Legacy Software Migration based on Timing Contract aware Real-Time Execution Environments

Irune Yarza[a,*], Mikel Azkarate-askatsua[a], Peio Onaindia[a], Kim Grüttner[b], Philipp Ittershagen[b], Wolfgang Nebel[c]

[a] *Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Pº J.M. Arizmendiarrieta, 2, 20500 Arrasate-Mondragón, Spain*
[b] *OFFIS - Institute for Information Technology, Eschwerweg 2, 26121 Oldenburg, Germany*
[c] *C.v.O. Universität Oldenburg, Ammerländer Heerstr. 114-118, 26121 Oldenburg, Germany*

Abstract

The evolution to next generation embedded systems is shortening the obsolescence period of the underlying hardware. As this happens, software designed for those platforms (a.k.a., legacy code), that might be functionally correct and validated code, may be lost in the architecture and peripheral change unless a retargeting approach is applied. Embedded systems often have real-time computing constraints, therefore, the legacy code retargeting issue directly affects real-time systems. When dealing with real-time legacy code migration, the timing as well as the functional behaviour must be preserved. This article sets the focus on the timing issue, providing a migration path to real-time legacy embedded control applications by integrating a portable timing enforcement mechanism into a machine-adaptable binary translation tool. The proposed timing enforcement solution provides at the same time means for validating the legacy timing behaviour on the new hardware platform using formal timing specifications in the form of contracts.

Keywords: legacy software, retargeting, real-time systems, time contract

1. Introduction

Companies within the embedded systems industry are facing a relentless demand for increasingly stringent requirements such as better performance, increased dependability, and energy efficiency, while offering a cost-effective product within a reduced time-to-market. This transition to next generation embedded systems is being encouraged by the rapid development of computing architectures. As a consequence, the obsolescence period of embedded systems is being shortened and there is a need to deal with legacy systems and their integration.

Legacy systems are characterized by some particular properties:

- Usually runs on obsolete hardware which is slow and expensive to maintain (Wu et al., 1997).
- Use customized and deprecated toolchain(s) (Wagner and Wagner, 2014).
- Have no or outdated documentation and original developers or users are no longer available (Wagner and Wagner, 2014).
- Are essential for the company (Bennett, 1995) since they comprise business knowledge (Wahler et al., 2015).

Due to their nature and particular properties, legacy systems present a complex scenario in software maintenance and evolution. Hence, the process of updating legacy systems is usually complex, error-prone, time-consuming and requires high cost investment.

Binary translation appears to be a standard approach when it comes to legacy software migration, since the binary that runs on the legacy hardware can be ported to a new hardware platform without a considerable expense of time, effort and money. Software recompilation is also a well known ap-

*Corresponding author
Email address: iyarza@ikerlan.es (Irune Yarza)

proach to port platform-independent legacy source code.

However, when dealing with Real-Time (RT) legacy code migration, not just the functional behaviour, but also the timing behaviour must be preserved. To the authors knowledge, limited solutions exist to port real-time legacy software, while existing solutions have limitations regarding their portability. Therefore, industry still needs a low-overhead embedded RT legacy software retargeting solution that can be easily ported to different source and target architectures.

In the direction to solve this problem, this work sets the focus on the timing issue, therefore, the overall goal of this research is to provide a migration path to real-time legacy embedded control applications by integrating a portable timing enforcement mechanism into a machine-adaptable binary translation tool. The proposed solution should also provide means to validate the legacy timing behaviour on the new hardware platform.

As a first step on this research, Yarza et al. (2020) studies the feasibility of two machine-adaptable binary translators, one dynamic and the other one static, for their use in a RT property conserving legacy software migration process. From this feasibility study, the static approach is selected to implement a timing contract aware real-time legacy software migration solution. The main contribution of this article are:

- The systematic annotation of legacy timing properties into the behavioural legacy code using a set of portable temporal construct that provide means to enforce a specific timing behaviour within the legacy software.
- The systematic transformation of legacy timing properties into formal timing specifications for their latter use within the timing validation phase.
- The integration of the temporal constructs within the binary translation process to achieve a timing-aware legacy software migration.

The remainder of this paper is organized as follows. An overview of related work in the area of timing-aware recompilation and machine-adaptable binary translation techniques is provided in Section 2. Then, in Section 3 the proposed migration path is constraint to a specific class of application. Then, based on these constraints, Section 6 presents the RT legacy software migration path. The proposed solution is then assessed in Section 7 and obtained results are analysed. Finally, Section 8 gives a conclusion and outlook on future work.

2. Related work

Given that legacy software migration is a common issue in industry, it has been widely studied during the last decades. However, when porting RT legacy software, not just the functional behaviour, but also the timing behaviour must be preserved. This section provides an overview of existing solutions for a timing-aware recompilation of legacy C source code, as well as binary translation tools targeting either a machine-adaptable or a RT legacy code migration solution. The related work identifies six timing-aware recompilation solutions (from R1 to R6) and three binary translation tools (from B1 to B3) for a latter analysis.

2.1. Timing-aware Recompilation

Software recompilation is a well known solution when it comes to port legacy software to a new Instruction Set Architecture (ISA). However, recompilation to be applicable on the legacy migration process, the legacy source code must be available and it must be at sufficient high level that it is independent (or almost independent) from the legacy hardware platform (e.g., dedicated instructions, specific hardware resources). In the following, timing-aware recompilation solutions are covered:

Resmerita et al. (2015) proposed a systematic approach to apply real-time programming to legacy embedded control systems composed of time- as well as event-triggered tasks with different priority levels. Using Timing Definition Language (TDL) the code is transformed and then compiled into E-code and interpreted at runtime by an Embedded Machine (E-machine) (Henzinger and Kirsch, 2007), which provides a real-time interaction among software and physical processes. There are several implementations of the E-machine, being one of them in C under Linux using Portable Operating System Interface (POSIX) threads and semaphores.

Le Nabec et al. (2016) describe the process of modelling RT legacy software using the Real-Time BIP (RT-BIP) (Abdellatif et al., 2013) framework, an extension to the Behaviour Interaction Priority (BIP) language for modelling real-time systems

together with a real-time engine used for execution. To this end, they define a configurable component pattern, called the Real-Time BIP Agent (RT-BIPAgent), that follows a classical template for a real-time task, composed of a start time, a period, a set of input and output data, and a computational function. then, based on the FreeRTOS platform, executable code is generated from the BIP model.

Natarajan and Broman (2018) proposed an extension of the C programming language, called Timed C, consisting of a set of primitives for defining soft and firm RT constants that ease RT systems' implementation. A Timed C file is then compiled into a target specific C file using their source-to-source compiler, KTC. The resulting file is linked against POSIX or FreeRTOS to implement the user defined timing and scheduling behaviour.

Real-Time Concurrent C (Gehani and Ramamritham, 1991) incorporates a set of temporal constructs into Concurrent C, a parallel superset of C. It provides means to specify strict timing constraints through the temporal constructs, which allow delaying program's execution, defining periodicity or specifying deadlines. Real-Time Concurrent C was designed for a UNIX-based implementation of Concurrent C and its compiler is no longer available (Natarajan and Broman, 2018).

The time measurement and control blocks (Bruns et al., 2019) are a C++ extension, implemented as a C++ library, that enable block-level timing annotations into embedded C++ software. These block annotations can be implemented to measure and profile the execution time of a given software block as well as to control and enforce a specific timing behaviour (time-budget or specific duration) at run-time. Time measurement and control blocks have been implemented for bare-metal C++ applications running on an Zynq-7000, using the Global Timer Counter and a Central Processing Unit (CPU) Watchdog.

The WCET-aware C Compiler (WCC) (Falk and Lokuciejewski, 2010) was the first compiler to provide means to reduce the Worst Case Execution Time (WCET) at both, source code and assembly code level. The WCC has a clear notion of the program's worst-case behaviour (combining measurement-based WCET analysis and static program analyses) and applies specialized compiler optimization to reduce the program's WCET. The WCC's target architecture is the Infineon TriCore processor heavily used in the automotive industry.

2.2. Binary Translation

Binary translation techniques have been widely studied and developed in the last decades. So, given the great amount of binary translation systems and the focus of this paper on embedded RT legacy software migration, just cross-platform translators heeding portability and/or RT applications will be considered in this section.

The TIBBIT project (Cogswell and Segall, 1995), developed the first binary translation approach for embedded RT applications (which are not assumed to be user-level processes) that needed to be migrated to a different processor but still maintaining the externally observable timing behaviour. To this end, both, the legacy application and the operating system code, are packed in a black box, converted into an equivalent C program, and then translated into the target binary format using the GNU Compiler Collection (GCC) retargetable compiler. During the translation process, timing code is inserted into each translation block, to maintain the same timing behaviour as in the original processor. If the execution is running faster than it did in the old processor, extra time is used to run other tasks until the execution is back on schedule.

Then, in 2008, Heinz (2008) proposed a system-level Static Binary Translation (SBT) approach to port RT legacy software. However, instead of dynamically computing a delay, as Cowsgell and Zary did on the TIBBIT project, the delay computation is shifted from run-time to compile-time. The translator selects from a set of precomputed delays the appropriate value according to the context of a program point, so there is no need to keep track of the execution time on the source machine.

UQBT (Cifuentes and Emmerik, 2000) was the first SBT tool designed with portability in mind. UQBT translates the user-level target binary into a machine-independent Intermediate Representation (IR), Higher-Level Register Transfer Language (HRTL), and then the intermediate code is translated into host machine binary code. This two phase translation, eases the portability to new source and target architectures. To handle indirect calls that could not be discovered at static time, the UQBT uses an interpreter.

Based on the UQBT, Ung and Cifuentes (2000) presented the first retargetable Dynamic Binary Translation (DBT) approach, UQDBT. Just like its predecessor, UQDBT does not support system-level emulation and provides a machine-adaptable

solution by separating the system into machine-dependent and machine-independent parts through a machine independent intermediate representation (I-RTL). However, the machine adaptability of the translator comes at the cost of performance. To improve generated code, UQDBT performs generic hot path optimizations that are applicable on different machines.

Since UQBT and UQDBT, there have been a wide variety of machine-adaptable dynamic as well as static Binary Translation (BT) tools. In 2005, Bellard (2005) developed Quick EMUlator (QEMU), a well known machine emulator build upon a fast and portable DBT system. QEMU uses Tiny Code Generator (TCG) to translate target source code into a machine independent IR and then translates IRs into host machine code. In order to reduce system overhead, QEMU applies Translation Block (TB) (unit of a basic block in QEMU) chaining, which directly jumps to the next TB without returning control to the execution engine.

DisIRer (Hwang et al., 2010), is a multi-target SBT tool that leverages the GCC infrastructure. DisIRer translates x86 user-mode instructions into GCC's Register-Transfer Level (RTL) and then translates RTL into GCC's Abstract Syntax Tree (AST). The fact that it is build upon the GCC optimizer and back-end makes the tool cost effective and easily adaptable to multiple targets (those supported by GCC).

Another machine-adaptable BT tool is CrossBit (Yang et al., 2010), which can dynamically translate user-level binaries from different source ISAs into binaries hosted by the same Operating System (OS) for different target architectures. This tool applies profiling information to determine the hot code where host-independent optimizations are applied. Moreover, just as QEMU does, CrossBit applies Basic Block chaining.

LLBT (Shen et al., 2012) is a multi-target SBT tool for embedded systems based on the Low Level Virtual Machine (LLVM) (Lattner and Adve, 2004) compilation framework that provides a source and target independent optimizer, as well as code generation support for multiple ISAs. Therefore, the LLVM compiler infrastructure provides LLBT with means for optimization and retargetability. Moreover, in order to make the system suitable for embedded systems, the size of the address mapping table was reduced.

Based on BT techniques, Rev.ng (Federico et al., 2017) is a machine-adaptable binary analysis framework that relies on QEMU (Bellard, 2005) and LLVM (Lattner and Adve, 2004) to perform the binary translation. Rev.ng takes advantage of the core element of QEMU, TCG, to translate user-level instructions of a supported ISA into a machine independent IR. Then, instead of generating machine code for the host architecture in emulation mode, QEMU IR is further translated into a higher level IR, LLVM IR. By employing LLVM as a back-end, the generated LLVM IR is translated into host machine code.

2.3. Analysis

Based on the literature review, existing solutions in the area of timing-aware recompilation and binary translation are mapped to the scope of this work. The related work identifies six timing-aware recompilation solutions (from R1 to R6) and three binary translation tools (from B1 to B3). Then, this identifiers are placed in the scope map according to the research are they cover. The following list shows the related work that has been mapped to the scope:

- ***R1*** Logical Execution Time (LET) & E-machine (Resmerita et al., 2015)
- ***R2*** BIP & FreeRTOS (Le Nabec et al., 2016)
- ***R3*** Timed C (Natarajan and Broman, 2018)
- ***R4*** Real-Time Concurrent C (Gehani and Ramamritham, 1991)
- ***R5*** Time Measurement and Control Blocks (Bruns et al., 2019)
- ***R6*** WCC (Falk and Lokuciejewski, 2010)

- ***B1*** TIBBIT (Cogswell and Segall, 1995)
- ***B2*** Heinz (Heinz, 2008)
- ***B3*** UQBT (Cifuentes and Emmerik, 2000) & UQDBT (Ung and Cifuentes, 2000)
  QEMU (Bellard, 2005)
  DisIRer (Hwang et al., 2010)
  CrossBit (Yang et al., 2010)
  Rev.ng (Federico et al., 2017)
  LLBT (Shen et al., 2012)

Figure 1 shows the diagram resulting from mapping related work to the scope, which is composed of four research areas: binary translation, where machine-adaptable solutions form a sub-area of research in binary translation; RT software, where RT legacy software is a sub-area in this group; timing enforcement that forms another research area

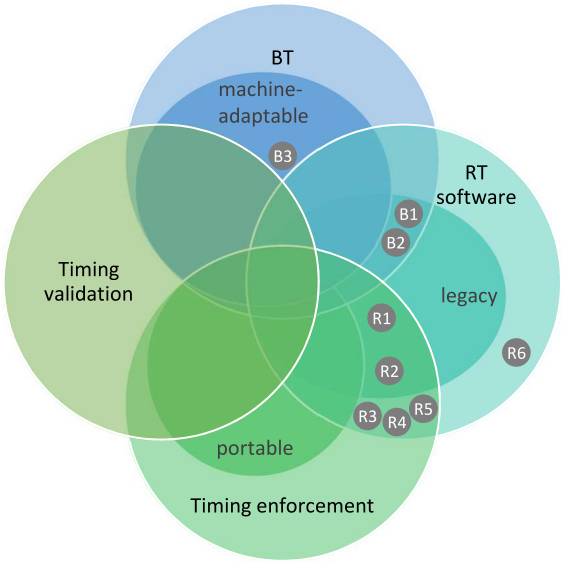


Figure 1: Related work analysis. Mapping related work to the scope.

with retargetable timing enforcement solutions as a subgroup of it; and timing validation which is the fourth research area covered on this research work.

