A Systematic Methodology to Migrate Complex Real-Time Software Systems to Multi-Core Platforms

Shaik Mohammed Salman[a], Alessandro V. Papadopoulos[b], Saad Mubeen[b], Thomas Nolte[b]

[a]*ABB AB, Västerås, Sweden*
[b]*Mälardalen University, Västerås, Sweden*

Abstract

This paper proposes 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 offer a promising solution to address these limitations by providing increased computational power and allowing parallel execution of different applications within the system. However, the transition from traditional single-core to contemporary multi-core computing platforms 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. The methodology was evaluated using a survey-based approach and the results indicate that the presented methodology is feasible, usable and useful for real-time industrial software systems.

Keywords: Real-time systems, multi-core, software architecture, software migration, robotics.

1. 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, revision or release [1, 2]. Many of the industrial systems have been developed over the decades [3], undergoing major revisions due to technology shifts, changing customer requirements, improved development processes, among others. One constant factor associated with the evolution of such systems is that the software architectures and the implementations have focused on single-core computing platforms. Integrating new data-intensive and computationally demanding applications withing the system, however, requires additional computational capacity. Moreover, with the decreasing availability of the single-core processors, migrating the existing software to multi-core computing platforms is becoming a necessity. By *migration*, we refer to the modification of the existing software to execute on the multi-core platforms, while ensuring that the performance and quality attributes, such as dependability [4, 5], match the current system quality and more optimistically, improved much further. Such migration is essential since the long life-cycle of existing software systems has resulted in the creation of assets that have become critical for a business [6] and that a complete redevelopment may not be feasible.

Migrating existing real-time software systems towards multi-core systems requires (i) Identifying the timing requirements of the existing software systems and (ii) Identifying the technical solutions that can improve the performance, resource usage and the timing predictability of the software systems [7, 8, 9]. Invariably, any migration approach should also address the extra-functional attributes such as scalability, maintainability and portability of the software. Furthermore, the migration should consider maximum reuse of the existing software while minimizing the re-engineering efforts.

To address these aspects for the migration of a complex real-time software system with strict timing and

Email addresses: shaik.salman@se.abb.com (Shaik Mohammed Salman), alessandro.papadopoulos@mdh.se (Alessandro V. Papadopoulos), saad.mubeen@mdh.se (Saad Mubeen), thomas.nolte@mdh.se (Thomas Nolte)

dependability requirements, we used 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 refined this question 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 motivated by the need for migrating a configurable robot controller software [4] developed at ABB Robotics[1], with functionality ranging from motion control to cloud connectivity. The controller software has close to 140 tasks and 71,128 methods, integrating 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 used a mixed research methodology utilising discussions within 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 systematic methodology for migrating complex embedded software from single-core to multi-core platforms;
- A review of tools that facilitate the migration process; and
- A survey-based evaluation of the proposed methodology.

This paper reinforces the validity of the methodology presented in our previous work [10] by including a survey-based evaluation of the methodology.

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

The rest of the paper is organised as follows. Section 2 provides an overview of a robotic system and its controller software. Section 3 reviews the existing software migration methods. Section 4 provides an overview of the overall methodology. Section 5 includes a systematic approach focusing on architecture migration, followed by implementation and verification of the migration in Section 6 and Section 7 respectively. A review of the tools facilitating the migration process is discussed in Section 8. Section 9 presents the evaluation of the proposed methodology. Finally, Section 10 concludes the paper.

2. System Overview

The system corresponds to a typical robotic system consisting of a manipulator arm, a controller, and a graphical controller interface. The paper focuses on the software functionality of the controller, which can be divided into functions concerning (i) configuration, (ii) communication, and (iii) control. The configuration functions provide the robot programming interface that allows a user to configure and specify the runtime behaviour 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 interactThe communication functions provide a real-time networking capability to enable the controller to interact with devices such as Programmable Logic Controllers (PLCs). It also includes a non-real-time communication capability that allows the controller to interact with enterprise network including PCs and the cloud. The control functions generate the path the manipulator has to follow based on the user-defined configuration. The output of the control functions is used to drive controllers that manage the low-level motor actuation.

The controller software has different runtime modes and the available functions vary between the modes. The main modes include the "Initialisation mode", "Safe-init mode", "System update and configuration mode", "Normal operation mode", and "Fail-safe mode" [11]. The different modes and the transition between the modes is shown in Fig. 1. At startup, the controller transitions into the initialisation mode. Here all the tasks are initialised with values based on the previously saved configuration settings.   The controller software is in the initialisation mode during startup. It enters the safe-init mode if there are errors during the startup. The behaviour of the controller software can be configured in the system update and configuration mode. ItOnce the required configuration has been set,

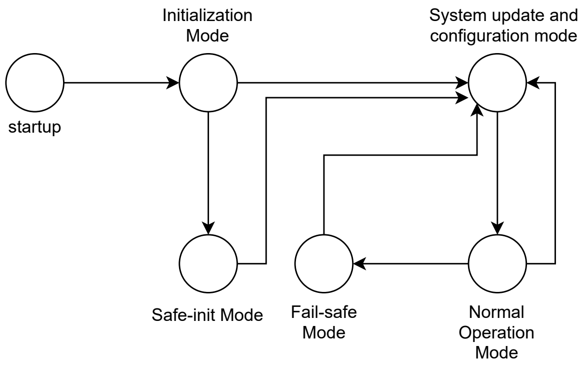

Figure 1: Main Modes in the System

the controller enters the normal operation mode. This is the operational mode of the controller, where the physical movement of the robot arm is enabled. It is in this mode that the controller executes the motion planning algorithms with real-time communication enabled during the normal operation modefor data exchange with external sensors and actuators. It transitions into a fail-safe mode from the normal operation mode if an unexpected error such as an unresponsive sensor, or detection of possible collision with unexpected objects 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.

