms were highlighted by Macher et al. [39], and Nemati et al. [40]. For example, use of single-core hardware implies that the system tasks execute in sequential manner. If run on multi-core, the task precedence constraints may not be maintained affecting system dependability. Additionally, systems designed for single-core do not require any mapping of software and multiple compute resources. However, predictable execution on multi-core is provided by partitioned scheduling approaches [38]. Ad hoc partitioning can affect system performance and scalability. Multi-level caching can cause data inconsistencies when tasks sharing a variable are executing on different cores [41]. In the case of fixed-priority scheduling, priority assignment can impact response times [37]. Along with these technical challenges, maximizing the re-use of the system artefacts and the minimal re-engineering of the system software are essential requirements that must be considered in the migration process.

*b) **Evaluation of the solutions**:* Once the potential solutions have been identified, the next step is to evaluate these solutions. By evaluation, we refer to the application of the potential solutions from the previous step to the abstract model from the architecture recovery stage and measurement of the identified metrics. The evaluation can be done in different ways as already highlighted earlier such as simulation in the case of ART-ML framework [30] or the Cheddar tool [42], analytical calculations if using techniques such as those identified in [38] or model-checking if using the timed automata approach specified in [43].

We would like to point out that given the safety-critical nature and complexity of the system, we hypothesize that the potential solution identification and evaluation steps are rather time consuming and are critical in the migration process. The time spent during these phases can potentially result in practical solutions that ensure that the migration process is successful in meeting the extra-functional requirements.

Moving forward, we return to the question of identifying the best solution among the many evaluated solutions. To guide in this direction, we use the ranking approach as follows.

*c) **Ranking of the solutions**:* The ranking step of the transformation phase basically orders the evaluated solutions in terms of certain criteria. To order the evaluated solutions, we propose to use the following multi-step approach:

- identify parameters to rank potential solutions;
- provide measurable definitions to the identified parameters;
- arrive at a consensus on measurement methods for the parameters;
- prioritize or assign weights to the parameters for trade-off analysis;
- rank the evaluated solutions.

We believe that this approach provides a systematic way to measure effectiveness of the evaluated solutions and guide in selection of the final solution. By identifying measurable parameters, the methods to measure them, and prioritize them if a trade-off is necessary, we can remove any ambiguity associated with the perceived effectiveness. To identify these parameters, we rely on expert interviews and a focus group discussion consisting of the different domain experts.

*d) **Selection of the solutions**:* Once the potential solutions have been evaluated and ranked, the selection of final solutions should be rather straight forward. However we would like to point out the fact that there could be solutions that may optimize one requirement while negatively affecting another. This situation would need a systematic trade-off analysis. We leave further discussion on trade-off analysis for future work.

E. Architecture Verification

The last step in the architecture transformation phase is the verification of the transformed architecture. Here we essentially verify if the transformed architecture complies with requirements from the architecture requirements specification phase and the recovery phase. The verification stage is rather simple and straight forward since the different steps in the

transformation phase involve verification in the evaluation stage with the systematic ranking and selection approach.

V. Implementation Migration

So far, we discussed the transformation at the architecture level of the system in our migration process. We now discuss the processes necessary to implement the transformed architecture at the source code level. Although not directly related to the migration process itself, we consider that some form of refactoring at the source-code level may be necessary prior to the migration process. Depending on the existing logical architecture and the quality of the software, the refactoring may address different concerns. For example, removal of duplicate and dead code, creating components based on functionality, adoption of the layered architecture similar to AUTOSAR. For further discussion, we assume that the system has a layered architecture with well-defined components, that the logical architecture is capable of handling new components and modifications in the abstraction layers, and that the source code is separated according to the components.

Further, we classify the architecture solutions as abstract component level or functional component level solutions. For example, if the solution is a new priority order for the tasks, then it is functional component level solution if the tasks are associated with the component and that the priorities can only be changed in the component files. If it is a new synchronization protocol, then it is an abstract level solution, which is used by all components and may need a new implementation. Therefore, before we make the changes, we identify components that need to be modified, map solutions that need new components and then implement the changes.