On the one hand, among the timing-aware recompilation solutions, ***R1*** and ***R2*** provide means to enforce a specific timing behaviour within RT software. Both of these solutions describe how they can be implemented on RT legacy software. ***R3***, ***R4***, and ***R5*** are also solution to enforce a specific timing behaviour. However, none of them describe how they can be integrated on RT legacy software. Nevertheless, none of the timing enforcement solutions presents a retargetable solution.

On the other hand, regarding BT tools, ***B1*** and ***B2*** are the only solutions that provide a migration path to RT legacy software. However, none of these translators is machine-adaptable. In contrast, many machine-adaptable BT tools exist (see ***B3***), but none of them supports a timing-aware binary translation, where not just the functional behaviour, but also the timing behaviour needs to be preserved during the translation process.

3. Real-time Legacy System Model Definition

The RT legacy control system is a computer system that executes a set of periodic tasks according to a predefined static scheduling policy. The following subsections describe through formal notation the main modelling elements in the considered RT legacy system.

3.1. Application Model

Table 1 shows an example set of tasks with the corresponding timing properties. Such a task set is described through the application model.

Definition 1. *(Application Model)* The legacy application A is composed of a set of periodic tasks T, where a task t_i can be represented by a tuple (p_i, ϕ_i, e_i, d_i, cr_i), where p_i is the period of the task, ϕ_i specifies a release time (as an offset relative to the start of the period), e_i is an upper bound of the execution time of the task (the WCET of the task can be used as this upper bound), d_i is the relative deadline of the task and cr_i is an identifier of the existence of a section of critical code within the task. A critical code section is set to be a section that generates an exchange of information among state machines (either hardware or software). Every element in the tuple, except for cr_i, is composed of the value v and the corresponding time unit tu.

3.2. Execution Model

The set of periodic tasks T is executed according to a predefined static schedule. Figure 2 depicts the execution trace of an example set of tasks.

Definition 2. *(Execution Model)* The execution E consists of a hyper-period H, which determines the time after which the task execution patter repeats itself, that is in turn composed of a set of frames $\{f_j\}$. The size of the hyper-period H is determined through the Least Common Multiple (LCM) among all tasks' period, $H = lcm(p_i)$ and the frame size F is determined through the Greatest Common Divisor (GCD) among all tasks' period, $F = gcd(p_i)$. Both, $H$and $F$are composed of the value v and the corresponding time unit tu. According to the hyper-period and frame size, the number of frames within a hyper-period is limited to: $max(j) = H/F$.

Each frame f_j is characterized with a set of time slots $\{sl_{j,1}, sl_{j,2}, \ldots, sl_{j,n}\}$, describing each time slot $sl_{j,k}$ as a tuple ($t_{j,k}$, $s_{j,k}$, $e_{j,k}$), where $t_{j,k}$ is the task mapped to the slot, $s_{j,k}$ is the start instant of $sl_{j,k}$, and $e_{j,k}$ the end instant of $sl_{j,k}$. Time slots are consecutively ordered so that $\forall k < n : e_{j,k} \leq s_{j,k+1}$.

The function $\alpha : t_i \rightarrow sl_{j,k}$ maps tasks to slots. A task can only be mapped to one slot $sl_{j,k}$ within

a frame f_j. However, a task can be mapped to the same slots within different frames. For example, task t_1 can be mapped to $sl_{1,2}$, $sl_{2,2}$ and $sl_{3,2}$, but never to $sl_{1,2}$ and $sl_{1,3}$, since a task can only run once in each frame.

When mapping tasks to slots, it is assumed that if a precedence relation exists among two tasks $t_i, t_l \in T$, such that t_l shares the results produced by t_i, then t_l will never start before t_i has finished execution: $\forall(t_i, t_l) \in T : \alpha(t_i).e_j < \alpha(t_l).s_j$

3.3. Example Application

For a better understanding of the presented model, consider an illustrative example that resembles the typical pattern of reactive control systems. The legacy system consists of seven tasks $t_i, i = 1, ..., 7$, where a task is a sequential code block that starts reading input data and its internal state and terminates when it provides the computed results and updates its internal state. Tasks are ordered considering precedence relation and data sharing among them. Through the use of timers and internal counters, these tasks run periodically following a static scheduling policy. Tasks t_1, t_2, t_5, t_6 and t_7 have a period of 20 ms, whereas tasks t_3 and t_4 have a period of 40 ms. Moreover, tasks t_1, t_3, t_4 and t_6 are critical sections. Therefore, in order to preserve correctness of the entire system's behaviour, the time interval at which these critical sections run must be kept equivalent on the migration process (same offset with respect to the start of the period as in the legacy system and minimum jitter among subsequent task instances). The offset of tasks t_1, t_3, t_4 and t_6 are $0, 10, 10$ and 15 ms respectively. On the contrary, for tasks t_2, t_5 and t_7, a variation in their offset and jitter among subsequent task instances does not hinder a correct behaviour of the overall system. However, precedence relation and data sharing among tasks must still be considered. Table 1 summarizes this information and completes it with the WCET of each task. Whereas, Figure 2 depicts the execution of the example task set.

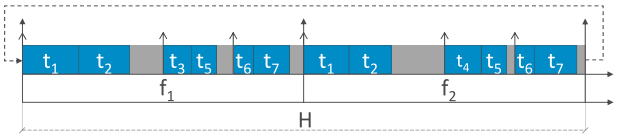


Figure 2: Example execution trace of the RT legacy application example. Execution is composed of two frames that compose the hyper-period.

Table 1: RT legacy application example. Tasks with their corresponding timing information, such as period, offset (if relevant, otherwise N/R is shown), an upper bound for the execution time, as well as identification of critical sections are shown.

Tasks	**Period** $[ms]$	**Offset** $[ms]$	**WCET** $[ms]$	**Critical section**
t_1	20	0	5	✓
t_2	20	N/R	5	-
t_3	40	10	3	✓
t_4	40	10	3	✓
t_5	20	N/R	2	-
t_6	20	15	2	✓
t_7	20	N/R	3	-

4. Time Measurement & Control Blocks

The proposed time measurement and control solution is based on a block-level source code (systematic) annotation approach.

4.1. Time Measurement Block

To measure the end-to-end duration of a code section a time measurement block has been defined, refered as Estimated Execution Time (EET).

4.1.1. Estimated Execution Time

The EET Block provides means to measure the execution time of the wrapped code block and perform a measurement based WCET analysis. For each wrapped code block the execution time duration is computed and stored under a block ID. After several runs, the average, standard deviation, 99%-quantile and maximum observed duration are computed, which are also represented in a histogram.

4.2. Time Control Blocks

According to typical patterns on RT control systems, four different time control blocks have been defined: Periodic Execution Time (PET), Forced Execution Time (FET), Budgeted Execution Time (BET), and Period N Execution Time (PNET).

4.2.1. Periodic Execution Time

The PET Block enforces a periodic execution of the wrapped code block. The only argument passed to this block specifies the execution period of the wrapped code block through the frame size (F). As shown in Figure 10, the PET block inserts a delay at the end of its execution in order to consume the remaining time (if any) and maintain the block's periodicity. On the contrary, if the block

takes longer than expected, leading to a period violation, a user defined error handling routine takes place. There should only exist one PET block in the whole legacy control application, which will always be the root node.

4.2.2. Forced Execution Time

The FET Block enforces a concrete duration (specified in its only input argument) for the wrapped code block. Taking a look into Figure 10 it can be seen that, as it is done in the PET block, extra duration at the end of the block will be consumed (delay insertion). However, if the specified duration is exceeded a user defined error handling routine takes place. FET blocks are always defined within another FET or PET block. The use of FET blocks is unlimited; nevertheless, the temporal overhead the block management structure entails should be taken into consideration.

4.2.3. Budgeted Execution Time

The BET Block defines an upper bound duration for the wrapped code block. A BET block should always be defined within another FET or PET block. As shown in Figure 10, the time remaining at the end of BET block's execution is passed to the next sibling BET or PNET block (described latter). The parent FET or PET block is in charge of managing the execution time budget. However, as FET and PET blocks have a fixed duration, BET blocks can only use the remaining time budget of earlier finished same level BET or PNET blocks. The use of BET blocks is unlimited; nevertheless, the timing overhead introduced by the block management structure should be taken into consideration.

4.2.4. Period N Execution Time

The PNET Block accepts three input arguments. The first one determines an upper bound duration for the wrapped code block, the second one determines the N^{th} period at which the blocks is active, whereas the third one determines the offset of the wrapped code block in periods with respect to the start of the execution. Combining the period and offset arguments, tasks can be mapped to different frames (i.e. N^{th} period 3 and offset 1 means that the block will run every three frames starting with the first run at the second frame due to 1 period offset). A PNET block must always be used within a FET or PET block and remaining time at the end of a PNET block is passed through sibling BET blocks (see Figure 10). The parent FET or PET block manages the execution time budget among sibling PNET and BET blocks. As in previously described control blocks, a user defined error handling routine takes place if the time budget is exceeded. The use of PNET blocks is unlimited, however, PNET blocks within the same parent FET or PET block should be defined combining the second and third parameters in such a way that they will never run at the same time. Moreover, the temporal overhead of the block management structure should also be considered.

5. Formal Timing Specification

Within this research work, formal timing specifications are based on MULTIC Time Specification Language (MTSL) (Bde et al., 2017), a timing specification language defined within the MULTIC project [1].

5.1. Components & Contracts

The MULTIC project assumes systems to be built from components (see Figure 3), which interact with the environment (including other components) through a set of ports linked with connectors. Within this context, timing specifications are defined over the ports of components, where any behaviour in the component model is observable.

Port :: *PortName* — *ComponentName* '**.**' *PortName*

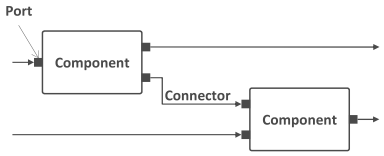


Figure 3: General Component Model.

Timing specifications about components are expressed in terms of contracts by instances of MTSL.

[1] Within the MULTIC project, *parameters* are written in *slanted* font, whereas **keywords** are written in **bold** font and additionally enclosed in quotation marks (as '**keyword**') when they are hardly recognisable. Optional parts are enclosed in brackets and followed by a question mark, such as [optional part]?. Whereas parts that may occur zero or more times are also enclosed in brackets but followed by a star, as for example [repeated part]*. Grammar patterns are composed (from left to right) of a name separated with a double colon :: from the definition. Alternatives within the definition are separated by a vertical bar denoting for this — that.

A contract states assumption(s) (denoted with an A) about the components environment and the behaviour that the component's implementation must guarantee (denoted with a G), considering the component is used in a context where the assumption about the environment is accomplished. Therefore, the example contract in Figure 4 states that assuming that an event occurs on *Entry* port every 20 ms, whenever an event is observed on *Entry* port, an event will be observed on *Exit* port within 0 to 20 ms.

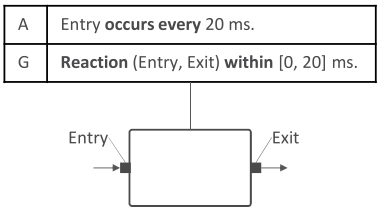


Figure 4: Contract example.

5.2. MULTIC Time Specification Language

MTSL is based specification patterns, natural language like statements defined in terms of time traces that satisfy a pattern. Time traces are based on the notion of sampled signals (MTSL focuses on discrete-event signals) to determine the value of variables in the time domain.

Definition 3. (Timed Trace). A timed trace observed at port p, is defined as a tuple $\omega_p = (t_i, \sigma_i)_{i\in\mathbb{N}}$, where $(t_i)_{i\in\mathbb{N}}$ is an infinite sequence of monotonic time instances and, for each time instance, $\sigma_i \in \Sigma_p$ denotes the corresponding element from the value domain of port p. Timed traces are required to be non-zero, therefore, for each $t \in \mathbb{T}$ exists a timed trace (t_i, σ_i) such that $t_i \geq t$. A set of timed traces observed at port p is denoted by $\Omega_p = \{\omega = (t_i, \sigma_i)_{i\in\mathbb{N}}\}$.

The same way, a set of timed traces observed at a port set $P = \{(p_i)_{i\in\mathbb{N}}\}$ is denoted by $\Omega_P = \{\omega_P = (t_i, \vec{\sigma_i})_{i\in\mathbb{N}}\}$, where $\vec{\sigma_i} = (\sigma_1, \ldots, \sigma_n) \in \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$.

Projection $\omega_P|_q$ of traces over a port set P to port $q \in P$, where $\omega_P|_q = (t_i, \sigma_i^q)_{i\in\mathbb{N}}$ if and only if $\omega_P = (t_i, (\ldots, \sigma_i^q, \ldots)_i)_{i\in\mathbb{N}}$.

Timing specifications describe relations among events, which are only observable at ports and fixed to a value domain. A timing specification refers to one or multiple events, where the event value may or may not be of importance:

EventSpec :: *Port* | *Port* '**.**' *EventValue*

The *EventValue* could consist of labels as well as (complex) values and if it is not specified in the *EventSpec*, any event observed at the *Port* is considered.

So, given a timed trace $\omega = (t_i, \sigma_i)_{i\in\mathbb{N}}$ and an event $\omega \in (t_i, \sigma_i)$, it is said to satisfy the event specification, denoted as $(t_i, \sigma_i) \models$*EventSpec*, if either *EventSpec* specifies a port and σ_i corresponds to its value domain or *EventSpec* specifies an event value and σ_i equals to it.