Timing related properties of a subset of the tasks that make up the robot controller is provided in the Table. 1. The RT communication component is responsible for ensuring real-time communication between the controller and the sensors and actuators. It consists of a network driver task along with a runtime middleware task that provides the necessary interface for data exchange with other tasks.There are two tasks, namely TS_Ethercat and TS_RT, that are responsible for real-time communication between the controller and the sensors and actuators. The TS_Ethercat task comprises the network driver, whereas the TS_RT task encapsulates the runtime middleware that provides the necessary interface for data exchange with other components. The two tasks are activated by periodic timers of 10 ms period each and their worst-case execution times (WCETs) are 120 μs and 80 μs respectively. The priorities of TS_Ethercat and TS_RT are 12 (highest) and 11 (second highest) respectively. Furthermore, the *utilization* of these two tasks are 0.012 and 0.008 respectively. The utilization of a task represents the portion of CPU time required by the task and is calculated by dividing the WCET of the task by its period. The Non RT communication component provides web-based connectivity for communication with enterprise network and for uploading robot programs and managing and updating the controller configurations. It consists of a network driver task, a non real-time middleware task and the web server task, with the web-server task providing the interface for data exchange between the controller and external devices.The TS_Ethernet, TS_NRT and TS_Web tasks are responsible for non real-time communication such as web-based connectivity for communication with enterprise network and for uploading robot programs and managing and updating the controller configurations. These tasks encapsulate the network drivers, non real-time middleware and web server providing an interface for data exchange between the controller and external devices respectively. The robot program interpreter component is responsible for converting the robot program into controller data structures that act as inputs for the trajectory generation component of the controller. It consists of two tasks, the TS_RPI and the TS_RPI_Transform.The robot program interpretation is performed by the TS_RPI and TS_RPI_Transform tasks. These tasks are responsible for converting the robot program into controller data structures that act as inputs for the trajectory generation functionality of the controller. The TS_RPI task parses the robot program and validates its syntactical correctness. The TS_RPI_Transform task then converts the robot program into a data structure that can be used as input for the trajectory generation functionality, which allows planning of the robot motion and generating the required set-points for the controller task (TS_Control). The trajectory generation functionality is realised with the tasks TS_IPL_Path and TS_IPL_JointPath. Further, the controller software includes the system state manager tasks, namely TS_Sys_Events and TS_Sys_Backup, that are responsible for managing different system level signals and generating events that define the behaviour of other tasks. For example, the system state manager task can observe a change in the state of the safety switch signal and generate an event that will trigger a mode change from normal operation mode to a fail-safe mode.

3. Related Work

Software migration is usually carried out when adopting a different architectural paradigm than the existing one, such as changing the programming language [12]

System Functions	Task functionality	Task	Task Trigger Type	Task Priority	Task Period (ms)	WCET (us)	Utilization
RT Comm.	Network driver	TS_Ethercat	timer	12	10	120	0.012
RT Comm.	Network middleware	TS_RT	timer	11	10	80	0.008
Non RT Comm.	Network Driver	TS_Ethernet	timer	5	10	75	0.0075
Non RT Comm.	Network Middleware	TS_NRT	timer	4	50	800	0.016
Non RT Comm.	Application	TS_Web	timer	2	100	200	0.002
Robot Program Interpreter	Parse robot program	TS_RPI	event from TS_NRT	3	50	4000	0.08
Robot Program Interpreter	Format data for trajectory generation	TS_RPI_Transform	event from TS_Sys_Events	6	20	200	0.01
System State Manager	Monitor and handle system state events	TS_Sys_Events	periodic	10	10	60	0.006
System State Manager	Create system backup	TS_Sys_Backup	event from TS_WEB	1	100	200	0.002
Trajectory Generation	Interpolate Cartesian Path	TS_IPL_Path	timer	7	20	2000	0.1
Trajectory Generation	Interpolate Joint Space Path	TS_IPL_JointPath	timer	8	20	200	0.02
Controller	Create setpoints and receive feedback for motor drivers	TS_Control	timer	9	2	100	0.05

Table 1: Subset of the tasks in the Robot Controller.

or when moving from native server deployments to cloud-based deployments [13, 14]. Sneed [15] 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. [16] 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 [17] presented the *RENAISSANCE* method for re-engineering a legacy system. The method focuses on planning and management of the evolution process.

Menychtas et al. [18] presented a framework called *ARTIST*, a three-phase approach for software modernization focusing on migration towards the cloud. They categorised the migration into three main phases, *Pre-migration, Migration and Modernisation* 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 modernisation phase, the actual migration is carried out and finally during the Post-migration phase, the system is deployed and validated. Forite et al. [19] 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 [20] 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 along with the interference due to the contention for shared resources such as the caches affecting

Figure 2: Proposed migration workflow.

the timing predictability of the overall software system.

4. Migration Methodology

Based on the reviewed methods and the extra-functional requirements, we create a migration workflow as depicted in Fig. 2 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 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.

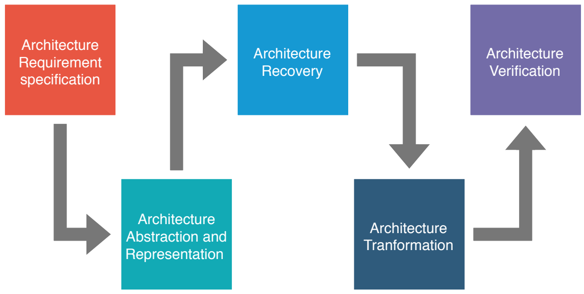

Figure 3: Various phases in the software architecture migration.

5. Software Architecture Migration

Many of the real-time systems including the robot controller software have a strong focus on timing, safety and dependability requirements. Therefore, we need a well-defined software architecture to support such requirements. As there are significant differences in the single-core and multi-core platforms, the existing software architecture should be modified to address the constraints of multi-core platforms and make the best use of the available resources. To approach this modification systematically, the software architecture migration stage is divided into five well-defined phases as shown in the Fig. 3. The five phases are :

1. Architecture requirements specification;

2. Architecture abstraction and representation;

3. Architecture recovery;

4. Architecture transformation; and

5. Architecture verification.

5.1. Architecture Requirements Specification

The architecture requirements specification is the first phase of the architecture migration process. The requirements are essentially high-level and the extra-functional requirements of scalability, performance and timing guarantees are the guiding principles for the complete migration process. The more concrete requirements are defined during the architecture recovery phase of the migration process. We also include the identification of a requirements specification and management process in this phase to better manage the requirements for the rest the migration process.