A. Component Identification and Creation

The solutions selected during the transformation phase may require that changes be made to the existing components in the system. For example, if the components use nested semaphores and if the identified solution does not support nested semaphores, then such nested semaphores need to be removed. To do this in a systematic manner, we index and categorize the transformed solutions, review the solutions with the domain experts and component owners and associate each component with the solution that requires that component to be modified. If there are solutions that are classified as abstract-level solutions or which could not be mapped to existing components, we create new components for such changes.

B. Implementation

Once all components have been identified for modification and new components created, the necessary changes are implemented in the source code. Although the concurrency related issues are addressed during the architecture transformation phase, it is possible that they could manifest during the implementation stage. Therefore, coding guidelines that address these issues are provided to the developers to minimise the manifestation of these issues during the implementation.

VI. Verification Migration

The system verification and validation stage is the final stage of the migration process. Typically, for the system such as the one being considered, a reliable verification process is already in place. This includes the usual verification approaches such as unit testing, functional testing, and system integration tests. Since the architectural transformation is primarily related to the runtime behaviour and performance, we expect that most, if not all existing tests related to functional behaviour to be valid. Therefore, we hypothesise that any failures here could be related to the concurrent execution of the system tasks. To maintain the quality of the system software, we focus on augmenting the existing tests with concurrency related testing approaches along with performance verification. Again, to approach this enhancement in a systematic way, we divide the verification migration process into concurrency testing and the migration validation phase.

A. Concurrency Testing

The goal during this phase is to augment the existing verification process to identify concurrency related issues. These include race conditions, atomicity violations and deadlocks. A comprehensive review can be found in the work by Bianchi et al. [44]. We propose the analysis of solutions during the architecture transformation phase to identify scenarios that could lead to potential concurrency issues. This way, it will be possible to create tests for those specific scenarios. Additionally, static code analysis that identifies concurrency bugs is added to enhance the verification process.

B. Migration Validation

During this phase, we focus on validation of the migration process itself. We begin by identifying the parameters to qualitatively validate the outcome of the process. We use two metrics for this purpose: (i) results of the functional and system integration tests, and (ii) performance related parameters such as response times. In the first case, the number of failures should not be greater than those from the pre-migration version. In the second, the values of the performance parameters should not be less than those measured with the pre-migration version. We point out here that although the validation is the last step, depending on the development process, this validation can be applied to each build prior to release. By using the results of the validation with each build, the pace of the migration process can be measured.

VII. Tools for Migration

Software migration from single-core to multi-core architectures is a complex process and requires the use of different tools at different stages of the migration process. Here, we review some of the tools that can be used during the different phases of the migration process.

A. Architecture Representation

Software requirements and the architecture can be described in natural language and as models using different modelling languages such as the UML. For embedded systems with timing requirements, there exist many tools that allow modelling and specification of different views of the system. The APP4MC tool[6], allows modelling and specification of the hardware as well as software components and provides support for scheduling algorithms. Another tool is the MARTE [45] profile for UML. The MARTE profile extends the UML models to include description of timing requirements. The MAST tool-suite[7] allows for modelling as well as performing automatic schedulabilty analysis and supports many of the common scheduling algorithms for single-core as well as multi-core architectures. UPPAAL [27] is another tool for modelling the software as timed-automata and it supports model checking for formal analysis and verification. A few concerns with many of these tools are that some have steep learning curves, while others such as UPPAAL are not scalable to large systems and almost all lack support for automatic conversion of existing source code to abstract models.

B. Architecture Recovery

For architecture recovery, static code visualization tools such as CodeSonar and Imagix could be used. For dynamic analysis, tools which provide visualization of the run-time behaviour along with statistical information on timing properties can be effective. For example, Tracelyzer allows visualization of the run-time behaviour and provides different views to analyse this information.

VIII. Conclusion

Migration of complex real-time embedded software from single-core to multi-core computing platforms is non-trivial. To ensure a successful migration of these software systems, a systematic approach is needed that takes multiple software engineering perspectives into account such as software processes, software architectures, requirements engineering, reverse engineering, model-based development, real-time scheduling and schedulability analysis. In this paper, we presented a systematic multi-stage methodology for migrating real-time industrial software systems from single-core to multi-core computing platforms. In this regard, we studied a complex real-time software system from the automation industrial domain that requires such a migration. We used focus group discussions, expert interviews and reviewed the literature to guide the development of the migration strategy. We identified the software architecture transformation as the main phase in the migration process and presented a systematic approach to perform the transformation with emphasis on the architecture recovery and an evaluation mechanism for possible multi-core solutions. To select suitable solutions from the set of evaluated approaches, we proposed ranking of these solutions based on measurable parameters for the final implementation and we reviewed some of the tools that can be used during the migration process.