Timing specifications can either refer to an only event, an event sequence or a set of events:

EventExpr :: *EventSpec* | '**(**' *EventList* '**)**' |
'**{**' *EventList* '**}**'
EventList :: *EventSpec* ['**,**' *EventSpec*]*

Extending the notion of satisfaction to event expressions, a timed trace $(t_i,\ \sigma_i),\ \ldots,\ (t_{i+n-1},\ \sigma_{i+n-1})$ is said to satisfy an event sequence $es = (e_1,\ \ldots,\ e_n)$ if every event $(t_{i+k-1},\ \sigma_{i+k-1})$, $1 \leq k \leq n$, satisfies the event specification e_k. While $(t_i,\ \sigma_i),\ \ldots,\ (t_{i+n-1},\ \sigma_{i+n-1})$ satisfies an event set $es = \{e_1, \ldots, e_n\}$ if a sequence $(e_{s_1}, \ldots, e_{s_n})$ exists that satisfies $\{e_{s_1}, \ldots, e_{s_n}\} = \{e_1, \ldots, e_n\}$.

Timing specifications may refer to a time point or interval:

TimeExpr :: *Value Unit*
Boundary :: '[' | ']'
Interval :: *TimeExpr* | *Boundary Value* '**,**'
Value Boundary Unit

Time units and values in time expressions are restrict to usual time units and simple numbers:

Unit :: **s** | **ms** | **us** | **ns**
Number :: **0** .. **9**[**0** .. **9**]*
Value :: *Number* | *Number* '**.**' *Number*

In the following some of the patterns defined within the MULTIC time specification language (those relevant for this work) are presented [2].

5.2.1. Event Occurrence

To describe a repetitive event occurrence in a particular port, such as periodic events or events with minimum and maximum inter-arrival times, the *Repetition* pattern is introduced:

[2]Detailed information on the Timing Specification Language can be found in (Bde et al., 2019), Chapter 3.

Repetition :: *EventList* **occurs every** $Interval_1$ [**with** *RepetitionOptions*]?.
RepetitionOptions :: *Jitter* [**and** *Offset*]? | *Offset* [**and** *Jitter*]?
Jitter :: **jitter** *TimeExpr*
Offset :: **offset** $Interval_2$

The parameter $Interval_1$ in the *Repetition* pattern determines the minimum and maximum time interval between subsequent occurrences of the *EventList*. The *Jitter* defines an additional delay between subsequent occurrences of the *EventList*, whereas the *Offset* defines a delay interval for the first occurrence of the *EventList*.

Definition 4. (Repetition pattern semantics). Semantics of the repetition pattern "EL **occurs every** I **with jitter** J **and offset** O." is defined as the set of timed traces $(t_i, \vec{\sigma_i})_{i\in\mathbb{N}}$ such that $\vec{\sigma_i}$ corresponds to the event list EL, and $t_i = u_i + j_i \wedge u_0 \in O \wedge u_{i+1} - u_i \in I \wedge j_i \in [0, J]$, where $I = (P^-, P^+)$ is the specified interval (in which (and/or) may be replaced by [and/or] respectively to indicate a closed upper and/or lower bound), $O = [O^-, O^+]$ is the offset interval, and $J \geq 0$ is the jitter. $P^- > 0$ is required.

Figure 5 shows multiple pattern instances with different parameters. The first one, shows a minimal instance of the pattern, with no jitter and no offset. In the second instance of the pattern a jitter up to 5 ms is added. The light blue bars in the time-line mark the period intervals as for the first patter instance, showing that the jitter is "added" to the "baseline" periodic behaviour. The third instance defines a period interval (between 20 and 25 ms). As none of the first three patterns defines an offset, the first event always occurs at 0 ms. On the contrary, the forth pattern defines an offset between [0, 10] ms. Therefore, the first event occurs somewhere in the interval (for example at 5 ms), whereas the time interval between two successive event occurrences is within an interval [20, 25] ms.

5.2.2. Reaction Constraints

The *Reaction* pattern, which describes a forward delay over events, event sets and event sequences is defined as follows:

Reaction :: **whenever** *EventExpr* **occurs then** *EventExpr* **occurs within** *Interval* [**once**]?.

Definition 5. (Reaction pattern semantics). Semantics of the reaction pattern "**whenever** es_1 **occurs then** es_2 **occurs within** I.", where es_1 and es_2 are either a set or a sequence of events that contain k and l events, respectively, is defined as a set of timed traces $(t_i, \sigma_i)_{i\in\mathbb{N}}$ such that $\forall (t_i, \sigma_i) \dots (t_{i+k-1}, \sigma_{i+k-1}) \models es_1 : \exists\ j \geq i + k : (t_j, \sigma_j) \dots (t_{j+l-1}, \sigma_{j+l-1}) \models es_2 \wedge t_{j+l-1} - t_{i+k-1} \in I$.

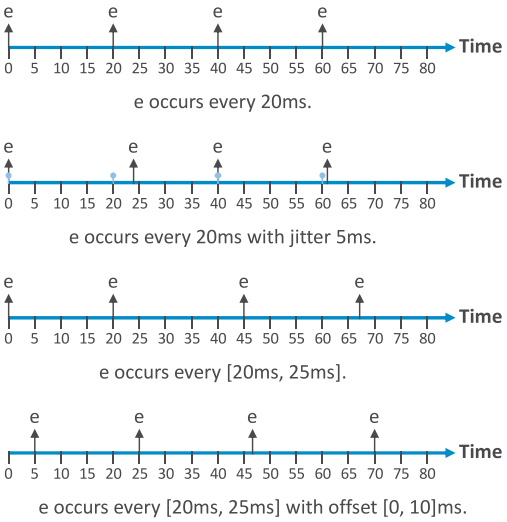


Figure 5: Event occurrence pattern examples.

The optional keyword **once** states that the pattern fails if more than one reaction occurs within the determined time window, therefore only one $j \geq i + k$ exists such that the corresponding sequence satisfies es_2.

Figure 6 depicts multiple instances of the reaction pattern. The first time-line shows a fragment of a pattern instance where event f occurs within [15, 25] ms since event e occurs. In contrast to the first instance, the second one forbids multiple occurrences of event f within the specified time-window using the keyword **once**. The third pattern instance defines an event sequence (f, g) instead of a single event as the reaction to event e.

5.2.3. Causal Event Relations

To be able to reason, beyond the timely behaviour of events, about relation of events the order of occurrence of different events shall be captured. MTSL allows defining basic functional relations [3] by assigning event values. The formal definition of *causal event relations* is as follows:

[3] MTSL does not (yet) support more complex functional relations of events.

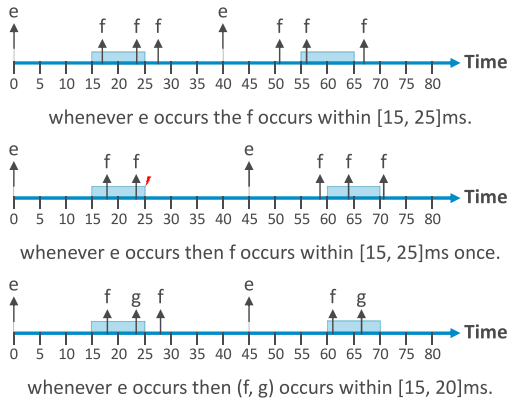


Figure 6: Reaction pattern examples.

Definition 6. (Causal Event Relation). Consider ports p_1 and p_2, and let Ω_{p_1,p_2} be the semantics of the ports. A causal event relation between p_1 and p_2 is a function

$$\triangleright(p_1, p_2) : (\mathbb{T} \times \Sigma_{p_1}) \rightarrow 2^{\mathbb{T} \times \Sigma_{p_2}}$$

where for all $\omega \in \Omega_{p_1,p_2}$ and for all event occurrences $(t_i, \sigma_i) \in \omega|_{p_1}$ exist (t_j, σ_j), ..., $(t_k, \sigma_k) \in \omega|_{p_2}$ such that it holds $\triangleright(p_1, p_2)((t_i, \sigma_i)) = \{(t_j, \sigma_j), \ldots, (t_k, \sigma_k)\}$ and $t_i \leq t_j, \ldots, t_k$.

Causal event relations are transitive, meaning that given three ports p_1, p_2, p_3 and causal event relations $\triangleright(p_1, p_2)$ and $\triangleright(p_2, p_3)$, the causal event relation $\triangleright(p_1, p_3)$ is given by:

$$\begin{aligned}(t_j, \sigma_j) &\in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge (t_k, \sigma_k) \\ &\in \triangleright(p_2, p_3)((t_j, \sigma_j)) \\ &\Rightarrow (t_k, \sigma_k) \\ &\in \triangleright(p_1, p_3)((t_i, \sigma_i))\end{aligned}$$

Based on the defined causal event relation, the causal version of the *Repetition* pattern is introduced:

CausalReaction :: **Reaction(***EventSpec* ',' *EventSpec* **within** *Interval.*

Definition 7. (Causal reaction pattern semantics). Semantics of the causal reaction pattern "**Reaction**(e_1, e_2) **within** I.", where e_1 and e_2 refer to p_1 and p_2, respectively, is defined as the set of timed traces $\omega \in \Omega_{p_1,p_2}$ where for all $(t_i, \sigma_i)_{i \in \mathbb{N}} \in \omega|_{p_1}$, $(u_i, \rho_i)_{i \in \mathbb{N}} \in \omega|_{p_2}$, and for all event occurrences $(t_i, \sigma_i) \in (t_i, \sigma_i)_{i \in \mathbb{N}}$ such that $\sigma_i \models e_1$, holds $\triangleright(p_1, p_2)((t_i, \sigma_i)) \neq \emptyset$ and $((u_j, \rho_j) \in \triangleright(p_1, p_2)((t_i, \sigma_i)) \wedge p_j \models e_2) \Rightarrow u_j - t_i \in I$.

An example of the causal reaction pattern is shown in Figure 7. The dashed light blue line states that those events are related by the definition of causal reaction event relation. In contrast to the first instance in the non-causal reaction pattern (see first pattern instance in 6), in the causal reaction pattern instance (see Figure 7) the reaction to the second occurrence of event e violates the pattern due to the causal relation.

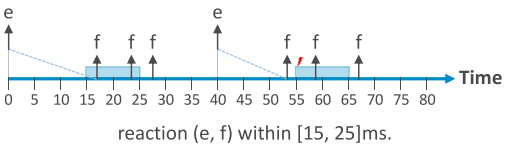


Figure 7: Causal reaction pattern example.

6. RT Legacy Software Migration

Figure 8 depicts (from left to right) the real-time legacy software migration process (described in the following subsections) that ports the RT legacy control software (that complies with the legacy system model defined in section 3) running on top of the legacy hardware platform to a new (different) hardware architecture.

The migration process consists of four main steps. The first step (marked with number 1 in Figure 8) corresponds to the process of **lifting** the legacy timing properties (extract legacy timing properties, make them implicit in the legacy application and transform them into formal timing specification), which is presented in Section 6.1. To this end, time measurement and control blocks are annotated within the legacy application. These blocks, which are based on the time measurement and control blocks presented by Bruns et al. (2019), provide means to extract the legacy timing properties and enforce them during execution. The annotated code is then **tested** to check whether it meets the legacy system's timing and functional properties (see number 2a in Figure 8). To do so, timing properties are transformed into formal timing specifications and compared against time traces, whereas a set of reference values for input state variables and the corresponding reference values for output control variables are extracted from the legacy system to check whether the functional behaviour is preserved. According to the results obtained, if any of the requirements (temporal and/or functional) are not met, time control blocks might have to be reallocated

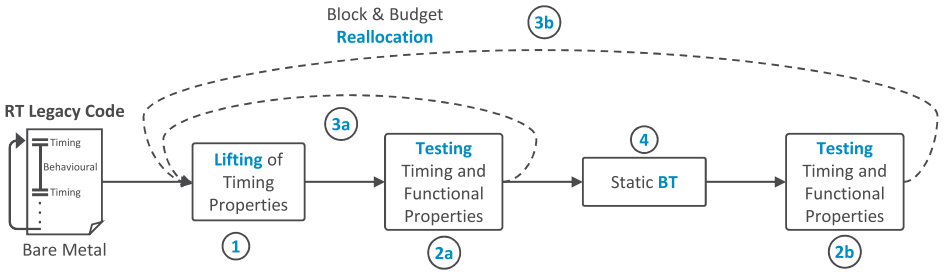


Figure 8: RT Legacy Software Migration flow. Lifting of timing properties (step 1) is described in Section 6.1. Testing timing and functional properties (step 2), as well as block and budget reallocation (step 3) are described in Section 6.2. Timing equivalent legacy software porting (step 4) is described in Section 6.3.

and the time budget as well as the timing specifications **adjusted**, see number 3a in the figure. The timing and functional test procedure as well as the block and budget reallocation process is described within Section 6.2. The next step, number 4 in Figure 8, consists of porting the timing annotated binary code to the ISA of the new hardware platform, with a focus on how the static **binary translation** tool handles the timing annotations. The timing block aware translation process is described in Section 6.3. Finally, after the annotated legacy code has been translated, temporal and functional requirements are **tested once again** (marked as 2b in Figure 8) and if needed time control blocks reallocated and time budget as well as timing specifications adjusted (marked as 3b in Figure 8). It is worth to mention that the block reallocation as well as the time budget and contract **adjustment** step might have to be accomplished several times during the migration process.

6.1. Lifting of Timing Properties

The timing property lifting process presented in fig. 9 consists of three main stages: profiling, annotation, and time specification.

6.1.1. Profiling Legacy System

Profiling constitutes the first stage, where code analysis, legacy system's specifications and timing measurements are evaluated by an expert to extract the necessary timing information from the legacy system. Information regarding the period of each task, precedence relation and data sharing among tasks, as well as identification of critical sections is obtained through the analysis of the legacy code, legacy system's specifications and expert knowledge. Whereas time measurement (the EET described in section 4.1.1), which are systematically annotated into the legacy code, provide means to measure the end-to-end duration of a specific code section and save the result in memory (under block's ID). The obtained timing data is then analysis to extract information regarding the WCET and offset of tasks. As a result of the profiling phase, legacy timing properties and the behavioural legacy code are obtained.

6.1.2. Legacy Timing Enforcement

To enforce an appropriate temporal behaviour after the migration process, implicit legacy timing properties are made explicit in the behavioural legacy code. The proposed time control solution is based on a block-level source code (systematic) annotation approach.