5.2. Architecture Abstraction and Representation

In this phase, we seek to identify an abstraction level that can accurately represent 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. Further, a representation model that can sufficiently capture the system properties should be identified. The representation model should be easy to comprehend, and should act as a communication tool between different stakeholders such as the system architects and developers. To address these issues, we rely on expert interviews and the review of 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.

Software Architecure, Real-time Task Models and Representation Tools. The system we considered provides multiple functionalities ranging from embedded control to cloud connectivity. 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 were able to identify that the task-level abstraction provides the necessary semantics to capture the system properties and therefore, can be used during the later stages of the migration process. Moreover, most of the literature in real-time systems uses the task-level abstraction for the system representation [21, 8].

Many modelling languages support the task level abstraction to represent the architecture of real-time systems.There are several modelling languages that allow modelling of software architectures and task-level abstraction models of real-time systems. The UML MARTE[2] profile [22], Rubus Component Model [23, 24], UPPAAL [25], MechatronicUML[3] [26], AUTOSAR [27], ART-ML Framework [28], are some of the possible modelling languages and frameworks that can be used to represent the system under discussion.

It is worthwhile to mention that although many of these languages, frameworks and supporting tools offer detailed semantics for capturing multiple viewpoints which are essential for managing real-time systems, the learning curve for many of these tools is however, rather steep, especially when being used for representing task-level abstraction of existing systems.To demonstrate the software architecture abstraction in the proposed methodology, we model the software architecture of the robot controller using the Rubus Component Model as shown in Fig. 4. Note that the Rubus Component Model and its runtime environment consider a one-to-one mapping between a *software component* and a task. A software component is the lowest-level hierarchical element in a component model that is used to model the software architecture of a system. The software component is a design-time entity that may correspond to one or more tasks at runtime. For example, the model of a software component that conforms to the Rubus Component Model (RCM) [23, 24] is shown in Fig. 5. A software component communicates with other components by means of input and output data and trigger ports. The trigger ports indicate when the task (corresponding to the software component) is activated for execution. A software component can be triggered by an independent source (e.g., a periodic clock) or by another software component. The properties of the software component such as their execution times, activation periods and priorities are specified using the values from Table 1. Note that there are two timing constraints, namely Age (50ms) and Reaction (50 ms), that are specified on a chain of software components within the software architecture in Fig. 4. These timing constraints conform to the AUTOSAR standard and are supported by several other modelling languages and methodologies for real-time systems [29].

5.3. 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

[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`

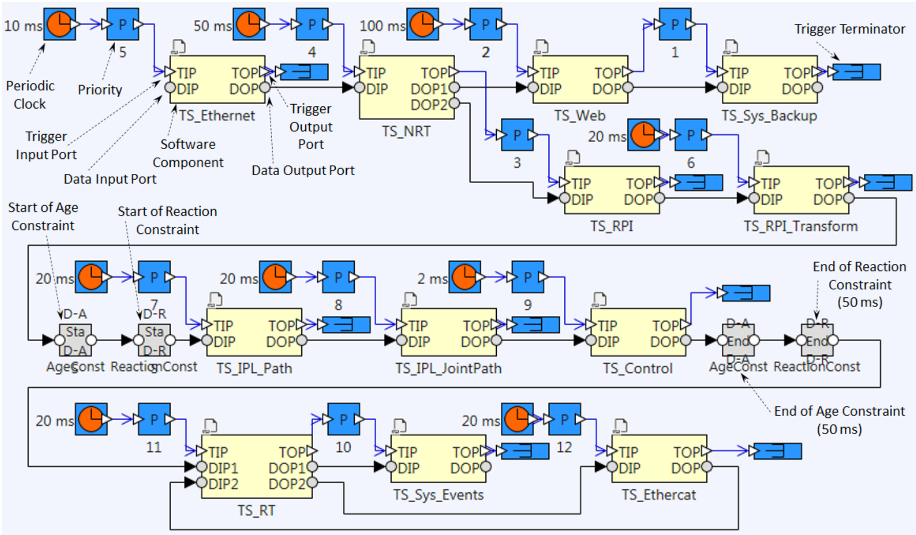

Figure 4: Software Architecture Representation.

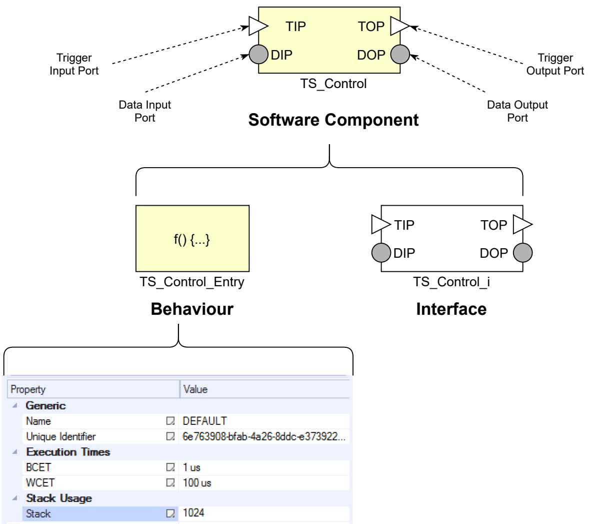

Figure 5: Properties of a Software Component.