[6]https://www.eclipse.org/app4mc/

[7]https://mast.unican.es/

Acknowledgement

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation, the Swedish Governmental Agency for Innovation Systems (VINNOVA) through the project DESTINE, and the Swedish Knowledge Foundation (KKS) through the projects HERO, FIESTA and DPAC.

References

[1] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance*, vol. 13, no. 1, pp. 3–30, Jan. 2001.

[2] Ralf Reussner, Michael Goedicke Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, Lukas Märtin, Ed., *Managed Software Evolution*. Springer Nature Switzerland AG, 2019.

[3] J. Kraft, Y. Lu, C. Norström, and A. Wall, "A Metaheuristic Approach for Best Effort Timing Analysis Targeting Complex Legacy Real-Time Systems," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 258–269.

[4] G. Mustapic, A. Wall, C. Norstrom, I. Crnkovic, K. Sandstrom, J. Froberg, and J. Andersson, "Real world influences on software architecture - interviews with industrial system experts," in *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture* , 2004.

[5] J. Guhl, S. Tung, and J. Kruger, "Concept and architecture for programming industrial robots using augmented reality with mobile devices like microsoft HoloLens," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation* , pp. 1–4.

[6] V. Paelke, "Augmented reality in the smart factory: Supporting workers in an industry 4.0. environment," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation* , pp. 1–4.

[7] G. Mustapić, J. Andersson, C. Norström, and A. Wall, "A Dependable Open Platform for Industrial Robotics – A Case Study," in *Architecting Dependable Systems II*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer Berlin Heidelberg, 2004, pp. 307–329.

[8] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, Jan 2004.

[9] S. Mubeen, E. Lisova, A. Feljan, "A Perspective on Ensuring Predictability in Time-critical and Secure Cooperative Cyber Physical Systems", *20th IEEE International Conference on Industrial Technology*, 2019.

[10] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 56:1–56:38, 2019.

[11] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for real-time systems," *LITES*, vol. 6, no. 1, pp. 04:1–04:53, 2019.

[12] G. Mustapić, J. Andersson, C. Norström, and A. Wall, "A dependable open platform for industrial robotics – a case study," in *Architecting Dependable Systems II*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 307–329.

[13] Leszek Wlodarski, Boris Pereira, Ivan Povazan, Johan Fabry, and Vadim Zaytsev, "Qualify First! A Large Scale Modernisation Report," in *SANER*. IEEE, 2019, pp. 569–573.

[14] P. Church, H. Mueller, C. Ryan, S. V. Gogouvitis, A. Goscinski, and Z. Tari, "Migration of a SCADA system to IaaS clouds – a case study," *Journal of Cloud Computing*, vol. 6, no. 1, p. 256, 2017.

[15] K. Plakidas, D. Schall, and U. Zdun, "Software Migration and Architecture Evolution with Industrial Platforms: A Multi-case Study," in *Software Architecture*, ser. Lecture Notes in Computer Science, C. E. Cuesta, D. Garlan, and J. Pérez, Eds. Cham: Springer International Publishing, 2018, vol. 11048, pp. 336–343.

[16] H. M. Sneed, "Planning the reengineering of legacy systems," *IEEE Software*, vol. 12, no. 1, pp. 24–34, 1995.

[17] Ravi Erraguntla and Doris L. Carver, "Migration of sequential systems to parallel environments by reverse engineering," *Information & Software Technology*, vol. 40, no. 7, pp. 369–380, 1998.

[18] M. Battaglia, G. Savoia, and J. Favaro, "Renaissance: a method to migrate from legacy to immortal software systems," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, 1998, pp. 197–200.

[19] A. Menychtas, K. Konstanteli, J. Alonso, L. Orue-Echevarria, J. Gorronogoitia, G. Kousiouris, C. Santzaridou, H. Bruneliere, B. Pellens, P. Stuer, O. Strauss, T. Senkova, and T. Varvarigou, "Software modernization and cloudification using the ARTIST migration methodology and framework," *Scalable Computing: Practice and Experience*, vol. 15, no. 2, 2014.