Based on the described time control blocks (see Section 4.2), the example legacy application presented in Section 3.3 is annotated as follows. The root node is a PET block with period set to 20 ms, which is the GCD of all tasks' period. Then, each task is wrapped into a BET or PNET block according to its period. Tasks with a period equal to that defined through the PET block are wrapped into BET blocks, whereas tasks with a greater period are mapped into PNET blocks. The budget for either block (BET or PNET) is determined by the WCET of the task it contains. Whereas the period and offset parameters of PNET blocks are determined according to the period of the task they wrap. For the example application, tasks t_3 and t_4 have a period two times greater than that defined on the PET block, therefore, the period argument is set to 2, whereas the offset parameter is set to 0 in the PNET block that wraps t_3 and to 1 in the block containing t_4. This way f_3 and t_4 will run every two periods starting at period 0 and period 1 respectively. Finally, in order to preserve a specific

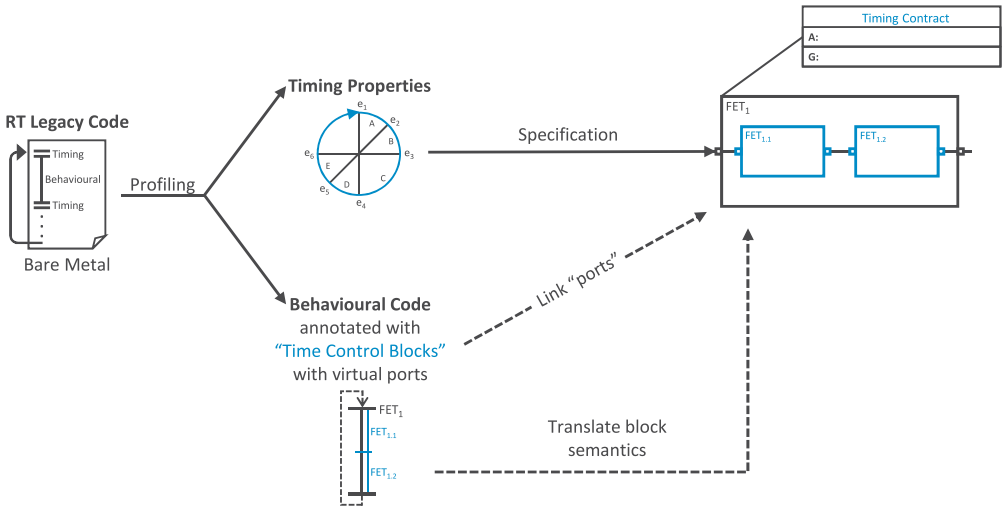


Figure 9: Lifting of timing properties. Profiling phase: extract timing properties and behavioural code from RT legacy code. Annotation phase: annotate behavioural code with time control blocks. Time Specification phase: transform annotated timing properties into timing specifications attached to virtual ports.

offset and minimum jitter among subsequent task instances for critical tasks, every BET or PNET block preceding a block wrapping a task marked as critical section must be wrapped into a FET block. Therefore, BET blocks corresponding to t_1 and t_2 are wrapped into a FET block. The duration of this FET block is set to 10 ms so that the next tasks marked as critical preserve their offset. The PNET blocks and the BET block corresponding to t_5 are wrapped into another FET block. The upper limit of PNET blocks that can be wrapped in a FET block is determined by the period argument of the PNET blocks, which has to be equal for every PNET block within a FET, while the offset has to be different for each PNET block within a FET. To preserve the offset of the following critical task (t_6), the duration of the FET block is set to 5 ms. The resulting annotated code is shown in Figure 10, and the corresponding tree diagram in Figure 11.

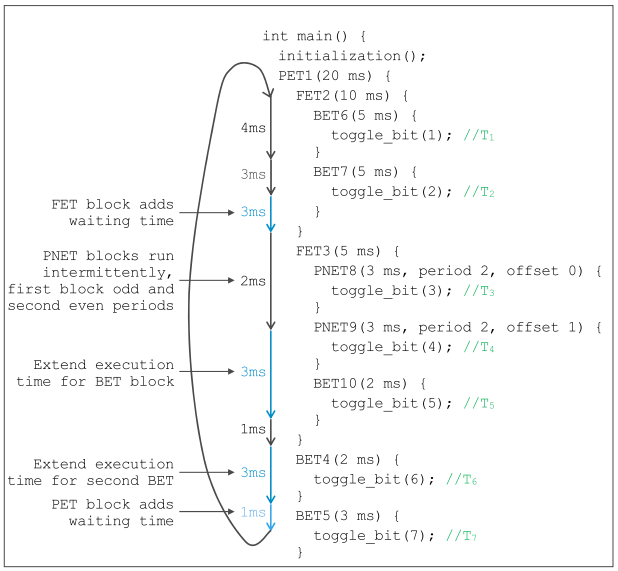


Figure 10: Time control blocks' structure and functionality. Description of execution time control blocks' structure and functionality through the RT legacy application example (see Section 3.3). PET block enforces a periodic execution. FET enforces a concrete duration. BET defines an upper bound duration and passes remaining time to next sibling BET or PNET blocks. PNET defines an upper bound duration for a code block that runs every N^{th} period starting at an specific period (according to its offset).

6.1.3. Extract Timing Specifications

Finally, the annotated legacy application is systematically transformed into formal time specifications (see Section 5) that provide means for the latter validation of legacy timing properties.

Based on MULTIC approach, the RT legacy software migration solution describes each time control block as a component. Within each component, virtual ports are defined, at which events are observable, as well as the corresponding contract, which is based on MTSL repetition and causal reaction patterns. Moreover, component-to-component connections are done according to a causality order. Therefore, the annotated legacy example application presented in Figure 10, which follows a tree di-

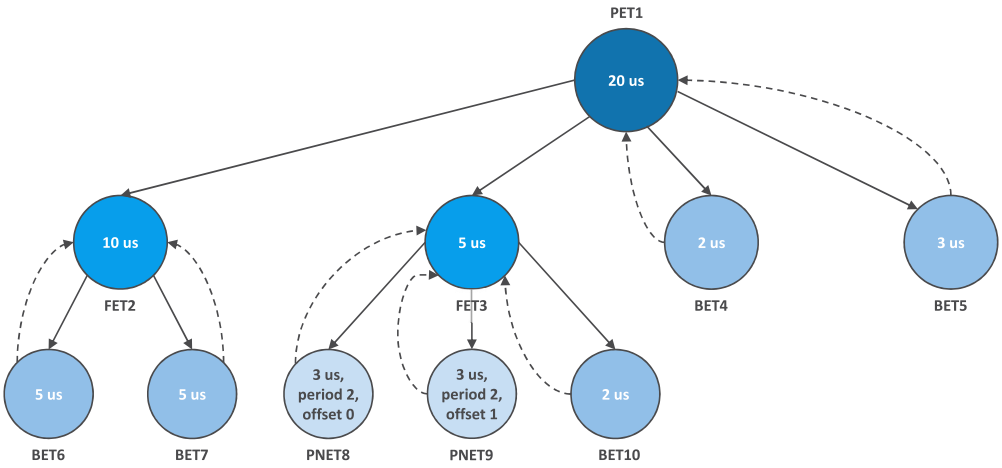


Figure 11: Time control blocks' behaviour and nesting. Description of execution time control blocks' behaviour and nesting through the RT legacy example application (see Section 3.3). PET1 is the root node. It has four child nodes: FET2, FET3, BET4 and BET5. PET1 manages the timing budget passed across BET4 and BET5 siblings. FET2 has two child nodes (BET6 and BET7) and manages the time budget passed across them. FET3 has three child nodes, two PNET blocks (PNET8 and PNET9) and a BET block (BET10). The timing budget passes across the sibling PNET and BET blocks is managed by their parent node, FET3.

agram as depicted in Figure 11, is transformed into a set of component nodes with their corresponding contract. The resulting component-contract structure is shown in Figure 12.

This component-contract structure will later be used to validate the timing behaviour of the annotated legacy application. However, before applying the time specification, a consistency check must be performed. To this end, the concept of Virtual Integration Testing (VIT) is being used, which checks compatibility and refinement conditions. The former verifies whether two (or more) components can be put together without violating any of the contracts. That means that two components are compatible if the assumption about the environment of a component are not violated by another component connected to this component. The latter verifies whether the composition of a set of sub-components satisfies the contract(s) of the parent component. This means that the guaranteed behaviour of a set of sub-components must refine the behaviour guaranteed by the parent contract, within an environment that complies with the parent contract assumption. The details about VIT are out of the scope of this paper and can be found in (Bde et al., 2017)

6.2. Testing, Reallocation & Adjustment

To check that timing and functional requirements are still met after the lifting process, the control block annotated legacy application is executed on the legacy hardware platform and collected functional and timing traces are compared against their reference values.

On the one hand, the timing test compares time traces that time control blocks generate at runtime against systematically obtained formal timing specifications using the MULTIC tooling (Bde et al., 2019), which allows expressing timing requirements in terms of contracts and provides means for their validation through a simulation based method based on the SystemC (Accellera, 2019) simulation framework. Figure 13 depicts the process of testing the timing properties.

On the other hand, the functional test feeds the annotated legacy application running on the legacy hardware platform with a set of reference input data for state variables (usually provided by an expert group) and compares the output values obtained for control variables against a set of reference output data, which is obtained running the (original) legacy application on the legacy hardware platform. Figure 14 depicts the process of testing the functional properties.

According to the timing and functionality test result, a time control block reallocation and budget adjustment process might be necessary. During the lifting process, source code is extracted, modified and rearranged. In this reverse engineering process, the functional and timing behaviour of the legacy

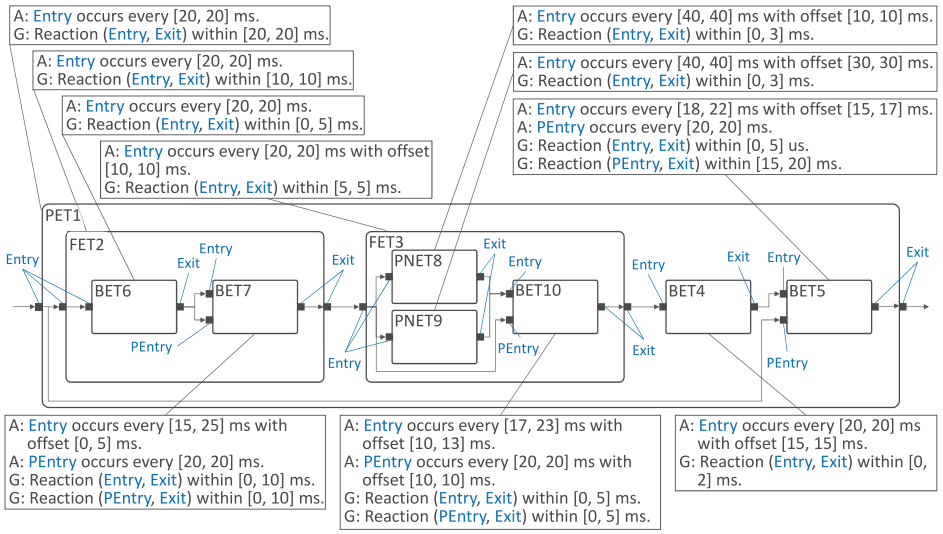


Figure 12: Ideal component-contract structure for the annotated example application. Description of component-contract structure through the RT legacy example application (see Section 3.3). Each component has its corresponding contract, composed of assumption(s) and guarantee(s), which are based on MTSL repetition and causal reaction patterns. Port names are shown in blue colour. Contacts describe an ideal behaviour and therefore do not describe possible time variations (jitter) present in real a scenario.

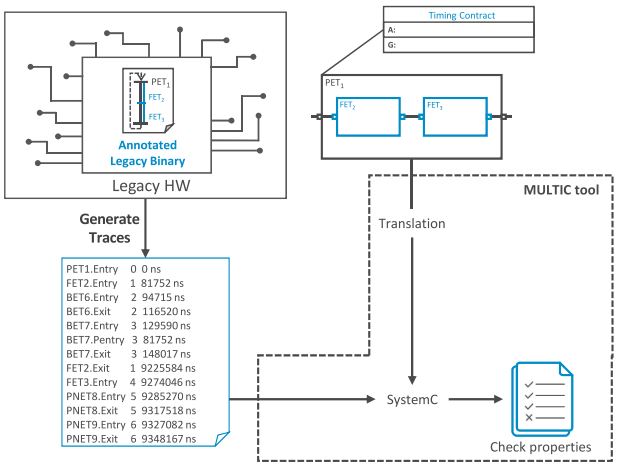


Figure 13: Description of the process for testing timing properties.

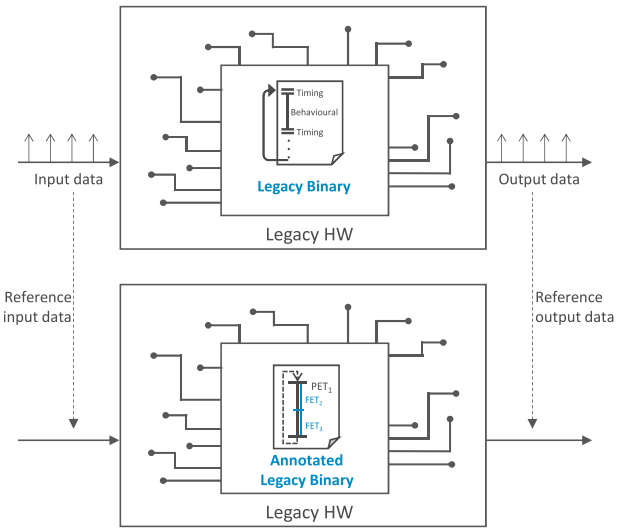


Figure 14: Description of the process for testing functional properties.

application might have been changed. Therefore, if the timing and/or functional test results is not successful (i.e., the allocated budget is not enough, the offset before critical section tasks is not appropriate, the task precedence relation has been corrupted), time control blocks need to be reallocated and budgets adjusted accordingly, while still ensuring an appropriate timing behaviour.

Systematically generated timing specifications (see Figure 12) describe an ideal timing behaviour. However, in a real scenario the timing behaviour is not ideal and timing deviations exist as a consequence to the overhead of time control block management or binary translation, as well as deviations caused by the underlying OS or hardware platform itself. As a consequence, formal timing specifications might need to be adjusted accordingly, while still ensuring an appropriate timing behaviour.

During the lifting process it might be necessary to repeat the temporal a functional property test, the control block reallocation and the budget and contract adjustment process several times, until all legacy system's requirements are met, temporal as well as functional requirements.