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,Since the transition to multi-core platforms in general affects the timing behaviour of the system, 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.For example, a timing requirement can be derived based on the communication between TS_IPL _Path and the TS_IPL _JointPath. Here, one job of the TS_IPL _Path generates data for n jobs of the TS_IPL _JointPath. The next instance of the TS_IPL _Path task should complete its execution before the nth job of the TS_IPL _JointPath is executed. Further, we consider the system to be modelled with cause-effect task chains [30, 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 [33]. 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 can be blocked only for a duration that does not violate this requirement. 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].

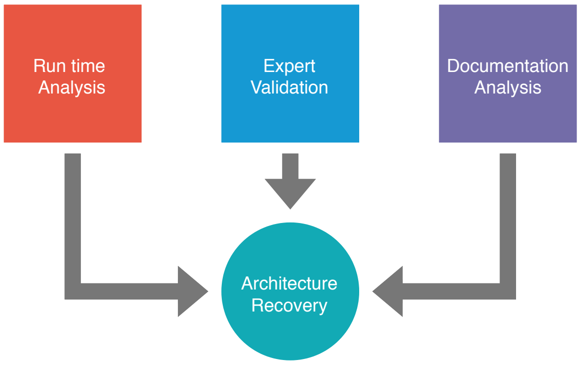

Figure 6: Architecture analysis.

To extract the necessary timing requirements, such as the periodicity, execution times and deadlines, we require analysis of multiple data sources. We identified that the architecture documentation, the run-time execution logs and expert validation of the analysis are essential resources for the architecture recovery phase of the migration process, also shown in Fig. 6.

Documentation Analysis. The architecture of large software intensive systems is normally documented according to the "4+1" architectural view model [34] or an enhanced variant. The format for architecture documentation can vary depending on the internal process and industry-relevant certification requirements. SysML [35] 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 the documentation, it should be possible to identify chains of dependent components, the tasks associated with these components and the expected timing behaviours. The system we considered was documented both in UML models, as well as textual documents.However, during our analysis, we found that existing documentation did not contain any information mapping different tasks to their respective components and there was limited information on expected timing behaviours either unavailable or was incomplete available in the architecture documents, necessitating other analysis approaches such as run-time analysis and expert validation.

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 [36]. 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 [37]. 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 [37], 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 run-time behaviour, it is difficult to identify a configuration that can be a 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. Further, since the controller software operates under different modes, the "maximum load" approach could be pessimistic. Depending on the system under migration, we will need to identify an appropriate configuration and analyse the run time behaviour of each mode independently. For the controller software considered, the "normal operation mode" had the highest resource demand and since all the other modes run only a subset of the "normal operation mode" tasks, we use the maximum load configuration of the "normal operation mode"and ensure that all the required 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. For example, the incorrect configuration resulted in the trace logs showing multiple instances of the jobs of a task as a single job of the same task. 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.

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 [36]. 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. For example, due to incorrect configuration of the code instrumentation library, the periodicity of the TS_RPI observed during run-time analysis phase did not match the values expected by the experts. the functional behaviour however, was accurate, prompting a separate analysis. This analysis identified incorrect configuration of the code instrumentation as the root cause for observed deviation in the periodicity.

5.4. 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. For example, we use scalability to refer to the capability of the controller software to control more than one manipulator on the same hardware platform. 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.

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 [8, 9, 38, 39]. Therefore, a good starting point in this stage are 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 [21] 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 [30].

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. [40], and Nemati et al. [41]. 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 [39]. 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 [42]. In the case of fixed-priority scheduling, priority assignment can impact response times [38].

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 [28] or the Cheddar tool [43], analytical calculations if using techniques such as those identified in [39], or model-checking if using the timed automata approach specified in [44]. For the system described in Section 2, one strategy could be to allocate the parts of the system that are constrained by the timing constraints to one core and rest of the software components to other cores (e.g., TS_IPL_Path, TS_IPL_JointPath, and TS_Control to one core and the rest of the components to the other core(s)). Another strategy could be to allocate the software components to the cores such that the specified age and reaction delays are minimized. Another strategy could be based on precedence constraints between the software components, which should be on the same core (e.g., TS_Web and TS_Sys_Backup have an implicit precedence constraint as the latter is triggered by the former, hence both should be on the same core). Similarly, another allocation strategy could be based on the criticality levels associated to the software components so that non safety-critical software cannot interfere with the safety-critical software as proposed in [45]. We would like to point out that given the safety-critical nature and complexity of the system, we hypothesise 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.

Ranking of the solutions. The ranking step of the transformation phase orders the evaluated solutions in terms of certain criteria. For example, the evaluated solution may be required to adhere to safety and security requirements of the domain. Further it may be possible that the extra-functional properties such as portability between different hardware platforms may be prioritised over performance improvement on a single hardware device. To address such requirements in a systematic manner, 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 propose focus group discussions involving the different domain experts.

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 requiring a trade-off analysis to select a final solution.

5.4.1. 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.

6. 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 a layered architecture among others. 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 synchronisation 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.

6.1. 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 categorise 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. For example, the trajectory generation component may require that its source code be modified to accommodate the changes necessary to migrate to multi-core platform. We then review the solution with the owners of the trajectory generation component. Further, 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. For example, if a new real-time middleware, that will provide a common inter-task communication mechanism is to be implemented, then a new component will be created.

6.2. 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.

7. 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.

7.1. 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. [46]. 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.

7.2. 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, no new failures should be introduced after the migration. 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.

8. 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.

8.1. 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 [47] 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 [25] 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.

8.2. 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

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

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

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.

9. Evaluation

We chose a survey-based approach to evaluate the proposed methodology. We followed the guidelines provided by Kitchenham et al. [48] for survey-based research and the discussion of the results. We begin by describing the design of the survey and then discuss the results of the survey.