[20] L. Forite and C. Hug, "FASMM: Fast and Accessible Software Migration Method," in *2014 IEEE Eighth International Conference on Research Challenges in Information Science*. IEEE, pp. 1–12.

[21] C. Wagner, *Model-Driven Software Migration: A Methodology*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014.

[22] M. Stigge and W. Yi, "Graph-based models for real-time workload: a survey," *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, 2015.

[23] Björn B. Brandenburg, "Scheduling and Locking in Multiprocessor Real-Time Operating Systems," PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[24] F.Herrera, H. Posadas, P.Peñil, E.Villar, F.Ferrero, R.Valencia, and G.Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55–78, (2014).

[25] K. Hänninen, J. Mäki-Turja, M. Sjödin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "The Rubus Component Model for Resource Constrained Real-Time Systems," in *3rd IEEE International Symposium on Industrial Embedded Systems*, 2011.

[26] S. Mubeen, H. Lawson, J. Lundbäck, M. Gålnander, and K. L. Lundbäck, "Provisioning of predictable embedded software in the vehicle industry: The rubus approach," in *IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, 2017.

[27] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, Dec. 1997.

[28] W. Schäfer and H. Wehrheim, *Model-Driven Development with Mechatronic UML*. Springer Berlin Heidelberg, 2010, pp. 533–554.

[29] The AUTOSAR Consortium, "Autosar technical overview," in *Version 4.3.*, May 2016, http://autosar.org.

[30] A. Wall, "Architectural Modeling and Analysis of Complex RealTime Systems," PhD thesis, Mälardalen University, Västerås Sweden, 2003.

[31] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, pp. 104 – 113, 2017.

[32] S. Mubeen, T. Nolte, M. Sjödin, J. Lundbäck, and K.-L. Lundbäck, "Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints," *Software & Systems Modeling*, vol. 18, no. 1, pp. 39–69, Feb 2019.

[33] P. B. Kruchten, "The 4+1 View Model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[34] E. Andrianarison and J. Piques, "SysML for embedded automotive systems: a practical approach." in *Conference on Embedded Real Time Software and Systems*. IEEE, 2010.

[35] U. Eliasson, R. Heldal, P. Pelliccione, and J. Lantz, "Architecting in the Automotive Domain: Descriptive vs Prescriptive Architecture," in *12th Working IEEE/IFIP Conference on Software Architecture, 2015*.

[36] J. Kraft, A. Wall, and H. Kienle, "Trace recording for embedded systems: Lessons learned from five industrial projects," in *Proceedings of the First International Conference on Runtime Verification (RV 2010)*. Springer-Verlag (Lecture Notes in Computer Science), November 2010.

[37] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems," *Journal of Systems Architecture*, vol. 65, pp. 64–82, 2016.

[38] B. B. Brandenburg and M. Gul, "Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations," in *IEEE Real-Time Systems Symposium, 2016*.

[39] G. F. H. Macher, Andrea Höller, Eric Armengaud, and C. J. Kreiner, "Automotive Embedded Software: Migration Challenges to Multi-Core Computing Platforms," in *Proceedings INDIN 2015*, 2015, pp. 110–118.

[40] F. Nemati, M. Behnam, and T. Nolte, "Efficiently migrating real-time systems to multi-cores," in *2009 IEEE Conference on Emerging Technologies & Factory Automation*, pp. 1–8.

[41] S. A. Asadollah, H. Hansson, D. Sundmark, and S. Eldh, "Towards Classification of Concurrency Bugs Based on Observable Properties," in *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pp. 41–47.

[42] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *SIGAda*. ACM, 2004, pp. 1–8.

[43] C. Norstrom, A. Wall, and W. Yi, "Timed automata as task models for event-driven systems," in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications*, 1999, pp. 182–189.

[44] F. A. Bianchi, A. Margara, and M. Pezz, "A survey of recent trends in testing concurrent software systems," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 747–783, Aug 2018.

[45] "The UML profile for MARTE: Modeling and analysis of real-time and embedded systems." OMG Group, 2010.