6.3. Timing Block support within Static BT

The RT legacy software migration approach relies on a static binary analysis framework, Rev.ng (Federico et al., 2017). The core element in Rev.ng is its SBT tool, that combines the benefits of QEMU (Bellard, 2005) with those of LLVM (Lattner and Adve, 2004). Revamb parses the statically linked Linux binary and uses QEMU's TCG as a front-end to generate tiny code instructions from any of the input architectures it supports. Then code in QEMU IR form is further translated into LLVM IR instructions. However, in QEMU, certain features such as syscalls and complex instructions (e.g. floating point division) are handled through a set of external functions (written in C) known as helper functions. Therefore, using Clang, QEMU helper functions are obtained in the form of LLVM IR and statically linked before generating the LLVM module. Besides the helper functions, additional support is needed mainly for initialization purposes. To this end, Revamb provides a set of support functions which are linked to the LLVM module. Then, the linked LLVM IR module is translated into machine code using LLVM compiler infrastructure.

For the static translation, the time control block annotated legacy application, obtained as a result of the lifting process, is statically linked to the Timing Measurement and Control Block (TMCB) library and compiled for the legacy hardware platform using a legacy Linux toolchain. The technical implementation of TMCBs as a library enables its use across multiple platforms. Then, the annotated binary is statically translated from legacy to the new ISA using Rev.ng tool-suite. Figure 15 depicts the described static binary translator based timing control block handling.

As it is done after the lifting process, after translating the annotated code temporal and functional properties need to be tested on the new hardware platform. If the test results are unsuccessful, time control blocks might have to be reallocated and/or the assigned time budget and/or timing specifications adjusted (see Section 6.2 for more information).

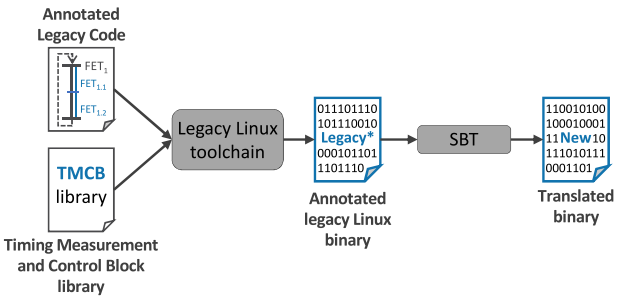


Figure 15: Description of the static binary translator based block handling.

7. Timing-aware Migration Assessment

As a proof of concept, the RT legacy software migration approach described in the previous chapter (see Section 6) is used to port ARM Cortex-A9 legacy software to an Intel Atom processor. Therefore, the Xilinx Zynq-7000 System on a Chip (SoC) ZC702 evaluation kit and the MinnowBoard Turbot Dual-Core board have been selected as source and target platforms, respectively. The timing-aware migration assessment describes the evaluation of the implemented block level timing enforcement mechanism as well as the evaluation of the timing-aware static translation process. For each of the evaluation scenarios, the functional as well as the timing behaviour have been assessed. The migration assessment is accomplished through the example application (described in Section 3.3) and a flight control application. The example application provides a complete analysis in the sense that it contains tasks with different periods and some of the tasks are considered a critical code section, therefore, through the example every time control block can be evaluated, analysing this way corner cases. However, this is a self generated application and therefore the great effort of porting non-proprietary code is excluded. On the contrary, the flight control application is a legacy application that was implemented by a third party and therefore, it is a suitable application to evaluate the effort of porting non-proprietary legacy code. Although every corner cases is not covered through the flight control application, the analysis of corner cases is achieved through the example application.

The example application, first introduced in Section 3.3, resembles the typical pattern of a reactive control system. This bare-metal application includes seven periodic tasks (with different periods) executed following a static scheduling policy.

The functional behaviour of the example application consists on a bit toggling sequence, where each of the sequential tasks toggles a specific bit in an output variable (the bit in the position corresponding to the task number, the bit in position zero never toggles). Running the legacy example application on the ARM Cortex-A9 processor for 50000 time steps (statistically representative enough), the output variable bit toggling sequence is observed and reference output data collected.

The flight control application that will be used in the evaluation process is the OFFIS multirotor (OFFIS, 2019), which supports a mixed-critical architecture and enables high-performance processing while still supporting a safe flight control. The flight control algorithm is responsible for computing the motor values (control variables) based on the control orders from the user (set-point) and the sensor data (process variables). Figure 16 depicts the main components of the flight controller, which are *Read Sensors*, *Sensor Processing*, *Read Remote*, *Remote Processing*, *Flight Controller*, and *Send Motor Values*. To guarantee a stable flight behaviour, an update rate of 500 Hz must be ensured in the motor drivers, therefore the control cycle cannot exceed 2.1 ms. The multirotor application running on top of the ARM Cortex-A9 processor is feed with the reference input data, generated through a flight simulation program (the multirotor takes off, hoovers for a while and then lands), the execution lasts for 3000 time steps (statistically representative enough) and control variables are observed to collect the reference output data.

7.1. Block-Level Timing Enforcement Assessment

The timing enforcement solution presented in Section 6.1, which consist of a block-level source code annotation mechanism is being assessed in this section. To this end, each application is systematically annotated with timing control blocks [4]. The annotated application is linked against the the TMCB library and compiled for the ARMv7-A ISA. Then, the time control block annotated application runs on top of the ARM Cortex-A9 processor (running Preempt-RT Linux at a 666 MHz operation frequency) and the corresponding functional and timing data is collected.

[4]To annotate the example application every type of timing control block has been used, whereas to annotate the multirotor applications just PET blocks are needed.

7.1.1. Timing test

As described in Section 6.2, to test the timing behaviour, the annotated code (example – see Listing 1, and multirotor – see Listing 2) is systematically transformed into formal timing specifications (example – see Figure 12, and multirotor – see Figure 17). These timing specifications, together with the time traces that the annotated binary generates when running on top of the ARM Cortex-A9 processor are fed to the MULTIC tool and thus the timing behaviour is validated.

```
void main() {
  initialization();
  BLOCK_PET(20_ms) {
    BLOCK_FET(10_ms) {
      BLOCK_BET(5_ms) {
        toggle_bit(1, &output_var); //t1
      }
      BLOCK_BET(5_ms) {
        toggle_bit(2, &output_var); //t2
      }
    }
    BLOCK_FET(5_ms) {
      BLOCK_PNET(3_ms, 2, 0) {
        toggle_bit(3, &output_var); //t3
      }
      BLOCK_PNET(3_ms, 2, 1) {
        toggle_bit(4, &output_var); //t4
      }
      BLOCK_BET(2_ms) {
        toggle_bit(5, &output_var); //t5
      }
    }
    BLOCK_BET(2_ms) {
      toggle_bit(6, &output_var); //t6
    }
    BLOCK_BET(3_ms) {
      toggle_bit(7, &output_var); //t7
    }
  }
}
```

Listing 1: Example application annotated with time control blocks.

```
void main(void)
{
  platform_init();
  BLOCK_PET(2_ms)
  {
    platform_execute();
  }
}
```

Listing 2: Multirotor application annotated with time control blocks.

Example application – ideal contracts.

Table 2 shows the time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 12).

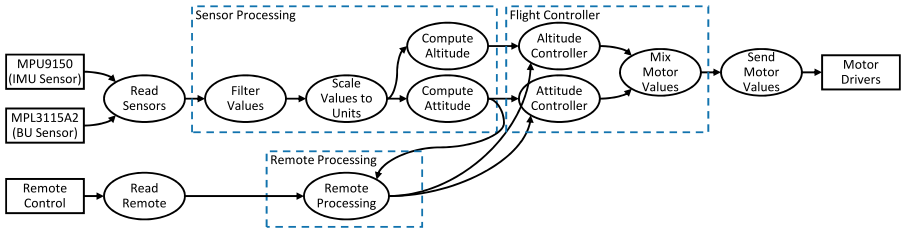


Figure 16: Overview of the flight controller SAFEPOWER (2017)

Table 2: Time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 12). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET20.Entry 0 ns	[0,0] ns	✓
FET21.Entry 504825 ns	[0,0] ns	✗
BET22.Entry 580910 ns	[0,0] ns	✗
BET22.Exit 706839 ns	[580910,5580910] ns	✓
BET26.Entry 761326 ns	[0,5000000] ns	✓
BET26.PEntry 504825 ns	[0,0] ns	✗
BET26.Exit 901679 ns	[504825,10504825] ns	✓
FET21.Exit 9987845 ns	[10504825,10504825] ns	✗
FET31.Entry 10070300 ns	[10000000,10000000] ns	✗
PNET32.Entry 10131856 ns	[10000000,10000000] ns	✗
PNET32.Exit 10248073 ns	[10131856,13131856] ns	✓
BET40.Entry 10304820 ns	[10000000,13000000] ns	✓
BET40.PEntry 10070300 ns	[10000000,10000000] ns	✗
BET40.Exit 10418145 ns	[10304820,15070300] ns	✓
FET31.Exit 14497737 ns	[15070300,15070300] ns	✗
BET45.Entry 14593920 ns	[15000000,15000000] ns	✗
BET45.Exit 14716674 ns	[14593920,16593920] ns	✓
BET49.Entry 14769626 ns	[15000000,17000000] ns	✗
BET49.PEntry 0 ns	[0,0] ns	✓
BET49.Exit 14907534 ns	[15000000,19769626] ns	✗
PET20.Exit 19989077 ns	[20000000,20000000] ns	✗
PET20.Entry 20052985 ns	[20000000,20000000] ns	✗
FET21.Entry 20112780 ns	[20504825,20504825] ns	✗
BET22.Entry 20171498 ns	[20580910,20580910] ns	✗
BET22.Exit 20284313 ns	[20171498,25171498] ns	✓
BET26.Entry 20337595 ns	[15761326,25761326] ns	✓
BET26.PEntry 20112780 ns	[20504825,20504825] ns	✗
BET26.Exit 20449783 ns	[20337595,30112780] ns	✓
FET21.Exit 29529340 ns	[30112780,30112780] ns	✗
FET31.Entry 29600643 ns	[30070300,30070300] ns	✗
PNET36.Entry 29663642 ns	[30000000,30000000] ns	✗
PNET36.Exit 29779277 ns	[29663642,32663642] ns	✓
BET40.Entry 29832028 ns	[27304820,33304820] ns	✓
BET40.PEntry 29600643 ns	[30070300,30070300] ns	✗
BET40.Exit 29944063 ns	[29832028,34600643] ns	✓
FET31.Exit 34023382 ns	[34600643,34600643] ns	✗
BET45.Entry 34083446 ns	[34593920,34593920] ns	✗
BET45.Exit 34195623 ns	[34083446,36083446] ns	✓
BET49.Entry 34248931 ns	[32769626,36769626] ns	✓
BET49.PEntry 20052985 ns	[20000000,20000000] ns	✗
BET49.Exit 34382318 ns	[35052985,39248931] ns	✗
PET20.Exit 39461774 ns	[40052985,40052985] ns	✗

Multirotor application – ideal contracts.

Table 3 shows the time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 17).

Table 3: Time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the ideal component-contract structure (see Figure 17). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET36.Entry 0 ns	[0,0] ns	✓
PET36.Exit 2042507 ns	[2000000,2000000] ns	✗
PET36.Entry 2122859 ns	[2000000,2000000] ns	✗
PET36.Exit 4155069 ns	[4122859,4122859] ns	✗
PET36.Entry 4221302 ns	[4122859,4122859] ns	✗
PET36.Exit 6251056 ns	[6221302,6221302] ns	✗
PET36.Entry 6313935 ns	[6221302,6221302] ns	✗
PET36.Exit 8344471 ns	[8313935,8313935] ns	✗
PET36.Entry 8422069 ns	[8313935,8313935] ns	✗
PET36.Exit 10452413 ns	[10422069,10422069] ns	✗
PET36.Entry 10515106 ns	[10422069,10422069] ns	✗
PET36.Exit 12545624 ns	[12515106,12515106] ns	✗
PET36.Entry 12607801 ns	[12515106,12515106] ns	✗
PET36.Exit 14636994 ns	[14607801,14607801] ns	✗
PET36.Entry 14698283 ns	[14607801,14607801] ns	✗
PET36.Exit 16727964 ns	[16698283,16698283] ns	✗
PET36.Entry 16790705 ns	[16698283,16698283] ns	✗
PET36.Exit 18820345 ns	[18790705,18790705] ns	✗
PET36.Entry 18893214 ns	[18790705,18790705] ns	✗
PET36.Exit 20923498 ns	[20893214,20893214] ns	✗

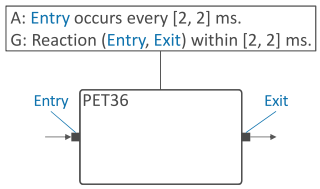


Figure 17: Ideal component-contract structure for the annotated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract describes an ideal behaviour and therefore does not describe possible time variations (jitter) present in real a scenario.

Given that the contracts presented in Figure 12 and Figure 17 are ideal and therefore do not con-

sider any timing variation cause by the overhead of time control block management or by the underlying OS and hardware platform itself, a great percentage of the traces did not pass the corresponding contract (about 60% of the time traces for the example application, and about 99% for the multirotor application). In order to cover the timing deviations present in a real scenario, contracts are adjusted as follows:

- Adjust the offset in some of the assumptions (following a repetition pattern) to cover little variations on the time distance from the start of execution to the first occurrence of an event on a particular port.
- Adjust the interval in some of the assumptions (following a repetition pattern) to cover little variations on the time distance between repetitive event occurrences on a particular port.
- Adjust the interval in some of the guarantees (following a causal reaction pattern) to cover little variations on the time distance between causally related events on input/output ports.

Example application – annotation adjusted contracts.

Figure 18 shows the adjusted component-contract structure for the annotated example application. Changes in the contracts with respect to the ideal component-contract structure in Figure 12 are shown in bold. Up to time steps 1600 adjustments had to be done on the contracts, then the example application run for over 5000 time steps without failing any contract. Table 4 shows the time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure presented in Figure 18. Changes in the time interval with respect to the validation against the ideal component-contract structure (see Table 2) are shown in bold.

Table 4: Time traces generated by the annotated example application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure (see Figure 18). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with ideal component-contract structure (see Table 2) are shown in bold. The third column shows contract pass/fail information.