9.1. Survey Design

As a first step in the survey-based evaluation, we identified (i) feasibility, (ii) usability and, (iii) usefulness as the evaluation objectives for the migration methodology. Next, we identified the target population for the evaluation to be those organisations that develop complex real-time software systems such as industrial automation systems and construction vehicles. We identified a sample from the target population in a non-probabilistic manner through convenience and judgement based sampling. We created the survey instrument in the form of online questionnaire that included both close and open ended questions. The close ended questions were designed to verify the generalisation of the observations and the applicability of the different steps in the methodology. The open ended questions required the respondents to provide their opinion in a textual format on feasibility and usefulness of the methodology. The complete questionnaire was piloted by requesting colleagues not involved in the study to ensure clarity of language before it was shared with the respondents. The questionnaire was made available digitally and included a brief overview of the purpose of the questionnaire. The respondents were requested to read about the presented methodology before they answered the survey. The received responses were then analysed to evaluate the methodology.

9.1.1. Evaluation Objectives

As previously mentioned, we identified three key objectives for the evaluation, namely feasibility, usability and usefulness of the methodology. For each of these objectives, we adopt the definitions used by Adesola et al. [49] to evaluate their business improvement process methodology. Briefly, we use *feasibility* to imply that all the steps in the methodology can be followed in practice. We use the term *usability* to refer to the ease of applicability of the methodology steps and the tools mentioned therein. We use *usefulness* to refer to the outcome of applying the methodology to relevant systems by an organisation. Furthermore, we also included the objective of validating the possibility of generalising key observations in the methodology.

9.1.2. Target Population and Sampling Strategy

To address the evaluation objectives, the target population was identified as organisations developing complex real-time systems. As for the sample, we identified 2 different departments within the same organisation working on independent and unrelated products and also two other organisations. We then identified 9 expert practitioners from the sample group as the most relevant for the evaluation. The participants were chosen based on their experience in managing and developing software(10+ years) for industrial systems and for background in multi-core technologies and their knowledge of the application domains.

9.1.3. Instrument Design

The survey was designed in the form of a questionnaire, combining nominal, close-ended questions, and the open-ended questions requiring textual input from the respondents. The questionnaire was designed to address two different aspects, (i) problem relevance and (ii) methodology evaluation. For the problem relevance, we developed six questions to verify if the respondents were considering multi-core platforms for their products. The rest of the questionnaire was focused on methodology evaluation. We classified the evaluation related questions as either implicit or explicit. The implicit questions required the respondents to reflect on the overall feasibility, usability and usefulness of the methodology. The explicit questions were designed to validate the generalisation of some of the observations made in the methodology. Table 2 shows the mapping among the different steps of the methodology, the evaluation type for each of the step and the associated question IDs. Appendix A.3 shows the questionnaire.

9.2. Survey Results and Discussion

As mentioned previously, the questionnaire was shared with nine carefully identified participants from the sample population. Of the nine participants invited, five respondents participated in the survey. We use the labels A,B,C,D and E to refer to each of the respondent individually. We discuss the results for the objectives of problem relevance, generalisation, overall feasibility, overall usability and the overall usefulness.

Table 2: Mapping among the different steps of the methodology, the evaluation type for each of the step and the associated question IDs.

Methodology Stage (Step)	**Evaluation Type/ No. Of Questions**	**Question ID**
Architecture Abstraction and Representation (General)	Explicit : 1 Implicit : 6	11 27-32
Architecture Abstraction and Representation (Expert Interviews)	Implicit : 6	27-32
Architecture Abstraction and Representation (State-of-art in Real-time Systems)	Implicit : 6	27-32
Architecture Abstraction and Representation (State-of-art in Model-Driven Engineering)	Implicit : 6	27-32
Architecture recovery (Documentation Analysis)	Explicit : 3 Implicit : 6	13-15 27-32
Architecture recovery (Runtime Analysis)	Explicit : 7 Implicit : 6	12, 16- 21 27-32
Architecture Recovery (Expert Validation)	Implicit : 6	27-32
Architecture Transformation (Identification of Potential Solutions)	Implicit : 6	27-32
Architecture Transformation (Evaluation of the Solutions)	Implicit : 6	27-32
Architecture Transformation (Ranking of the Solutions)	Explicit : 3 Implicit : 6	22- 24 27-32
Architecture Transformation (Selection of the solutions)	Implicit : 6	27-32
Architecture Verification	Implicit : 6	27-32
Implementation Migration (Component Identification and Creation)	Implicit : 6	27-32
Implementation Migration (Implementation)	Implicit : 6	27-32
Verification Migration (Concurrency Testing)	Implicit : 6	27-32
Verification Migration (Migration Validation)	Explicit : 2 Implicit : 6	25-26 27-32
Tools for Migration (Architecture Representation)	Implicit : 6	27-32
Tools for Migration (Architecture Recovery)	Implicit : 6	27-32

Problem Relevance. From the problem relevance perspective, 4 of the 5 the respondents, (A,B,C and E) said that their applications were not designed for multi-core. Respondent *D* said that their applications were designed for multi-core but they have been developed from the scratch with only limited reuse of existing code. Respondents *C* and *E* confirmed that they are planning to migrate to a multi-core platform while the rest of the respondents did not provide any information. Additionally, the same four respondents chose the option of redesigning the application while reusing the existing code over developing the application from scratch. The responses indicate that migration to multi-core platforms is being considered in the industry and at the same time, the respondents prefer reusing the existing code over the development of the applications from scratch.

Generalisation and Feasibility. Since the methodology was developed based on observations of one system, we created the questionnaire to verify if the observations made in different steps can be generalised for other complex real-time software systems as well. This was done by asking directed nominal questions focused on architecture representation, architecture recovery (runtime analysis and documentation), architecture transformation (ranking of solutions), and verification migration. For the architecture representation, the results indicate that only parts of the application can be described by timing properties such as worst-case execution times, periods and deadlines.

Similar to the observations about lack of information in the documentation, 4 of the 5 the respondents, (A,B,C and E) said that the application design was not fully documented. Further, only one respondent said that the timing properties were discussed in the design documentation while the rest of the respondents said that the timing properties of only a few critical parts of the application were discussed in the documentation.