Time Trace	**Contract limitation**	**Pass Fail**
PET20.Entry 0 ns	[0,0] ns	✓
FET21.Entry 504825 ns	[0,**600000**] ns	✓
BET22.Entry 580910 ns	[0,**600000**] ns	✓
BET22.Exit 706839 ns	[580910,5580910] ns	✓
BET26.Entry 761326 ns	[0,5000000] ns	✓
BET26.PEntry 504825 ns	[0,**600000**] ns	✓
BET26.Exit 901679 ns	[761326,10504825] ns	✓
FET21.Exit 9987845 ns	[**9904825**,10504825] ns	✓
FET31.Entry 10070300 ns	[10000000,**10100000**] ns	✓
PNET32.Entry 10131856 ns	[10000000,**10200000**] ns	✓
PNET32.Exit 10248073 ns	[10131856,13131856] ns	✓
BET40.Entry 10304820 ns	[10000000,13000000] ns	✓
BET40.PEntry 10070300 ns	[10000000,**10100000**] ns	✓
BET40.Exit 10418145 ns	[10304820,15070300] ns	✓
FET31.Exit 14497737 ns	[**14470300**,15070300] ns	✓
BET45.Entry 14593920 ns	[**14500000**,15000000] ns	✓
BET45.Exit 14716674 ns	[14593920,16593920] ns	✓
BET49.Entry 14769626 ns	[**14700000**,17000000] ns	✓
BET49.PEntry 0 ns	[0,0] ns	✓
BET49.Exit 14907534 ns	[**14769626**,19769626] ns	✓
PET20.Exit 19989077 ns	[**19400000**,20000000] ns	✓
PET20.Entry 20052985 ns	[**19400000**,**20100000**] ns	✓
FET21.Entry 20112780 ns	[**19904825**,**20604825**] ns	✓
BET22.Entry 20171498 ns	[**19980910**,**20680910**] ns	✓
BET22.Exit 20284313 ns	[20171498,25171498] ns	✓
BET26.Entry 20337595 ns	[15761326,25761326] ns	✓
BET26.PEntry 20112780 ns	[**19904825**,**20604825**] ns	✓
BET26.Exit 20449783 ns	[20337595,30112780] ns	✓
FET21.Exit 29529340 ns	[**29512780**,30112780] ns	✓
FET31.Entry 29600643 ns	[**26970300**,**32570300**] ns	✓
PNET36.Entry 29663642 ns	[**29600000**,30000000] ns	✓
PNET36.Exit 29779277 ns	[29663642,32663642] ns	✓
BET40.Entry 29832028 ns	[**27204820**,33304820] ns	✓
BET40.PEntry 29600643 ns	[**26970300**,**32570300**] ns	✓
BET40.Exit 29944063 ns	[29832028,34600643] ns	✓
FET31.Exit 34023382 ns	[**34000643**,34600643] ns	✓
BET45.Entry 34083446 ns	[**31393920**,**37293920**] ns	✓
BET45.Exit 34195623 ns	[34083446,36083446] ns	✓
BET49.Entry 34248931 ns	[**31569626**,**37469626**] ns	✓
BET49.PEntry 20052985 ns	[**19400000**,**20100000**] ns	✓
BET49.Exit 34382318 ns	[**34352985**,39248931] ns	✓
PET20.Exit 39461774 ns	[**39452985**,40052985] ns	✓

Multirotor application – annotation adjusted contracts.

Figure 19 shows the adjusted component-contract structure for the annotated multirotor application. Changes in the contracts with respect to the ideal component-contract structure in Figure 17 are shown in bold. On the first two time steps adjustments had to be done on the contract to cover the previously mentioned needs. The interval in the assumption is adjusted to $[2000, 2200]$ us and the interval in the guarantee is set to $[2000, 2100]$ us. Then, the flight control application runs for almost 3000 time steps without any further failure. Table 5 shows the time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure presented in Figure 19. Changes in the time interval with respect to the validation against the ideal component-contract structure (see Table 3) are shown in bold.

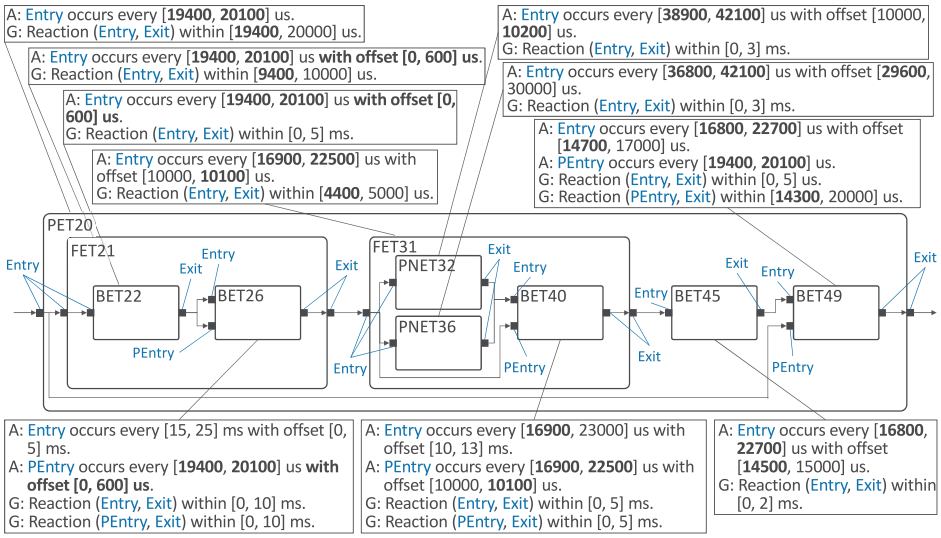


Figure 18: Adjusted component-contract structure for the annotated example application. Each component has its corresponding contract. Contracts describe the assumptions and guarantees observable at input/output ports (named in blue colour). Contracts have been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the annotated example application running on the ARM Cortex-A9 processor. Changes in the contracts with respect to the ideal component-contract structure in Figure 12 are shown in bold.

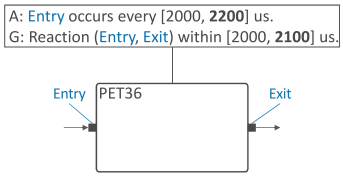


Figure 19: Adjusted component-contract structure for the annotated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible timing deviations present on a real scenario, in this case adjustments are applied for the annotated multirotor application running on the ARM Cortex-A9 processor.

Table 5: Time traces generated by the annotated flight control application running on the ARM Cortex-A9 processor validated against the adjusted component-contract structure (see Figure 19). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with ideal component-contract structure (see Table 3) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET36.Entry 0 ns	[0,0] ns	✓
PET36.Exit 2042507 ns	[2000000,**2100000**] ns	✓
PET36.Entry 2122859 ns	[2000000,**2200000**] ns	✓
PET36.Exit 4155069 ns	[4122859,**4222859**] ns	✓

Continued on next page

Table 5 – *Continued from previous page*

Time Trace	Contract limitation	Pass Fail
PET36.Entry 4221302 ns	[4122859,**4322859**] ns	✓
PET36.Exit 6251056 ns	[6221302,**6321302**] ns	✓
PET36.Entry 6313935 ns	[6221302,**6421302**] ns	✓
PET36.Exit 8344471 ns	[8313935,**8413935**] ns	✓
PET36.Entry 8422069 ns	[8313935,**8513935**] ns	✓
PET36.Exit 10452413 ns	[10422069,**11422069**] ns	✓
PET36.Entry 10515106 ns	[10422069,**12422069**] ns	✓
PET36.Exit 12545624 ns	[12515106,**13515106**] ns	✓
PET36.Entry 12607801 ns	[12515106,**14515106**] ns	✓
PET36.Exit 14636994 ns	[14607801,**15607801**] ns	✓
PET36.Entry 14698283 ns	[14607801,**16607801**] ns	✓
PET36.Exit 16727964 ns	[16698283,**17698283**] ns	✓
PET36.Entry 16790705 ns	[16698283,**18698283**] ns	✓
PET36.Exit 18820345 ns	[18790705,**19790705**] ns	✓
PET36.Entry 18893214 ns	[18790705,**20790705**] ns	✓
PET36.Exit 20923498 ns	[20893214,**21893214**] ns	✓

Result analysis.

Result show that systematically generated formal timing specifications needed to be adjusted to cover the timing deviations caused by the overhead of time control block management or by the underlying OS and hardware platform itself. After the adjustment of formal timing specifications, time traces generated at runtime by the annotate example and multirotor applications fit into the defined timing contracts. Future work considers characterizing the overhead generated by control block management and adjusting the blocks accordingly to, at some point, overcome it.

7.1.2. Functional test

The functional test compares the reference output value for control variables with the output value, for those control variables, when running the annotated applications (example – see Listing 1, and multirotor – see Listing 2) on the ARM Cortex-A9 processor, using reference input data for state variables [5].

Example application.

Figure 20 shows the bit toggling sequence of the output variable on both test scenarios. However, due to scaling problems, the figure depicts just the output variable value for the first 100 time steps.

Multirotor application.

Figure 21 shows the output values for each of the multirotor application control variables on both test scenarios. Control variables are observed for an interval of about 3000 time steps (about 6 seconds simulation).

Result analysis.

Results show that the signal observed on every control variable is equivalent (in value) before and after the lifting of timing properties. However, due to time control block management overhead, a delay (with respect to non-annotated code) is observable on every control variable output signal. This delay in the response of the controller might slightly disturb the functional behaviour of the overall control system, therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the lifting of timing properties. Future work considers providing support to port legacy platform Input/Output (I/O) port dependent code, which will provide means to perform a functional test on a more realistic scenario.

7.2. Timing-aware Static Translation Assessment

This section evaluates the timing equivalent static BT approach described in Section 6.3. As the timing enforcement assessment, the assessment of the timing-aware static legacy software translation is accomplished through the example and multirotor applications. To this end, each of the time control block annotated applications that was statically linked and compiled for the ARMv7-A ISA is statically translated, using Rev.ng, into an equivalent binary for x86 ISA. The translated binary is then executed on top of the Intel Atom E3866 processor (running Preempt-RT Linux at a 1463 MHz operation frequency) and the corresponding functional and timing data is collected.

7.2.1. Timing test

The timing test validates the timing behaviour of translated applications (example – see Listing 1, and multirotor – see Listing 2) running on the Intel Atom E3866 processor. Using MULTIC tool, the time traces that each of the translated applications generates at run-time are validated against the corresponding component-contract structure (example – see Figure 18, and multirotor – see Figure 19) obtained as a result of the lifting of timing properties and adjusted as part of the timing enforcement assessment.

Example application – annotation adjusted contracts.

Table 6 shows the time traces generated by the translated example application running on the Intel Atom E3866 processor validated against the component-contract structure presented in Figure 18, adjusted as part of the timing enforcement assessment.

Table 6: Time traces generated by the translated example application running on the Intel Atom E3866 processor validated against (before translation) adjusted component-contract structure (see Figure 18). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET20.Entry 0 ns	[0,0] ns	✓
FET21.Entry 860897 ns	[0,600000] ns	✗
BET22.Entry 1093058 ns	[0,600000] ns	✗
BET22.Exit 1447619 ns	[1093058,6093058] ns	✓
BET26.Entry 1597033 ns	[0,5000000] ns	✓
BET26.PEntry 860897 ns	[0,600000] ns	✗
BET26.Exit 1966338 ns	[1597033,10860897] ns	✓
FET21.Exit 11138808 ns	[10260897,10860897] ns	✗
FET31.Entry 11445947 ns	[10000000,10100000] ns	✗
PNET32.Entry 11644891 ns	[10000000,10200000] ns	✗
PNET32.Exit 12063298 ns	[11644891,14644891] ns	✓
BET40.Entry 12233075 ns	[10000000,13000000] ns	✓
BET40.PEntry 11445947 ns	[10000000,10100000] ns	✗
BET40.Exit 12580840 ns	[12233075,16445947] ns	✓

Continued on next page

[5]The example application does not require input data.

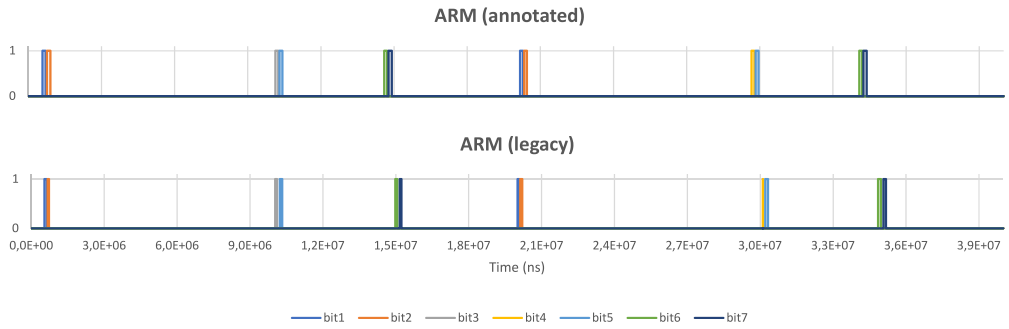
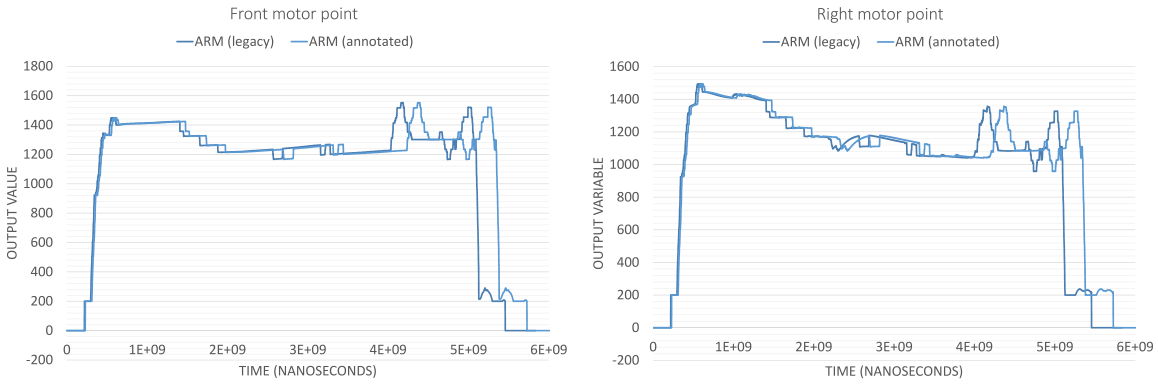
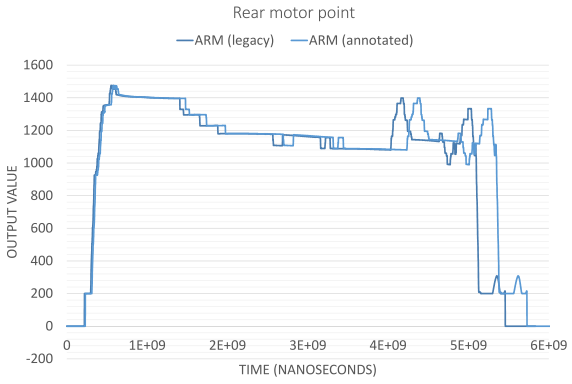


Figure 20: Functional test results for the annotated example application running on the ARM Cortex-A9 processor. For each time step (X-axis) the corresponding output variable value (Y-axis) is shown, both, for the legacy example application (without annotations) as well as for the annotated example application. On both test scenarios (legacy and annotated), the bit toggling sequence is equal, although there is a delay on the annotated application with respect to the legacy application due to time control block management.

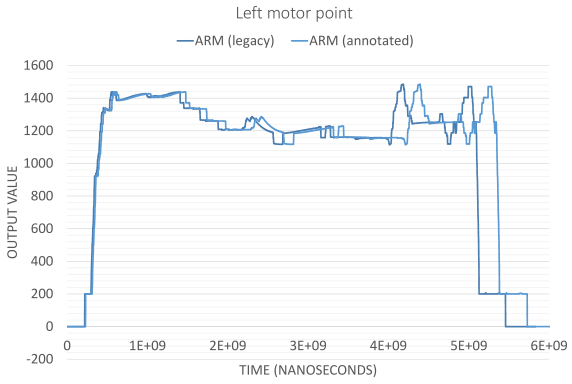


(a) Front motor point

(b) Rear motor point



(c) Right motor point



(d) Left motor point

Figure 21: Functional test results for the annotated multirotor application running on the ARM Cortex-A9 processor. (a) shows for each time step (in the X-axis) the corresponding value of motor front point variable (in the Y-axis), (b) shows for each time step (in the X-axis) the corresponding value of motor rear point variable (in the Y-axis), (c) shows for each time step (in the X-axis) the corresponding value of motor right point variable (in the Y-axis), and (d) shows for each time step (in the X-axis) the corresponding value of motor left point variable (in the Y-axis). Every graph, (a), (b), (c), and (d), show the results obtained for the legacy multirotor application (without annotations) as well as for the annotated multirotor application. On both test scenarios (legacy and annotated), the output value of control variables is equal, although there is a delay on the annotated application with respect to the legacy application due to time control block management. The average delay observed on the 2 ms control cycle over a 3000 time step simulation is 97 μs.

Table 6 – *Continued from previous page*

Time Trace	Contract limitation	Pass Fail
FET31.Exit 17023455 ns	[15845947,16445947] ns	✗
BET45.Entry 17404573 ns	[14500000,15000000] ns	✗
BET45.Exit 17816493 ns	[17404573,19404573] ns	✓
BET49.Entry 17987424 ns	[14700000,17000000] ns	✗
BET49.PEntry 0 ns	[0,0] ns	✓
BET49.Exit 18307373 ns	[17987424,20000000] ns	✓
PET20.Exit 20486905 ns	[19400000,20000000] ns	✗
PET20.Entry 20932500 ns	[19400000,20100000] ns	✗
FET21.Entry 21241476 ns	[20260897,20960897] ns	✗
BET22.Entry 21532219 ns	[20493058,21193058] ns	✗
BET22.Exit 22164893 ns	[21532219,26532219] ns	✓
BET26.Entry 22428261 ns	[16597033,26597033] ns	✓
BET26.PEntry 21241476 ns	[20260897,20960897] ns	✓
BET26.Exit 23046048 ns	[22428261,31241476] ns	✓
FET21.Exit 31745743 ns	[30641476,31241476] ns	✗
FET31.Entry 32165890 ns	[28345947,33945947] ns	✓
PNET36.Entry 32493916 ns	[29600000,30000000] ns	✗
PNET36.Exit 33168537 ns	[32493916,35493916] ns	✓
BET40.Entry 33432596 ns	[29133075,35233075] ns	✓
BET40.PEntry 32165890 ns	[28345947,33945947] ns	✓
BET40.Exit 34055520 ns	[33432596,37165890] ns	✓
FET31.Exit 37729554 ns	[36565890,37165890] ns	✗
BET45.Entry 38097877 ns	[34204573,40104573] ns	✓
BET45.Exit 38600576 ns	[38097877,40097877] ns	✓
BET49.Entry 38878960 ns	[34787424,40687424] ns	✓
BET49.PEntry 20932500 ns	[19400000,20100000] ns	✗
BET49.Exit 39354809 ns	[38878960,40932500] ns	✓
PET20.Exit 41428915 ns	[40332500,41032500] ns	✗

Multirotor application – annotation adjusted contracts.

Table 7 shows the time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the component-contract structure presented in Figure 19, adjusted as part of the timing enforcement assessment.

Table 7: Time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against (before translation) adjusted component-contract structure (see Figure 19). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET36.Entry 0 ns	[0,0] ns	✓
PET36.Exit 2124215 ns	[2000000,2100000] ns	✗
PET36.Entry 2541230 ns	[2000000,2200000] ns	✗
PET36.Exit 4868417 ns	[4541230,4641230] ns	✗
PET36.Entry 5146059 ns	[4541230,4741230] ns	✗
PET36.Exit 7643136 ns	[7146059,7246059] ns	✗
PET36.Entry 7935216 ns	[7146059,7346059] ns	✗
PET36.Exit 10450728 ns	[9935216,10035216] ns	✗
PET36.Entry 10743715 ns	[9935216,10135216] ns	✗
PET36.Exit 13303320 ns	[12743715,12843715] ns	✗
PET36.Entry 13596742 ns	[12743715,12943715] ns	✗
PET36.Exit 16099024 ns	[15596742,15696742] ns	✗
PET36.Entry 16501496 ns	[15596742,15796742] ns	✗
PET36.Exit 19017908 ns	[18501496,18601496] ns	✗

Continued on next page

Table 7 – *Continued from previous page*

Time Trace	Contract limitation	Pass Fail
PET36.Entry 19434240 ns	[18501496,18701496] ns	✗
PET36.Exit 21984387 ns	[21434240,21534240] ns	✗
PET36.Entry 22414182 ns	[21434240,21634240] ns	✗
PET36.Exit 24975024 ns	[24414182,24514182] ns	✗
PET36.Entry 25401653 ns	[24414182,24614182] ns	✗
PET36.Exit 27949025 ns	[27401653,27501653] ns	✗

The contracts presented in Figure 18, and Figure 19 had already been adjusted, however, they do not consider any timing variation cause by the overhead of the static translation process. Moreover, the code is now running on the new hardware platform, so time variations caused by the underlying OS and hardware platform itself, might vary. This caused many of the traces to fail the corresponding contract (almost 50% of the traces for the example application and every trace for the multirotor application). In order to cover the timing deviations caused by the static translation and the new hardware platform, contracts are adjusted as it was done before translation.

Example application – translation adjusted contracts.

Figure 22 shows the adjusted component-contract structure for the translated example application. Changes in the contracts with respect to (before translation) adjusted component-contract structure in Figure 18 are shown in bold. Up to time step 1400 adjustments had to be done in the contracts, then the example application runs for over 5000 time steps without failing any contract. Table 8 shows the time traces generated by the annotated example application running on the Intel Atom E3866 processor validated against the adjusted component-contract structure presented in Figure 22. Changes in the time interval with respect to the validation against (before translation) adjusted component-contract structure (see Table 6) are shown in bold.

Table 8: Time traces generated by the translated example application running on the Intel Atom E3866 processor validated against (after translation) adjusted component-contract structure (see Figure 22). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with (before translation) adjusted component-contract structure (see Table 6) are shown in bold. The third column shows contract pass/fail information.

Time Trace	Contract limitation	Pass Fail
PET20.Entry 0 ns	[0,0] ns	✓
FET21.Entry 860897 ns	[0,**900000**] ns	✓
BET22.Entry 1093058 ns	[0,**1100000**] ns	✓
BET22.Exit 1447619 ns	[1093058,6093058] ns	✓
BET26.Entry 1597033 ns	[0,5000000] ns	✓
BET26.PEntry 860897 ns	[0,**900000**] ns	✓
BET26.Exit 1966338 ns	[1597033,10860897] ns	✓
FET21.Exit 11138808 ns	[10260897,**11660897**] ns	✓
FET31.Entry 11445947 ns	[10000000,**11500000**] ns	✓
PNET32.Entry 11644891 ns	[10000000,**11700000**] ns	✓
PNET32.Exit 12063298 ns	[11644891,14644891] ns	✓
BET40.Entry 12233075 ns	[10000000,13000000] ns	✓
BET40.PEntry 11445947 ns	[10000000,**11500000**] ns	✓
BET40.Exit 12580840 ns	[12233075,16445947] ns	✓
FET31.Exit 17023455 ns	[15845947,**17345947**] ns	✓
BET45.Entry 17404573 ns	[14500000,**17500000**] ns	✓
BET45.Exit 17816493 ns	[17404573,19404573] ns	✓
BET49.Entry 17987424 ns	[14700000,**18000000**] ns	✓
BET49.PEntry 0 ns	[0,0] ns	✓
BET49.Exit 18307373 ns	[17987424,20000000] ns	✓
PET20.Exit 20486905 ns	[19400000,**21000000**] ns	✓
PET20.Entry 20932500 ns	[19400000,**21000000**] ns	✓
FET21.Entry 21241476 ns	[20260897,**22160897**] ns	✓
BET22.Entry 21532219 ns	[**17793058**,**26093058**] ns	✓
BET22.Exit 22164893 ns	[21532219,26532219] ns	✓
BET26.Entry 22428261 ns	[16597033,26597033] ns	✓
BET26.PEntry 21241476 ns	[20260897,**22160897**] ns	✓
BET26.Exit 23046048 ns	[22428261,31241476] ns	✓
FET21.Exit 31745743 ns	[30641476,**32041476**] ns	✓
FET31.Entry 32165890 ns	[28345947,33945947] ns	✓
PNET36.Entry 32493916 ns	[29600000,**32500000**] ns	✓
PNET36.Exit 33168537 ns	[32493916,35493916] ns	✓
BET40.Entry 33432596 ns	[29133075,35233075] ns	✓
BET40.PEntry 32165890 ns	[28345947,33945947] ns	✓
BET40.Exit 34055520 ns	[33432596,37165890] ns	✓
FET31.Exit 37729554 ns	[36565890,**38065890**] ns	✓
BET45.Entry 38097877 ns	[34204573,40104573] ns	✓
BET45.Exit 38600576 ns	[38097877,40097877] ns	✓
BET49.Entry 38878960 ns	[34787424,40687424] ns	✓
BET49.PEntry 20932500 ns	[19400000,**21400000**] ns	✓
BET49.Exit 39354809 ns	[38878960,40932500] ns	✓
PET20.Exit 41428915 ns	[40332500,**41932500**] ns	✓

Table 9: Time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against (after translation) adjusted component-contract structure (see Figure 23). The first column shows the time trace. The second column shows the valid time interval according to time traces and the corresponding contract, changes with respect to the validation with (before translation) component-contract structure (see Table 7) are shown in bold. The third column shows contract pass/fail information.

Multirotor application – translation adjusted contracts.

Figure 23 shows the adjusted component-contract structure for the translated multirotor application. Up to time step 1000 adjustments had to be done in the contract to cover the previously mentioned needs. The interval in the assumption is set to $[2000, 3600]$ ms, whereas the interval in the guarantee is set to $[2000, 3000]$ us. Then, the multirotor application runs for over 3000 time steps without any further failure. Table 9 shows the time traces generated by the translated flight control application running on the Intel Atom E3866 processor validated against the adjusted component-contract structure presented in Figure 23. Changes in the time interval with respect to the validation against (before translation) adjusted component-contract structure (see Table 7) are shown in bold.

Time Trace	Contract limitation	Pass Fail
PET36.Entry 0 ns	[0,0] ns	✓
PET36.Exit 2124215 ns	[2000000,**3000000**] ns	✓
PET36.Entry 2541230 ns	[2000000,**3600000**] ns	✓
PET36.Exit 4868417 ns	[4541230,**5541230**] ns	✓
PET36.Entry 5146059 ns	[4541230,**6141230**] ns	✓
PET36.Exit 7643136 ns	[7146059,**8146059**] ns	✓
PET36.Entry 7935216 ns	[7146059,**8746059**] ns	✓
PET36.Exit 10450728 ns	[9935216,**10935216**] ns	✓
PET36.Entry 10743715 ns	[9935216,**11535216**] ns	✓
PET36.Exit 13303320 ns	[12743715,**13743715**] ns	✓
PET36.Entry 13596742 ns	[12743715,**14343715**] ns	✓
PET36.Exit 16099024 ns	[15596742,**16596742**] ns	✓
PET36.Entry 16501496 ns	[15596742,**17196742**] ns	✓
PET36.Exit 19017908 ns	[18501496,**19501496**] ns	✓
PET36.Entry 19434240 ns	[18501496,**20101496**] ns	✓
PET36.Exit 21984387 ns	[21434240,**22434240**] ns	✓
PET36.Entry 22414182 ns	[21434240,**23034240**] ns	✓
PET36.Exit 24975024 ns	[24414182,**25414182**] ns	✓
PET36.Entry 25401653 ns	[24414182,**26014182**] ns	✓
PET36.Exit 27949025 ns	[27401653,**28401653**] ns	✓

Result analysis.

Result show that formal timing specifications adjusted as part of the timing enforcement assessment needed to be re-adjusted to cover the timing deviations generated due to translation overhead. After the re-adjustment of formal timing specifications, time traces generated at runtime by the translated example and multirotor applications fit into the defined timing contracts. However, the static translation process involves a considerable overhead (analysed through a feasibility study by Yarza et al. (2020)) which might not be affordable depending on the application to be ported. For each particular case, an expert should assess, considering the incurred overhead, whether the RT legacy software migration solution is acceptable. Future work considers optimizing time control blocks to, at some point, overcome the block management and translation overhead.