The methodology relies on the presence of diagnostic information such as execution times and periodicity for architecture recovery. All the respondents said that their systems provide such diagnostic information. Furthermore, all the respondents mentioned that their applications had multiple configurations and that the runtime behaviour depended on the configuration. None of the respondents said that they tested all possible configurations but only a few. Four out of five respondents (A, B,C and D) said they tested average-case configurations. Furthermore, respondents A and E said that they test the worst-case configurations while respondent D said that they test the best-case, average-case as well as the worst-case configurations. This indicates that identifying a representative configuration for architecture is not straight forward and can depend on individual application requirements.

Evaluation and ranking of solutions is an important step in the methodology. Here we assumed that it will be possible to identify and provide measurable metrics for ranking possible multi-core solutions. To verify if the assumptions are valid, the respondents were explicitly asked if they can provide measurable parameters and also prioritise them. Four out of five respondents (A, B D and E) agreed that they can define as well as prioritise, while respondent C answered negatively.

For the verification migration stage of the methodology, a key assumption is that the complex real-time systems such as the one discussed in this paper have a robust testing mechanism in place for verifying functional correctness. All the respondents agreed that they do have such a mechanism in place. Further, all respondents agreed that they will reuse the existing tests to verify the behaviour of the systems after migration, which is consistent with the assumptions made in the proposed methodology.

The results of the questionnaire so far indicate that much of the observations can be generalised to other complex real-time systems. One key observation however, is that describing all of the application components with timing properties may not be possible. For the steps not discussed in generalisation, we address them from the overall feasibility perspective discussed next.

Overall Feasibility. In order to validate the feasibility of the methodology, i.e., to verify if all the steps of the methodology can be followed, the respondents were asked to answer if they found the methodology feasible and to describe the rationale behind their choice. Four out of five respondents (A B D and E) considered the methodology to be feasible while respondent C considered otherwise. When describing the rationale, respondent C said that they needed more information and the correct answer would actually be that they are not sure. Respondents B and E did not explain the rationale. Respondent A and D agreed that it is possible to represent the architecture at a feasible abstraction level and that the methodology covered all the critical steps. One concern however was that the industrial applications are rather big, and therefore we need to address the migration in parts and avoid a "big bang" approach.

Overall Usability. The survey also included questions to evaluate the overall usability of the methodology, i.e., to verify if the steps in the methodology are workable and are easy to apply in practice. Similar to the question of feasibility, four out of five respondents (A B D and E) answered positively while respondent C said no. When describing the rationale, respondent C said that their correct answer would actually be that they are not sure. Respondent A and B said that the transformation phase was uncertain but the steps are general enough to be followed and that the difficulty in following the steps may depend on the "architecture, requirements and availability of tools". Similar response was provided by respondent D who said that the level of modelling may vary depending on the company. Based on the responses it can be observed that the steps in the proposed methodology can be followed in general but the overall usability is dependent on individual applications.

Overall Usefulness. Another objective of the evaluation is to assess overall usefulness of the methodology for the target population. To address this, the respondents were asked to evaluate "Usefulness: if the methodology can produce results that the organisation will find useful?". Two out of five respondents (A and B) consider the methodology to be useful for the industry, whereas the remaining three respondents consider the methodology to be "partially" useful. Respondent B justified their choice by highlighting the general applicability of the steps and respondent A said that having such a methodology will create a "common understanding" between the different stakeholders and the developers, thus *increasing the possibility of success and decreasing risks*. Respondent C said the it may not be possible to follow the steps completely, but the ideas can be "useful". A similar observation was made by respondent D who said it will be necessary to consider the product to see if the methodology fits the product being considered for migration. Although it is not possible to draw a straight forward conclusion about the usefulness of the methodology, we can observe from the responses that having a methodology can reduce the risks of migration projects but the methodology will have to be adapted to suit individual application needs in the industry.

Discussion. The proposed methodology was evaluated for feasibility, usability and usefulness by expert practitioners via a questionnaire. From the feasibility perspective, the analysis of the questionnaire responses indicate that the methodology covers the critical steps necessary for a software migration. From the usability perspective, the analysis of the responses shows that the different steps can be applied in practice but depending on the application, the abstraction level and the modelling requirements will depend on individual applications. From the usefulness perspective, the responses show that following the methodology steps can decrease the risks associated with the migration. From the Generalisation perspective, the response show that the observations made in the methodology can be extended to systems other than the robotic system considered, while highlighting the fact that it may not always be possible to describe the timing properties for all of the application components.

Threats to Validity. Since the evaluation of the methodology has been carried out using a survey, we include a discussion on the validity of the results. Kitchenham et al. [48] advocates that a survey is reliable if it has been administered multiple times and if we get similar results each time. In our case, the survey was administered only once. This implies that the results may vary if the respondents were to answer questionnaire at different times. However, much of the questionnaire had nominal questions and the number of options provided were binary but with an additional option to provide textual information thereby limiting the possibility of variability in the responses. Furthermore, although the sample group was carefully chosen in a non-probabilistic manner, it is possible that a different sample of respondents may have provided different responses, affecting the validity of the conclusions drawn from the survey results. While the survey included questions relating to generalisation of the observations, not all of the methodology steps were explicitly considered but were included under the general questions of overall feasibility, usability and usefulness. Explicit questions may have lead to a different conclusion from the one discussed in the paper.

10. Conclusion

Migration of complex 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. We used task-level abstraction of the system to drive the transformation and associated timing properties to task-level models and proposed their use as input for the evaluation of 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. We evaluated feasibility, usability and usefulness of the methodology using a survey-based approach. Majority of the respondents agreed that the methodology is feasible, usable and useful in general for the industrial applications. The evaluation also revealed that the methodology will have to be individually adapted to each system under migration.

Acknowledgements

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 projects DESTINE and PROVIDENT, and the Swedish Knowledge Foundation (KKS) through the projects HERO, FIESTA and DPAC. We would also like to thank our industrial partners for providing valuable feedback via the survey.

References

[1] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, W.-G. Tan, Types of software evolution and software maintenance, Journal of Software Maintenance 13 (1) (2001) 3–30.