7.2.2. Functional test

The functional test compares the reference output values for control variables with the output value (for those control variables) when running the systematically annotated and then translated applications (example – see Listing 1, and multirotor

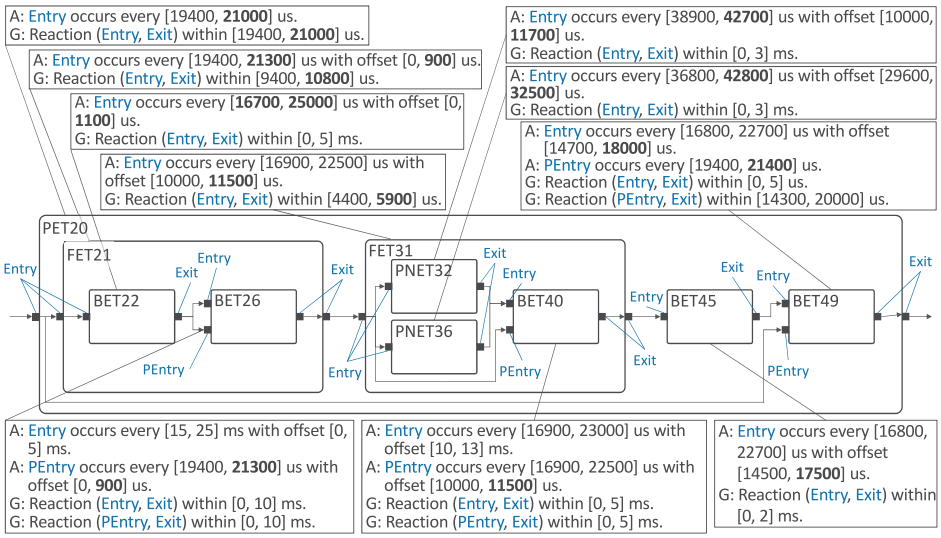


Figure 22: Adjusted component-contract structure for the translated example application. Each component has its corresponding contract. Contracts describe the assumptions and guarantees observable at input/output ports (named in blue colour). Contracts have been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the translated example application running on the Intel Atom E3866 processor. Changes in the contracts with respect to (before translation) adjusted component-contract structure in Figure 18 are shown in bold.

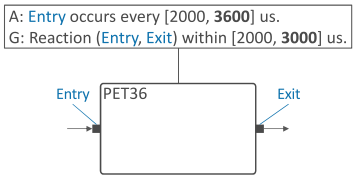


Figure 23: Adjusted component-contract structure for the translated multirotor application. The only component (PET36) has its corresponding contract. The contract describes the assumption and guarantee observable at Entry/Exit ports (named in blue colour). The contract has been adjusted to cover possible jitter present on a real scenario, in this case adjustments are applied for the translated multirotor application running on the Intel Atom E3866 processor.

– see Listing 2) on top of the Intel Atom E3866 processor, using reference input data for state variables [5].

Example application.

Figure 24 shows the bit toggling sequence of the output variable on both test scenarios. However, due to scaling problems, the figure depicts just the output variable value for the first 100 time steps.

Multirotor application.

Figure 25 shows the output values for each of the multirotor control variables on both test scenarios. Control variables are observed for a 3000 time step interval (about 6 seconds simulation).

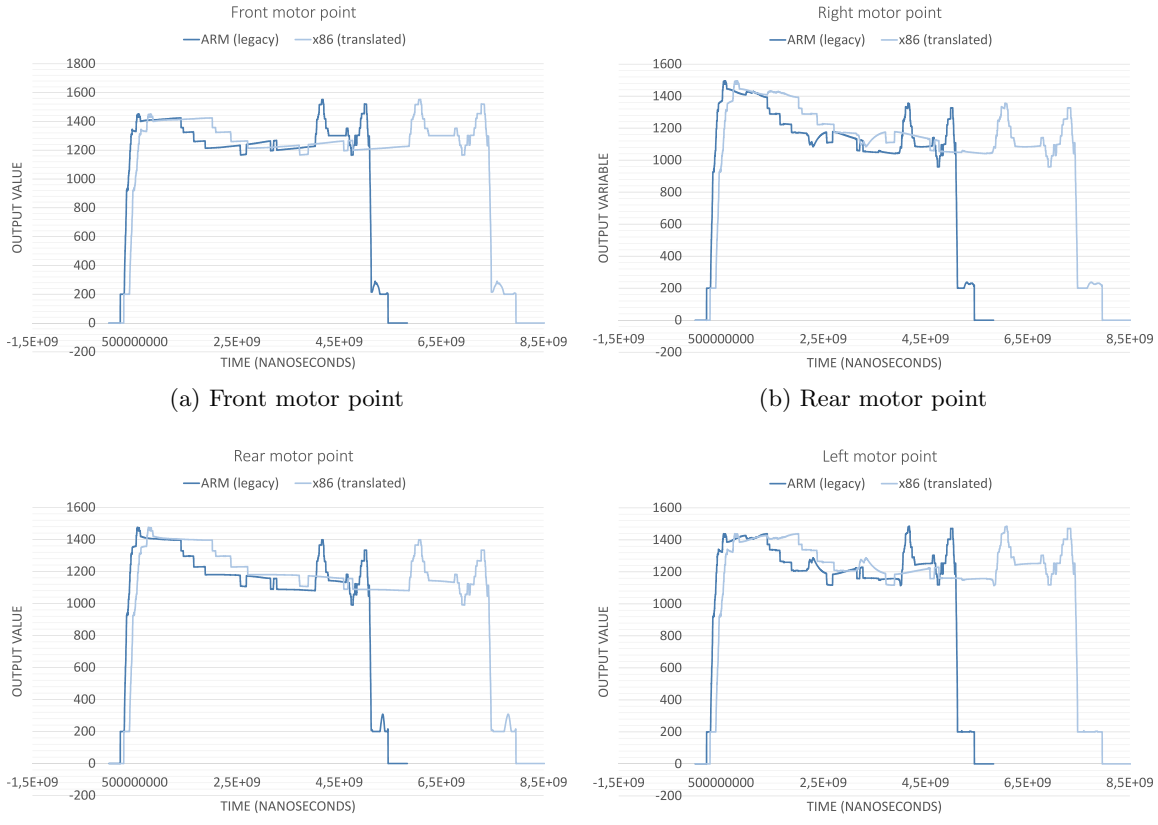
Result analysis.

Results show that the signal observed on every control variable is equivalent (in value) before and after the timing-aware migration process. However, due to time control block management and static translation overhead, a delay (with respect to the legacy platform) is observable on every control variable output signal. This delay in the response of the controller might disturb the functional behaviour of the control system, therefore, for each particular case a further analysis (on a more realistic scenario) would be necessary to determine whether the functional behaviour is acceptable after the RT legacy software migration process. Future work considers providing support to port legacy platform I/O port dependent code, which will provide means to perform a functional test on a more realistic scenario.

8. Conclusion

This research work, first, reasons about the need for a portable legacy software migration solution that preserves the timing as well as the functional behaviour of the retargeted application. In the direction to cover this gap, a RT legacy software mi-

Figure 24: Functional test results for the translated example application running on the Intel Atom E3866 processor. For each time step (X-axis) the corresponding output variable value (Y-axis) is shown, both, for the legacy example application (without annotations) as well as for the translated example application. On both test scenarios (legacy and translated), the bit toggling sequence is equal, although there is a delay on the translated application with respect to the legacy application due to time control block management and static translation overhead.



(a) Front motor point

(b) Rear motor point

(c) Right motor point

(d) Left motor point

Figure 25: Functional test results for the translated multirotor application running on the Intel Atom E3866 processor. (a) shows for each time step (in the X-axis) the corresponding value of motor front point variable (in the Y-axis), (b) shows for each time step (in the X-axis) the corresponding value of motor rear point variable (in the Y-axis), (c) shows for each time step (in the X-axis) the corresponding value of motor right point variable (in the Y-axis), and (d) shows for each time step (in the X-axis) the corresponding value of motor left point variable (in the Y-axis). Every graph, (a), (b), (c), and (d), show the results obtained for the legacy multirotor application (without annotations) as well as for the annotated and the translated multirotor application. On both test scenarios (legacy and translated), the output value of control variables is equal, although there is a delay on the translated application with respect to the legacy application due to time control block management and static translation overhead. The average delay observed on the 2 ms control cycle over a 3000 time step simulation is 922 μs.

gration solution is proposed, which is based on existing static binary translation solution enhanced with a timing enforcement mechanism that at the same time provides means for validating the enforced timing behaviour. The proposed solution is then evaluated, through multiple legacy applications (example and multirotor). Legacy code is ported from the ARM Cortex-A9 processor to the Intel Atom E3866.

The timing-aware migration assessment concludes that the defined temporal constructs provide means to enforce a specific timing behaviour. The enforces timing behaviour can then be validated on the new hardware platform combining the use of the time traces that temporal constructs generate at runtime, systematically generated formal timing specifications, and MULTIC tool. Results show that although timing specifications needed to be relaxed such that they reflect time uncertainties generated by time control block management, the static translation process as well as the new hardware platform itself, it is possible to achieve a timing behaviour equivalent to that in the legacy system.

Future work considers lifting I/O dependent code and implementing an I/O virtualization mechanism to provide the ported legacy application means to interact with the external environment when running on the new hardware platform. Moreover, in order to overcome block management overhead, a characterization phase should be accomplished for each particular migration case to adjust the temporal constructs accordingly.

References

Abdellatif, T., Combaz, J., Sifakis, J., 2013. Rigorous implementation of real-time systemsfrom theory to application. Mathematical Structures in Computer Science 23, 882–914.

Accellera, 2019. Homepage of the accellera systems initiative. http://www.accellera.org.

Bde, E., Bker, M., Damm, W., Ehmen, G., Frnzle, M., Gerwinn, S., Goodfellow, T., Grttner, K., Josko, B., Koopmann, B., Peikenkamp, T., Poppen, F., Reinkemeier, P., Siegel, M., Stierand, I., 2017. Design paradigms for multi-layer time coherency in adas and automated driving (multic), in: FAT-Schriftenreihe 302. 302 ed.. Forschungsvereinigung Automobiltechnik e.V. (FAT). FAT-Schriftenreihe. URL: https://www.vda.de/en/services/Publications/fat-schriftenreihe-302.html.

Bde, E., Damm, W., Ehmen, G., Frnzle, M., Grttner, K., Ittershagen, P., Josko, B., Koopmann, B., Poppen, F., Siegel, M., Stierand, I., 2019. Multic-tooling, in: FAT-Schriftenreihe 316. 316 ed.. Forschungsvereinigung Automobiltechnik e.V. (FAT). FAT-Schriftenreihe. URL: https://www.vda.de/de/services/Publikationen/fat-schriftenreihe-316.html.

Bellard, F., 2005. Qemu, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, pp. 41–46.

Bennett, K., 1995. Legacy systems: Coping with success. IEEE software 12, 19–23.

Bruns, F., Ittershagen, P., Grttner, K., 2019. Timing measurement and control blocks for bare-metal c++ applications, in: Forum on Specification and Design Languages (FDL).

Cifuentes, C., Emmerik, M.V., 2000. Uqbt: adaptable binary translation at low cost. Computer 33, 60–66. doi:10.1109/2.825697.

Cogswell, B., Segall, Z., 1995. Timing insensitive binary to binary translation of real time systems, in: Workshop on Architectures for Real-Time Applications, ISCA.

Falk, H., Lokuciejewski, P., 2010. A compiler framework for the reduction of worst-case execution times. Real-Time Systems 46, 251–300.

Federico, A.D., Payer, M., Agosta, G., 2017. rev.ng: a unified binary analysis framework to recover cfgs and function boundaries, in: Proceedings of the 26th International Conference on Compiler Construction, ACM, 3033028. pp. 131–141. doi:10.1145/3033019.3033028.

Gehani, N., Ramamritham, K., 1991. Real-time concurrent c: A language for programming dynamic real-time systems. Real-Time Systems 3, 377–405. URL: https://doi.org/10.1007/BF00365999, doi:10.1007/bf00365999.

Heinz, T., 2008. Preserving temporal behaviour of legacy real-time software across static binary translation, in: Proceedings of the 1st workshop on Isolation and integration in embedded systems, ACM. pp. 1–4.

Henzinger, T.A., Kirsch, C.M., 2007. The embedded machine: Predictable, portable real-time code. ACM Transactions on Programming Languages and Systems (TOPLAS) 29, 33.

Hwang, Y.S., Lin, T.Y., Chang, R.G., 2010. Disirer: Converting a retargetable compiler into a multiplatform binary translator. ACM Transactions on Architecture and Code Optimization (TACO) 7, 18.

Lattner, C., Adve, V., 2004. Llvm: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society. p. 75.

Le Nabec, B., Hedia, B.B., Babau, J.P., Jan, M., Guesmi, H., 2016. Modeling legacy code with bip: how to reduce the gap between formal description and real-time implementation, in: 2016 Forum on Specification and Design Languages (FDL), IEEE. pp. 1–8.

Natarajan, S., Broman, D., 2018. Timed c: An extension to the c programming language for real-time systems, in: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE. pp. 227–239.

OFFIS, 2019. Multi-rotor demonstrator. https://multirotor.offis.de/wordpress/.

Resmerita, S., Naderlinger, A., Huber, M., Butts, K., Pree, W., 2015. Applying real-time programming to legacy embedded control software, in: 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, IEEE. pp. 1–8.

SAFEPOWER, C., 2017. D4.6 final cross-domain public demonstrator URL: http://safepower-project.eu/consultant_project/mortgage-advisor-2-2/.

Shen, B.Y., Chen, J.Y., Hsu, W.C., Yang, W., 2012. Llbt: an llvm-based static binary translator, in: Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems, ACM. pp. 51–60.

Ung, D., Cifuentes, C., 2000. Machine-adaptable dynamic binary translation, in: ACM SIGPLAN Notices, ACM. pp. 41–51.

Wagner, C., Wagner, C., 2014. Model-Driven Software Migration. Springer.

Wahler, M., Eidenbenz, R., Franke, C., Pignolet, Y.A., 2015. Migrating legacy control software to multi-core hardware, in: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, pp. 458–466. doi:10.1109/ICSM.2015.7332497.

Wu, B., Lawless, D., Bisbal, J., Grimson, J., Wade, V., O'Sullivan, D., Richardson, R., 1997. Legacy system migration: A legacy data migration engine, in: Proceedings of the 17th International Database Conference (DATASEM'97), pp. 129–138.

Yang, Y., Guan, H., Zhu, E., Yang, H., Liu, B., 2010. Crossbit: a multi-sources and multi-targets dbt.

Yarza, I., Azkarate-askatsua, M., Onaindia, P., Grüttner, K., Ittershagen, P., Nebel, W., 2020. Static/dynamic real-time legacy software migration - a comparative analysis, in: Proceedings of the Rapido'20 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3375246.3375257, doi:10.1145/3375246.3375257.