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

[5] A. Avizienis, J. . Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (Jan 2004).

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

[7] S. Mubeen, E. Lisova, A. V. Feljan, Timing predictability and security in safety-critical industrial cyber-physical systems: A position paper, Applied Sciences 10 (9) (2020) 3125.

[8] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, R. I. Davis, A survey of timing verification techniques for multi-core real-time systems, ACM Comput. Surv. 52 (3) (2019) 56:1–56:38.

[9] R. I. Davis, L. Cucu-Grosjean, A survey of probabilistic schedulability analysis techniques for real-time systems, LITES 6 (1) (2019) 04:1–04:53.

[10] S. M. Salman, A. V. Papadopoulos, S. Mubeen, T. Nolte, A systematic migration methodology for complex real-time software systems, in: 2020 IEEE 23rd IEEE International Symposium on Real-Time Distributed Computing, pp. 192–200.

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

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

[13] P. Church, H. Mueller, C. Ryan, S. V. Gogouvitis, A. Goscinski, Z. Tari, Migration of a SCADA system to IaaS clouds – a case study, Journal of Cloud Computing 6 (1) (2017) 256.

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

[15] H. M. Sneed, Planning the reengineering of legacy systems, IEEE Software 12 (1) (1995) 24–34.

[16] Ravi Erraguntla, Doris L. Carver, Migration of sequential systems to parallel environments by reverse engineering, Information & Software Technology 40 (7) (1998) 369–380.

[17] M. Battaglia, G. Savoia, 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.

[18] 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, T. Varvarigou, Software modernization and cloudification using the ARTIST migration methodology and framework, Scalable Computing: Practice and Experience 15 (2) (2014).

[19] L. Forite, 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.

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

[21] M. Stigge, W. Yi, Graph-based models for real-time workload: a survey, Real-Time Systems 51 (5) (2015) 602–636.

[22] F.Herrera, H. Posadas, P.Peñil, E.Villar, F.Ferrero, R.Valencia, G.Palermo, The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems, Journal of Systems Architecture 60 (1) (2014) 55–78.

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

[24] S. Mubeen, H. Lawson, J. Lundbäck, M. Gålnander, 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.

[25] K. G. Larsen, P. Pettersson, W. Yi, Uppaal in a nutshell, Int. J. Softw. Tools Technol. Transf. 1 (1-2) (1997) 134–152.

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

[27] The AUTOSAR Consortium, Autosar technical overview, in: Version 4.3., 2016, `http://autosar.org`.

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

[29] L. Lo Bello, R. Mariani, S. Mubeen, S. Saponara, Recent advances and trends in on-board embedded and networked automotive systems, IEEE Transactions on Industrial Informatics (2019).

[30] M. Becker, D. Dasari, S. Mubeen, M. Behnam, T. Nolte, End-to-end timing analysis of cause-effect chains in automotive embedded systems, Journal of Systems Architecture 80 (2017) 104 – 113.

[31] M. Becker, S. Mubeen, D. Dasari, M. Behnam, T. Nolte, A generic framework facilitating early analysis of data propagation delays in multi-rate systems (invited paper), in: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017, pp. 1–11.

[32] S. Mubeen, T. Nolte, M. Sjödin, J. Lundbäck, K.-L. Lundbäck, Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints, Software & Systems Modeling 18 (1) (2019) 39–69.

[33] T. A. Henzinger, B. Horowitz, C. M. Kirsch, Giotto: a time-triggered language for embedded programming, Proceedings of the IEEE 91 (1) (2003) 84–99.

[34] P. B. Kruchten, The 4+1 View Model of architecture, IEEE Software 12 (6) (1995) 42–50.

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

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

[37] J. Kraft, A. Wall, 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), 2010.

[38] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, A. Burns, A review of priority assignment in real-time systems, Journal of Systems Architecture 65 (2016) 64–82.

[39] B. B. Brandenburg, M. Gul, Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations, in: IEEE Real-Time Sys-

tems Symposium, 2016.
[40] G. F. H. Macher, Andrea Höller, Eric Armengaud, C. J. Kreiner, Automotive Embedded Software: Migration Challenges to Multi-Core Computing Platforms, in: Proceedings INDIN 2015, 2015, pp. 110–118.
[41] F. Nemati, M. Behnam, T. Nolte, Efficiently migrating real-time systems to multi-cores, in: 2009 IEEE Conference on Emerging Technologies & Factory Automation, pp. 1–8.
[42] S. A. Asadollah, H. Hansson, D. Sundmark, 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.
[43] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Cheddar: a flexible real time scheduling framework, in: SIGAda, ACM, 2004, pp. 1–8.
[44] C. Norstrom, A. Wall, 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.
[45] A. Bucaioni, S. Mubeen, F. Ciccozzi, A. Cicchetti, M. Sjödin, Modelling multi-criticality vehicular software systems: evolution of an industrial component model, International Journal on Software and Systems Modeling 19 (2020) 1283–1302.
URL http://www.es.mdh.se/publications/5787-
[46] F. A. Bianchi, A. Margara, M. Pezzè, A survey of recent trends in testing concurrent software systems, IEEE Transactions on Software Engineering 44 (8) (2018) 747–783.
[47] The UML profile for MARTE: Modeling and analysis of real-time and embedded systems., OMG Group, 2010.
[48] B. A. Kitchenham, S. L. Pfleeger, Personal opinion surveys, in: Guide to Advanced Empirical Software Engineering, Springer London, London, 2008, pp. 63–92.
[49] Adesola Sola, Baines Tim, Developing and evaluating a methodology for business process improvement, Business Process Management Journal 11 (1) (2005) 37–46.

Appendix A.

Table A.3 summarises the survey questionnaire and shows the mapping between the questions and the different stages of the methodology.

. Alessandro Vittorio Papadopoulos is an Associate Professor of Computer Science at Mälardalen University, Västerås, Sweden. He is part of the SPEC Research Group on cloud computing, with a leading role in the definition of methodologies for benchmarking and performance evaluation in cloud deployments. He was elevated to Senior Member of IEEE in 2019. He received his B.Sc. and M.Sc. (summa cum laude) degrees in Computer Engineering from the Politecnico di Milano, Milan, Italy, and his Ph.D. (Hons.) degree in Information Technology from the Politecnico di Milano, in 2013. From 2014 to 2016, he was a Post-Doctoral Researcher with the Department of Automatic Control, Lund University, Lund, Sweden, and he was also a member of the Lund Center for Control of Complex Engineering Systems, Linnaeus Center, Lund University. He was a Postdoctoral Research Assistant at the Dipartimento di Elettronica, Informazione e Bioingegneria at the Politecnico di Milano (2016). His research interests include robotics, control theory, real-time and embedded systems, and autonomic computing.

. Thomas Nolte: Professor Nolte is leading the Complex Real-time Embedded Systems research group at Mälardalen University, Sweden. He is a senior member of IEEE. He was awarded a B.Eng., an M.Sc., a Licentiate, and a Ph.D. degree in Computer Engineering from the same University in 2001, 2002, 2003, and 2006, respectively. He has been a Visiting Researcher at University of California, Irvine (UCI), Los Angeles, USA, in 2002, and a Visiting Researcher at University of Catania, Italy, in 2005. He has been a Postdoctoral Researcher at University of Catania in 2006, and at Mälardalen University in 2006-2007. He has co-authored over 300 research publications in peer-reviewed conferences, workshops, books and journals.

. Saad Mubeen is an Associate Professor at Mälardalen University, Sweden. He has previously worked in the vehicle industry as a Senior Software Engineer at Arcticus Systems and as a Consultant for Volvo Construction Equipment, Sweden. He is a Senior Member of IEEE and a Co-chair of the Subcommittee on In-vehicle Embedded Systems within the IEEE IES Technical Committee on Factory Automation. His research focus is

on model- and component-based development of predictable embedded software, modeling and timing analysis of in-vehicle communication, and end-to-end timing analysis of distributed embedded systems. Within this context, he has co-authored over 135 publications in peer-reviewed international journals, conferences and workshops. He has received several awards, including the IEEE Software Best Paper Award in 2017. He is a PC member and referee for several international conferences and journals respectively. He is a guest editor of IEEE Transactions on Industrial Informatics (TII), Elsevier's Journal of Systems Architecture and Microprocessors and Microsystems, ACM SIGBED Review, and Springer's Computing journal. He has organized and chaired several special sessions and workshops at the international conferences such as IEEE's IECON, ICIT and ETFA. For more information see $http : //www.es.mdh.se/staff/280 - Saad_{M}ubeen$.

Table A.3

No.	Methodology Stage/ Purpose	Question
1	Participant information	Before you proceed, please take the time to read the paper describing the methodology.
2	Participant information	Name of the organization:
3	Participant Relevance	Does your application have real-time components?
4	Participant Relevance	Is your application designed to run on multi-core platforms?
5	Participant Relevance	Have you in the past, migrated your application to a multi-core platform?
6	Participant Relevance	Are you considering migrating the application to a multi-core platform?
7	Exploratory	Did you follow any specific methodology or guidelines to migrate the application to a multi-core platform?
8	Exploratory	Will you recommend the existing approach to others?
9	Exploratory	If you'd like to provide more information about the used methodology, please do so here.
10	Problem relevance	Select the preferred option : a) I prefer to redesign and redevelop the application from scratch for a multi-core platform. b) I prefer to redesign but also reuse the existing code for a multi-core platform.
11	Architecture Abstraction and Representation	Can your application be described with timing properties such as "worst case execution times", "period", "deadlines"?
12	Architecture Recovery (runtime analysis)	Is it possible to identify a particular build of the application that can be used to recover the timing requirements of the application and can such timing requirements be used to create a model of the application?
13	Architecture recovery (documentation)	Is the design of your application documented?
14	Architecture recovery (documentation)	Does the application design documentation contain timing properties?
15	Architecture recovery (documentation)	The behaviour of your application can be:(choose one) a) accurately inferred from the design documentation b) cannot be accurately inferred from the design documentation
16	Architecture Recovery (runtime analysis)	Does your application provide diagnostic logs of runtime behaviour?
17	Architecture Recovery (runtime analysis)	The code instrumentation : a) is fully reliable. b) may not be fully reliable.
18	Architecture Recovery (runtime analysis)	Does your application have multiple configurations?
19	Architecture Recovery (runtime analysis)	Does the runtime behaviour of the application depend on the configuration?
20	Architecture Recovery (runtime analysis)	Do you test all possible configurations of the applications?
21	Architecture Recovery (runtime analysis)	Which configuration do you test
22	Architecture Transformation (Ranking of solutions)	Do you have any existing process/guidelines in place to evaluate and choose between different solutions that may be specific to multi-core platforms?
23	Architecture Transformation (Ranking of solutions)	Is it possible to define measurable parameters that will suit your application's timing requirements to choose one solution over the other?
24	Architecture Transformation (Ranking of solutions)	Is it possible to prioritize the measurable parameters that will suit your application requirements to choose one solution over the other?
25	Verification Migration	Does your application have a verification and validation process in place for checking functional correctness?
26	Verification Migration	Will you reuse the existing tests to verify the behaviour on multi-core platforms?
27	Feasibility	Feasibility: Can the methodology described be followed?
28	Feasibility	Please briefly describe the reason behind your answer here:
29	Usability	Usability: Is the methodology workable? Are the steps and tools easy to use and apply?
30	Usability	Please briefly describe the reason behind your answer here:
31	Usefulness	Usefulness: Is the methodology worth following? Does the methodology produce results that the business will find helpful?
32	Usefulness	Please briefly describe the reason behind your answer here:
33	Overall comments	Which part of the methodology will you like to improve? (you can choose multiple options)
34	Overall comments	Please provide any suggestions and improvements you want to see in the methodology here: