A Framework for Building Verifiable Scalable Embedded Systems Interfacing through Sensors and Actuators

Sensors & Transducers, Vol. 265, Issue 2, May 2024, pp. 27-39

www.sensorsportal.com

Sensors & Transducers

Published by IFSA Publishing, S. L., 2024
http://www.sensorsportal.com

A Framework for Building Verifiable Scalable Embedded Systems Interfacing through Sensors and Actuators

[1, *] Morgan MCCOLL, [1] Callum MCCOLL, [1] Gervase TUXWORTH, [2] Aaron PEREIRA and [1] Rene HEXEL

[1] Griffith University, 170 Kessels Rd, Nathan, 4111, Australia
[2] Sub mm-wave Technology Group, Jet Propulsion Laboratory, NASA/Caltech, Pasadena, CA, USA
[*] E-mail: morgan.mccoll@griffithuni.edu.au

Received: 5 April 2024 / Accepted: 8 May 2024 / Published: 30 May 2024

Abstract: Embedded systems are at the core of modern industrial applications that interface with the environment through sensors and actuators. As software-defined modelling revolutionises the development of safety-critical systems, such as self-driving cars in the automotive industry, rapid turnaround and integration into cloud-based software development services are increasingly becoming critical. Recent parts shortages and supply-chain constraints have highlighted the importance of dynamic, cross-platform codebases that allow scalability and quick re-deployment to different hardware. In this paper, we show the ability of our development framework to create decomposable, embedded systems that consist of software that factors out hardware dependencies and can thus be easily ported and deployed to multiple hardware architectures. We demonstrate how our embedded cross framework allows us to decouple hardware-specifics from the requirements for the software that implements the behaviour of the system. To this end, we show how to design and build scalable software that integrates with multiple hardware architectures, operating systems, and middleware for embedded systems. For the first time, we not only show how such systems can be developed for microcontrollers, but how the same embedded cross framework can be utilised for Field-Programmable Gate Arrays (FPGAs). We demonstrate how software systems that utilise our framework can seamlessly integrate with continuous integration and continuous deployment (CI/CD) processes. This allows the flexibility of testing and integration using local and cloud-based systems, as well as end-to-end hardware-in-the-loop approaches.

Keywords: Cross compilation, CI/CD, Hardware abstraction, Build system, Safety-critical systems, Embedded software.

1. Introduction

Embedded systems that utilise microcontrollers and FPGAs are ubiquitous when developing systems that interact with their physical environment utilising sensors and actuators. These systems often perform safety-critical tasks or operate within remote and harsh environments that require a high level of dependability. This creates the need for robust and reliable designs. The promise of Model-Driven Development (MDD) is to enable reliable designs at a high level [1] that can be translated into software that runs on a target system, often using the concept of Executable Models that automate this translation while staying true to the intent and semantics of the high-level design [2, 3]. However, this is in stark contrast to the realities of software architectures for embedded platforms that often utilise limited high-level abstractions and exhibit a prevalence of application-specific and tightly coupled code. This limits robustness and flexibility (as, e.g., provided by dynamic codebases often used in systems in stable environments with reliable connectivity to the internet) [4, 5].

Furthermore, these realities contradict the ethos of modern software engineering principles and methods such as agile development, where software is iteratively developed and updated post-release to accommodate evolving requirements and a staged rollout [6]. A dynamic codebase should support

runtime decoupling [7, 8] and modularity [9] to effortlessly integrate with multiple, diverse software architectures and hardware platforms [10]. The prevalent use, to this date, of a mixture of programming constructs that are both application-specific as well as hardware-specific often requires the use of outdated approaches, such as a waterfall or hybrid development model that requires distinct stages of detailed, upfront planning, development, testing, and hardware integration [11].

By contrast, what is considered best practice in software development today importantly includes the ability to create reusable components that are independent of and abstract away the specifics of a particular hardware architecture [12]. Moreover, to be able to develop composable systems that remain loosely coupled, it is imperative that developers reuse any tried, verified, and tested components [9], regardless of whether these components are specific to a particular target system or whether they are cross-platform and portable. Examples of the former are vendor-specific hardware abstraction layers, drivers, solution stack and protocol implementations. Examples of portable, cross-platform components and environments include open-source operating systems and frameworks, such as FreeRTOS [13], or implementations of CMSIS [14]. In this paper, we showcase our embedded cross framework [15], a versatile software development framework for embedded systems that facilitates the segregation of hardware-specific, application-specific, and reusable code. Our framework builds on technology utilised widely in the software development community and allows support for new architectures as well as the seamless integration of existing codebases in a reusable fashion. We not only demonstrate how to use the embedded cross framework with microcontrollers, but also introduce the ability to create software for Field-Programmable Gate Arrays (FPGAs) utilising the same software engineering techniques and workflows.

The rest of this paper is structured as follows. Section 2. gives an overview of frameworks and development environments for embedded systems software development. We introduce our embedded-cross framework in Section 3, explaining the structure of projects, firmware, operating systems, libraries, and frameworks, as well as their usage with microcontrollers and FPGAs. We demonstrate the effectiveness of the embedded-cross framework through a simple, bare-metal microcontroller case study in Section 4. In Section 5, we show how to use the embedded-cross framework with an FPGA before demonstrating a hardware-agnostic solution in Section 6. We discuss our findings in Section 7 and present our conclusions in Section 8.

2. Background

Numerous different attempts have been made to facilitate cross-platform and hardware-independent development for embedded systems. Among these, Arduino [16] has gained prominence as an Integrated Development Environment (IDE) by providing a standardised Application Programming Interface (API) across multiple microcontroller families. The Arduino API provides a unified Hardware Abstraction Layer (HAL) for embedded platforms, allowing developers to leverage data types and function calls from the Arduino API that separate hardware-specific code from business logic. The widespread support for this approach led many developers to use the GNU Compiler Collection (`gcc`) [17] for cross-compilation, utilising the same Arduino API across distinct hardware platforms and designs. However, the monolithic design of the Arduino IDE exposed its limitations, leading to the emergence of newer, more modular frameworks such as PlatformIO [18].

While PlatformIO acts as an Integrated Development Environment (IDE), its Graphical User Interface (GUI) is realised through plug-ins and extensions [19] to popular editors and development environments such as Visual Studio Code [20], Eclipse [21], Qt Creator [22], Vim [23], etc. More importantly, at its core, PlatformIO is a cross-platform, cross-architecture framework for embedded systems engineers and software developers to write applications for embedded products [18]. It provides the means to incorporate different boards, microcontrollers, and compilers, and may be invoked from the command line, allowing a more seamless integration into Continuous Integration and Continuous Deployment (CI/CD) pipelines. The main limitation of PlatformIO is its bespoke build system that limits the use of existing projects that utilise more versatile build environments. The Arduino API [24] is a further point of concern, as its primary intention was that of an educational tool for students to learn embedded system development, not the development of dependable embedded systems or automation for advanced industrial use. Inspecting the implementation of the Arduino API on different hardware reveals that implementations are typically not focusing on robust programming and error-checking or recovery techniques. Well-established frameworks for quality software, such as MISRA-C [25], are not even on the radar for vendors supporting this API.

The Robot Operating System (ROS) operates at a different level [26]. While it offers a middleware that provides sophisticated algorithms for high-level, robotic tasks, such as planning, machine learning, vision, and inter-process communication, its focus is on utilising a sophisticated desktop-grade Operating System such as Linux, with only limited support for embedded systems utilising real-time Operating Systems, bare-metal hardware [27], or FPGAs that we are focusing on here. Other approaches for embedded systems [28] try to create custom libraries by generating HALs, drivers, start-up code, and implementations of common communication protocols via configuration files specified in XML.

While the goal of this approach is laudable, it requires a lot of boilerplate and upfront work supporting a particular embedded system, duplicating many features already commonplace in modern build systems such as *Cmake* [29].

When it comes to FPGAs, most mainstream development happens on proprietary IDEs provided by the hardware manufacturer. Unlike many embedded systems IDEs that nowadays are predominantly derived from the Eclipse Open-Source project, the FPGA IDEs are typically closed-source and focus on a GUI-dominated workflow.

A similar scenario presents itself when it comes to high-level modelling on FPGAs. Frameworks exist that translate high-level language source code to microcontroller representations that run on soft cores on the FPGA [30], but when it comes to efficient representations, these are typically relegated to hardware description languages (HDLs) such as Verilog and VHDL. This is largely due to the serial nature of imperative programming languages typically used in mainstream and embedded software development, although attempts exist to create a more readily parallelisable and efficient FPGA representation for programming paradigms, such as pure functional programming utilising immutable data structures [31].

Model-driven development on FPGAs has traditionally faced challenges for similar reasons, largely due to the event-driven nature of most modelling languages, that is ill-fitted towards the state-based nature of FPGA components [32-38]. This challenge is compounded when performing formal verification in embedded systems as mathematical formulae often ignore the limitations of physical hardware by placing assumptions on runtime behaviour [39-42]. In FPGAs, the problem becomes even more troublesome as synthesis can introduce semantic differences in favour of runtime efficiency and placement optimisations, invalidating preconceived notions of execution semantics [43, 44]. The most accurate form of execution requires one-to-one mappings of models into embedded hardware, preserving formal semantics within embedded structures. For safety-critical systems, formal verification must occur on a system that represents the exact semantics at runtime, typically through methods such as runtime verification [45, 46] of the final executable. Previously, we have demonstrated that creating high-level, executable models for FPGAs is possible [47] using Logic-Labelled Finite-State Machines (LLFSMs) [48]. We have shown that such a model provides the one-to-one mapping required for accurate formal verification at runtime [49, 50]. Nevertheless, these approaches still focus on the generation of low-level HDL representations that then need to be synthesised using proprietary manufacturer frameworks. Here, we describe how we can utilise our embedded-cross framework to not only design and implement microcontroller-based embedded systems that integrate with modern build systems such as *cmake*, but also how the same framework can be used in the same fashion for the development of FPGA-based systems and integration into CI/CD workflows.

3. The Embedded-cross Framework

We now present the structure and use of our *embedded-cross framework* [15, 51]. This framework focuses on using mainstream software development tools and frameworks to reap the benefits of existing hardware abstraction layers and middleware without requiring extensive boilerplate and configuration. We achieve this while maintaining compatibility with external libraries and projects that utilise other build systems. To this end, the embedded-cross framework utilises Cmake Presets [52] that have been available since *cmake* 3.19. These allow us to specify the configuration options for cross-compilation from the host system to the embedded system [52]. In particular, the `configurePresets` section tells us which cross compiler and target to use (e.g. `arm-none-eabi-gcc`). The section is subdivided into a base configuration (e.g. `arm-none-eabi-gcc-base`) and common configurations such as `debug` and `release` (e.g. `arm-none-eabi-gcc-debug` and `arm-none-eabi-gcc-release`). This configuration allows us to support any number of different compilers and target architectures, as long as the corresponding executables can be found in the search `PATH` of the system. Out of the box, the embedded-cross framework currently supports the C, C++, and Swift programming languages utilising either `gcc` [17], `clang` [53], or `swiftc` [54] as the cross-compiler for the selected microcontroller architecture. Additional compilers and target architectures can be added by either modifying the `CmakePresets.json` file or by creating a separate `CmakeUserPresets.json` file.

Let us now explore the structure of the embedded- cross framework and how we can utilise this to create reusable components and projects. Fig. 1 shows the folder structure of the framework.

```
├── boards
├── cmake
├── firmware
├── middleware
├── projects
└── toolchains
```

Fig. 1. The directory structure of the embedded-cross framework.

The `cmake` folder contains the core infrastructure of the embedded-cross framework that handles compilation, target dependencies and provides the infrastructure for the main `Cmakelists.txt` as well as the *cmake* fragments in the other folders.

The `toolchains` folder contains a number of subfolders, one for each combination of compiler and target architecture. For example, `arm-none-eabi-clang` denotes the clang compiler for cross-compiling to the arm architecture. Each subfolder contains a `toolchain.cmake` file that configures the compiler and associated programs (such as assembler and linker), their command-line options and common flags, as well as related locations of toolchain-specific files. Additional files to support a particular toolchain can also be stored underneath the folder that contains the `toolchain.cmake` file.

The `firmware` folder contains the manufacturer-specific files for their target hardware as well as operating systems, such as FreeRTOS [13], ported to that hardware. These follow a two-layer structure grouped by vendor (e.g. an `ST` folder for firmware and development frameworks related to STMicroelectronics specific hardware, and similar for other vendors). To decouple the embedded-cross framework from these vendor-specific files, the folders containing the specific support files are not embedded directly in the `git` [55] file structure. Instead, these are added as git submodules [56] to de-couple the release cycles of the embedded-cross framework from those of the vendors.

A similar folder, `middleware`, allows for software libraries that sit on top of the firmware and hardware abstraction layers or operating system. Examples for middleware are inter-process communication protocols such as a whiteboard [57], networking stack implementations such as CANopenNode [58], or compiler runtimes such as µSwift [59].

The specifics on how to compile code for a particular hardware configuration, e.g., the combination of microcontroller and peripherals present on a system development board, are configured in subfolders contained in the `boards` folder. The `boards` folder follows a similar 2-level folder structure, grouping similar boards underneath a manufacturer or architecture subfolder (e.g. `TI` for Texas Instruments boards). Custom boards also get a 2nd-level folder. Each board needs to define a `board.cmake` file containing the specifics of how to compile for that particular board (e.g. the compiler and target architecture to use and references to firmware, hardware abstraction layers, operating systems (if any), middleware, and libraries available for the hardware connected to or present on the board.

Finally, the `projects` folder contains any projects that utilise the embedded-cross framework. These are again organised in a 2-level structure, with the upper level denoting the operating system that is used for the project (e.g. `free_rtos` for FreeRTOS), or whether the final program will run on bare metal, in which case the project will be placed in the `bare_metal` subfolder. Similar to the vendor folders, the actual projects are not part of the embedded-cross framework, but are usually organised as separate repositories that are checked out independently.

The advantages of our framework include the ability to continually modify, compile, test, and deploy software during the development process. The compilation command-line interface (CLI) is identical for all target architectures ranging from embedded microcontrollers to more sophisticated CPUs and reconfigurable architectures such as FPGAs. This methodology is compatible with version control tools such as *git* [55], deploying via remote environments such as *GitHub* [60] and *Gitlab* [61].

To integrate new code into a large project, the user can simply add a folder to the `projects` subdirectory where the new code will live. This approach also allows the further integration of existing repositories by using git submodules. Integrating code in this manner creates a simple approach to targeting new hardware that is already supported by the project. The hardware abstraction itself exists within the other subdirectories and, once implemented, can be seamlessly targeted by new code. We have used this structure to create systems targeting a Xilinx Zynq7000 System-on-Chip (SoC), sensor abstraction of the Bosch BMP280 barometric sensor, and ARM-based microcontrollers such as the STM32 and TMS570 range.

3.1. Supporting FPGA and SoC Architectures

Previously, we have discussed the embedded-cross framework's ability to support FPGA and SoC architectures without discussing the details of such types of compilation. Due to the nature of these architectures, compilation (and synthesis) presents a unique challenge as typical compilation methodology does not support these distinct architectures.

The main difference between typical compilation and FPGA compilation is the absence of instruction-based binaries that execute within a processor platform. In FPGAs, the binary files describe a mapping of hardware-logic that is routed through a configurable structure. This compilation strategy is vastly different to microcontroller or processor-based compilation with FPGAs requiring different tools and custom make directives on top of native *cmake*. The advantage of the embedded-cross framework is the abstraction of this entire problem by maintaining API stability between the *cmake* command-line interface previously discussed and the commands required for FPGA synthesis. In laymen's terms, the exact process for performing FPGA synthesis is the same as any other project within the embedded-cross framework – the project is first configured with a configuration preset and built using a build preset.

We begin by defining the project structure for FPGA synthesis. Within the root directory of the embedded-cross framework exists a `projects` subdirectory containing projects for bare-metal and other embedded implementations. FPGA projects

exist within the `fpga` subdirectory in this folder. An FPGA project requires two things at a minimum, the first being a `project.cmake` that defines the language and sources of the project, and a folder containing the HDL project in *vivado*. By default, the vivado project should exist within a folder called `vivado_project` but may be configured by overwriting the *cmake* variable `${CMAKE_PROJECT_NAME}_PROJECT_DIRECTORY_NAME`. This definition may be created within the `project.cmake` file. The other important difference is that the languages within `project.cmake` are *HDL* – a custom language that only exists within the embedded-cross framework. We will present a case study with the files present for a simple FPGA example in Section 5. With these files created correctly, the user may now use *vivado* to create HDL implementations on a development computer and successfully compile using the embedded-cross frameworks CLI.

4. Bare Metal Case Study

While *cmake* gives us the flexibility to choose the project structure we would like to use for a project, this is not always the case with IDEs provided by embedded systems vendor, as they are often closed source or utilise open source development environments, such as Eclipse [21], that do not use *cmake*. This is not a problem, of course, if we create our own projects from scratch that use the embedded-cross framework with IDEs that fully support *cmake*, such as Visual Studio Code [20]. The downside, however, of creating projects from scratch without using the manufacturer IDE is that we may not be able to fully benefit from the simplicity of utilising these tools, particularly when configuring and starting a new project.

We will now demonstrate how to use the embedded-cross framework by utilising a workflow that benefits from both the ease of use of vendor tools as well as the tools available in the embedded-cross framework. For demonstration purposes, we create a project for an STM32 Nucleo-144 F207zg development board using the STM32CubeIDE version 1.14.0 (CubeIDE), which is based on Eclipse. The steps are very similar for other vendor IDEs, particularly those that build on the Eclipse platform as well.

As our first step, we clone our embedded-cross framework from GitHub:

```
git clone https://github.com/mipalgu/
    embedded-cross-framework.git
```

We then start a new project from the IDE ("Start new STM 32 project" button in the CubeIDE), filter for Nucleo-144 in the Board Filters, and select NUCLEO-F207ZG under Commercial Part No. We create a project named `STM32GPIO` in a folder of the same name inside the `projects/bare_metal` folder inside the checked out `embedded-cross-framework` repository (overriding the default location in the CubeIDE), initialising all peripherals with their default mode and opening the device configuration tool. There, we configure our peripherals as needed (in our example, we leave the defaults for the board, but disable ethernet to ensure the generated code will not block without a network connection). We then close the `STM32GPIO.ioc` file, saving the configuration changes we made and tell the IDE to generate Code and open the C/C++ perspective. In the IDE, we can now see the program structure shown in Fig. 2.

This project should now compile and work on the target hardware, but do nothing. If we add the following code to our `main.c`, we should get a blinking, blue LED (`LD1` on port `B0`):

```
if (HAL_GetTick() % 500 = = 0) {
    HAL_GPIO_TogglePin(LD1_GPIO_Port,
     LD1_Pin);
}
```

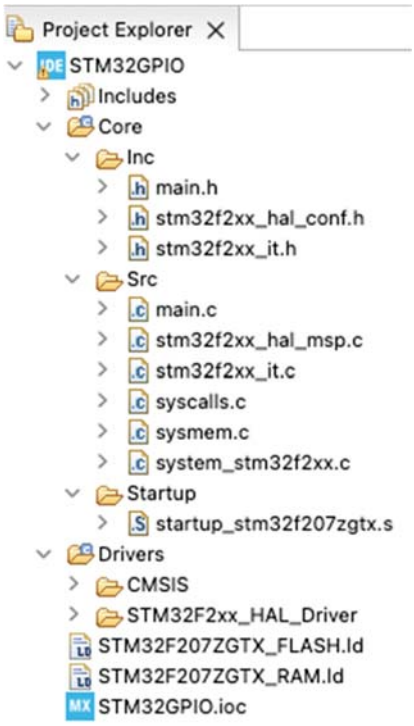

Fig. 2. Original CubeIDE Project Structure.

To make this project build with our embedded-cross framework, we first need to install the ARM GNU Toolchain, making sure the compiler can be found in the command line PATH. Then we can create our `STM32GPIO/project.cmake` fragment to configure our project (Fig. 3).

We can now configure and build our program using *cmake* from inside the embedded-cross-framework folder (Fig. 4).

These commands now create the same target, but use the original ARM GNU Toolchain instead of the CubeIDE. As we are now using a standard *cmake* structure, we can easily reconfigure our project to use a different compiler. For example, if we have a working installation of the clang compiler, we can now simply build our project by replacing `gcc` with `clang` as shown in Fig. 5.

```
set(${project_name}_VERSION_MAJOR 1)
set(${project_name}_VERSION_MINOR 0)
set(${project_name}_VERSION_PATCH 0)

# Core directories
set(${project_name}_CORE_DIR ${${project_name}_DIR}/Core)
set(${project_name}_CORE_SRCDIR ${${project_name}_CORE_DIR}/Src)
set(${project_name}_CORE_INCDIR ${${project_name}_CORE_DIR}/Inc)

# Driver directories
set(${project_name}_DRIVER_DIR ${${project_name}_DIR}/Drivers)
set(${project_name}_HAL_DIR ${${project_name}_DRIVER_DIR}/STM32F2xx_HAL_Driver)
set(${project_name}_HAL_SRCDIR ${${project_name}_HAL_DIR}/Src)
set(${project_name}_HAL_INCDIR ${${project_name}_HAL_DIR}/Inc)
set(${project_name}_CMSIS_DIR ${${project_name}_DRIVER_DIR}/CMSIS/Device/ST/STM32F2xx)
set(${project_name}_CMSIS_SRCDIR ${${project_name}_CMSIS_DIR}/Source)
set(${project_name}_CMSIS_INCDIR ${${project_name}_CMSIS_DIR}/Include)

# Sources for the project.
set(${project_name}_SOURCES
    ${${project_name}_CORE_SRCDIR}/main.c
    ${${project_name}_CORE_SRCDIR}/syscalls.c
    ${${project_name}_CORE_SRCDIR}/sysmem.c
    ${${project_name}_CORE_SRCDIR}/system_stm32f2xx.c
    ${${project_name}_CORE_SRCDIR}/stm32f2xx_it.c
    ${${project_name}_CORE_SRCDIR}/stm32f2xx_hal_msp.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_cortex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_dma.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_dma_ex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_flash.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_flash_ex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_gpio.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_pcd.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_pcd_ex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_pwr.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_rcc.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_rcc_ex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_tim.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_tim_ex.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_hal_uart.c
    ${${project_name}_HAL_SRCDIR}/stm32f2xx_ll_usb.c
)

# Include directories for the project.
set(${project_name}_INCDIR
    ${${project_name}_CMSIS_INCDIR}
    ${${project_name}_HAL_INCDIR}
    ${${project_name}_CORE_INCDIR}
    ${${project_name}_CANOPEN_INCDIRS}
)

# Linker script and startup file for the project.
set(${project_name}_nucleo_f207zg_LINKER_SCRIPT
${${project_name}_DIR}/STM32F207ZGTX_FLASH.ld)
set(${project_name}_nucleo_f207zg_STARTUP_SRC
${${project_name}_CORE_DIR}/Startup/startup_stm32f207zgtx.s)
```

Fig. 3. *Cmake* fragment for the project.

```
cmake –preset arm-none-eabi-gcc-debug -DBOARDS = nucleo_f207zg -DPROJECTS = STM32GPIO
cmake –build –preset arm-none-eabi-gcc-debug
```

Fig. 4. Commands for building the project.

```
cmake –preset arm-none-eabi-clang-debug -DBOARDS = nucleo_f207zg -DPROJECTS = STM32GPIO
cmake –build –preset arm-none-eabi-clang-debug
```

Fig. 5. Commands for building the project.

5. FPGA Case Study

The previous example demonstrated the ability to compile a bare metal program from a CubeIDE project. This previous case study is a great example for demonstrating the ease of using a fully-fledged IDE with a GUI, while maintaining the ability to cross-compile code on the command line. We will now extend this capability further to reconfigurable architectures that contain proprietary build tools and varying techniques for performing compilation.

Consider a Zynq-7000 architecture containing two ARM Cortex A9 processors and programmable logic in the form of an FPGA. This architecture allows compilation targeting the CPU (through bare metal or other operating systems such as Linux), and the FPGA logic itself. Following the previous example, we could target the processors with a bare metal implementation containing embedded C code. However, for this example we are going to focus on supporting FPGA synthesis using the Vivado IDE and CLI.

To begin, we first create a subdirectory inside the `projects/fpga` folder of the embedded-cross framework. This subdirectory will represent our project that will contain the HDL code. Within the embedded-cross-framework itself, you can find an example project called `EmptyProject`. This project contains a *vivado* project located in the `vivado_project` directory that may be initialised from a submodule (see Fig. 6).

```
git submodule update –init
    projects/fpga/EmptyProject/vivado_project
```

Fig. 6. Initialising the Vivado Submodule.

To support HDL compilation, we create a `project.cmake` file within our project directory.

You will notice in Fig. 7 that the language of this project is *HDL*. This language is custom to the embedded-cross-framework and may only be used within its directory structure (the `vivado_project` subdirectory by default, as discussed above).

Once this file is created, the configuration within the embedded-cross-framework is complete and the user may start performing FPGA synthesis via the command line. In this example, we will be compiling for a Zybo Z720 device that contains a Zynq-7000 SoC. To demonstrate a small working program, we have created a synchronous program in VHDL that toggles LEDs from a switch input. You may see the code for this example in Fig. 8.

This program simply checks the value of the `sw` input every rising clock edge for a logic high value. When the switch is logic high, the LEDs are also set to high, otherwise they will assume a logic low value. The constraints file mapping the inputs into this module with the pins on the FPGA is also provided in the Vivado project (see Fig. 9).

To compile this project, we follow the exact same procedure as the previous case study by using the `fpga` preset, `EmptyProject` project, and *`zybo_z720`* board (see Fig. 10).

```
cmake_minimum_required(VERSION 3.12)
project(EmptyProject LANGUAGES HDL)

#set(${CMAKE_PROJECT_NAME}_PROJECT_DIRECTORY_NAME vivado_project)
```

Fig. 7. The `project.cmake` file for the FPGA.

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity top is
port(
    sysclk: in std_logic;
    sw: in std_logic;
    led: out std_logic_vector(3 downto 0)
);
end top;
architecture Behavioral of top is
begin
process(sysclk)
begin
if rising_edge(sysclk) then
    if sw = '1' then
        led <= (others => '1');
    else
        led <= (others => '0');
    end if;
end if;
end process;
end Behavioral;
```

Fig. 8. The VHDL code that toggles the LEDs.

```
   ##Clock signal
   set_property -dict { PACKAGE_PIN K17     IOSTANDARD LVCMOS33 } [get_ports { sysclk }];
#IO_L12P_T1_MRCC_35 Sch=sysclk
   create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { sysclk }];
   ##Switches
   set_property -dict { PACKAGE_PIN G15      IOSTANDARD LVCMOS33 } [get_ports [62]];
#IO_L19N_T3_VREF_35 Sch=sw[0]set_property -dict { PACKAGE_PIN G15     IOSTANDARD LVCMOS33 }
[get_ports [62]]; #IO_L19N_T3_VREF_35 Sch=sw[0]
   ##LEDs
   set_property -dict { PACKAGE_PIN M14     IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
#IO_L23P_T3_35 Sch=led[0]
   set_property -dict { PACKAGE_PIN M15     IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
#IO_L23N_T3_35 Sch=led[1]
   set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; #IO_0_35
Sch=led[2]
   set_property -dict { PACKAGE_PIN D18     IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
#IO_L3N_T0_DQS_AD1N_35 Sch=led[3]
```

Fig. 9. The Constraints File.

```
cmake –preset fpga-xilinx-unknown-vivado -DPROJECTS = EmptyProject -DBOARDS = zybo_z720

cmake –build –preset fpga-xilinx-unknown-vivado
```

Fig. 10. The Command to Configure and Build the FPGA project.

The embedded-cross-framework will now compile the vivado project for the FPGA local to the *zybo_z720* board.

5.1. Introducing Verifiable Models

We now demonstrate how to convert the example program in Fig. 8 into a model-driven solution utilising LLFSMs [48]. The advantage of this approach is the ability to perform efficient runtime verification [49] that significantly mitigates the state explosion problem [63] while maintaining intrinsic synchronisation [47] and mutual exclusion between LLFSMs executing in parallel [64].

A quick introduction to LLFSMs is to consider them UML state machines without the presence of events, i.e. a system of states with transitions dictating the conditions under which the state changes. In LLFSMs, the transitions are labelled with *logical* expressions, rather than events or guarded events. Under LLFSM semantics [65, 66], the control of the FSMs execution is purely controlled by the FSM itself and not dictated by external factors such as interrupts/events originating from other parts of the system [7, 57]. In this manner, LLFSMs are predictable in their execution and suitable for state-based architectures [48] such as FPGAs.

To start creating LLFSMs in the embedded cross framework, a code generator [67] must be installed that converts LLFSMs designed in our editor [68] into VHDL source files that may be compiled into the FPGA. The instructions for installing this generator are provided in the code generators repository [67]. Once the code generator is installed, you may use an LLFSM editor to create an executable model containing VHDL code. In this example, our LLFSM looks like Fig. 11.

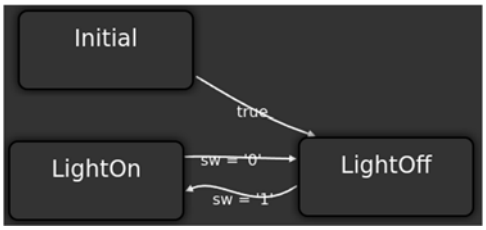

Fig. 11. The `LEDBlinker` LLFSM.

This machine begins in the `Initial` state where it will immediately transition into `LightOff` (specified by the `true` condition on its transition). When in this state, the LLFSM will poll the *sw* signal at a set period. When the `sw` signal is high, the machine will transition to `LightOn` where it will turn the *LED* on. The LLFSM will continue polling *sw* and transition back to `LightOff` when it's value is low, at which time the LLFSM will turn the LED off.

Similar to Fig. 8, the program is synchronous but takes additional clock cycles to complete a single states execution. A consistent view of the sensors is read at the start of a state's execution, called a *snapshot* that the state will act upon during its execution. At the end of the state's execution, the LLFSM will write any local changes to the corresponding actuators (the LED in this case). The exact time of reading and writing shared variables is determined *a priori*, and guaranteed throughout the entire LLFSMs deterministic execution. State executions are cyclic in that the LLFSM is continually performing a single iteration of a state. This mechanism provides the polling semantics as the LLFSM takes snapshots at the start of each iteration. The snapshot semantics are incredibly powerful when performing formal verification as they severely

mitigate the state explosion problem [2, 3, 49, 50, 63, 69].

Now that we have designed our LLFSM, we can save it to the filesystem in a model suitable for our editor and code generator. In our editor, two files are saved to the file system, the first is the LLFSM model of this machine named `LEDBlinker.machine`, and the second is a file called `Arrangement1.arrangement`. Arrangement1 is important to use as this file will allow us to generate and synthesise the VHDL code for our machine using the embedded cross framework. An arrangement is simply a group of LLFSMs that are executing in parallel, and acts as the top module to all LLFSMs on the FPGA. The machines within an arrangement may share variables with each other or interface with the environment through external sensors and actuators. The exact type of variable that an LLFSM will use is defined in the arrangement which will map the LLFSMs variables to the respective local variables or IO pins on the FPGA. In this example, the arrangement has defined the LED and the `sw` variable as signals external to the arrangement (i.e. within the environment). We now wish to incorporate these files into the embedded cross framework so that we may compile them in the same manner as all other projects. To begin, we modify our `project.cmake` slightly.

As can be observed in Fig. 12, the `project.cmake` is very similar to the HDL example. The only addition is the new language *LLFSMHDL* and a variable that specifies the location of the arrangement you wish to compile relative to the projects folder. You will also need to provide a vivado project like the standard HDL methodology previously discussed.

We can now begin compiling our LLFSMs by using the presets previously created.

The embedded cross framework will now generate the VHDL code for the LLFSMs before compiling them like the other HDL projects. This entire example is provided within the embedded cross framework [15] as an FPGA project titled LLFSMExample.

```
cmake_minimum_required(VERSION 3.12)
project(LLFSMExample LANGUAGES LLFSMHDL HDL)

set(${CMAKE_PROJECT_NAME}_ARRANGEMENT machines/Arrangement1.arrangement)
```

Fig. 12. The project.cmake for the LLFSM Example.

```
cmake --preset fpga-xilinx-unknown-vivado -DPROJECTS = LLFSMExample -DBOARDS = zybo_z720

cmake --build --preset fpga-xilinx-unknown-vivado
```

Fig. 13. The commands to configure and build the LLFSM example.

6. Hardware-agnostic Middleware Case Study

The embedded-cross framework also facilitates developing and deploying hardware-agnostic software modules to support multi-architecture development. This case study uses a proof-of-concept communication protocol implementation loosely based on TTCAN called G-TTCAN [70]. G-TTCAN is driven by callback functions implemented on each device, with functionality passed to G-TTCAN through some simple wrapper functions.

As a first step, we need to add the G-TTCAN implementation into the `middleware` folder of the embedded-cross framework. Rather than cloning the folder into the framework, it may be desirable to add it as a submodule as we should not be making changes to the repository inside the embedded cross framework.

```
git submodule add https://github.com/cpslabgu/gttcan.git

git submodule update --init --recursive
```

Using similar principles to the bare-metal case study outlined in Section 4, we can then create projects for different hardware platforms inside the `projects` folder. For each project, we should add the following to the `project.cmake` to include the G-TTCAN middleware.

This allows multiple projects to build using the same middleware, which is particularly useful when wanting to ensure that each project has the same version of the middleware. We can now implement the required callback functions and wrappers for each piece of hardware to implement G-TTCAN. In this case, the required function implementations are:

- transmit_frame – a function that will take a specified header and payload and transmit over CAN;
- set_timer_int – a function that will create a timer interrupt for a specified time from "now".

Additionally, the following functions must be called by the project:

- GTTCAN_transmit_next_frame – to be called when the timer interrupt triggers;
- GTTCAN_process_frame – to be called when the CAN receive interrupt triggers.

Some example code fragments for an STM32 and a TMS570 board are provided in Figs. 15 and 16.

```
## Middleware directory
set(${project_name}_MIDDLEWARE_DIR ${top_level}/middleware)

## GTTCan directory
set(${project_name}_GTTCAN_DIR
${${project_name}_MIDDLEWARE_DIR}/gttcan/Sources/gttcan)

## Add to Sources for the project.
${${project_name}_GTTCAN_DIR}/gttcan.c

## Add to Include Directories for the project.
${${project_name}_GTTCAN_DIR}/include
```

Fig. 14. *project.cmake* fragment to add G-TTCAN middleware.

```
void set_timer_int(uint32_t time) {
    rtiREG1->CMP[0U].COMPx = (time / CLOCK_DIVIDER) + timer_value;
}

void transmit_frame(uint32_t can_frame_id, uint64_t data) {
    can_frame_id = 0x60000000U | can_frame_id;
    canUpdateID(canREG4, canMESSAGE_BOX1, can_frame_id);
    uint8_t tx_data[8] =
    {
        data & 0xFF,
        (data >> 8) & 0xFF,
        (data >> 16) & 0xFF,
        (data >> 24) & 0xFF,
        (data >> 32) & 0xFF,
        (data >> 40) & 0xFF,
        (data >> 48) & 0xFF,
        (data >> 56) & 0xFF
    };
    canTransmit(canREG4, canMESSAGE_BOX1, tx_data);
}

void rtiNotification(rtiBASE_t *rtiREG, uint32 notification)
{
  GTTCAN_transmit_next_frame(&gttcan);
}

void canMessageNotification(canBASE_t *node, uint32 messageBox)
{
    current_time = get_current_timer();
    canGetData(canREG4, canMESSAGE_BOX2, (uint8 * )&rx_data[0]);
    uint32_t frameID = canGetID(canREG4, canMESSAGE_BOX2);
    GTTCAN_process_frame(&gttcan,current_time, frameID, data);
}
```

Fig. 15. Code fragment from TMS570 project.

```
void set_timer_int(uint32_t time, void* ctx) {
   __HAL_TIM_SET_AUTORELOAD(&htim2, time);
}

void transmit_frame(uint32_t can_frame_id, uint64_t data, void* ctx) {
  CAN_TxHeaderTypeDef tx_header;
  tx_header.IDE = CAN_ID_EXT; tx_header.ExtId = can_frame_id; tx_header.RTR = CAN_RTR_DATA;
  tx_header.DLC = 8;  tx_header.TransmitGlobalTime = DISABLE;
  uint32_t tx_mbox = 0;
  HAL_CAN_AddTxMessage(&hcan2, &tx_header, (uint8_t *)&data, &tx_mbox);
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  GTTCAN_transmit_next_frame(&gttcan);
}


void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
  current_time = get_current_timer();
  CAN_RxHeaderTypeDef rx_header;
  uint8_t rx_data[8] = {0, 0, 0, 0, 0, 0, 0, 0};
  HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &rx_header, (uint8_t *)rx_data);
  const uint64_t * const dataptr = (const uint64_t* const)rx_data;

  GTTCAN_process_frame(&gttcan, current_time, rx_header.ExtId, *dataptr);
}
```

Fig. 16. Code fragment from STM32 project.

Each HAL does things slightly differently (particularly with timers), but the required hardware-specific code is straightforward and architecturally very similar. Each project can now be built using the embedded-cross framework as per Section 4 and deployed to each device.

7. Discussion

The flexibility we are gaining by utilising the embedded-cross framework now allows us to refactor the software and add additional modules by simply modifying the `project.cmake` file accordingly. For example, the code managed by the CubeIDE mixes generated code with code written by the user. This not only violates important Software Engineering principles such as separation of concerns, but also creates fragility, requiring extra vigilance when making modifications or re-generating code after hardware re-configuration.

We have successfully used the embedded-cross-framework for the design and implementation of high-level executable models [71] that utilise logic-labelled finite-state machines (LLFSMs) [48] that allow us to create verifiable and composable [72], executable models that are guaranteed to exhibit the same behaviour semantics at runtime as during system validation, regardless of whether we run our code on an embedded microcontroller or an FPGA. Moreover, our framework allows us to perform continuous integration, for example through GitHub workflows, enabling agile software development for embedded and robotic systems. Through hardware-in-the-loop integration, we can download artefacts to the embedded system as part of our continuous integration workflow and run automated integration tests in a similar fashion to unit tests.

By factoring out low-level code, we can factor out hardware specifics and develop or utilise existing cross-platform hardware abstractions. This allows our code base to be reusable and re-targetable towards different hardware architectures or operating systems with minimal effort. This is particularly important in an age of chip shortages and supply chain instabilities. For complex, high-level code, this also significantly reduces the development overhead for different components in larger and more complex systems.

8. Conclusions

Our framework focuses on compatibility among the different stages of the deployment cycle. The framework can produce build artefacts for all viable hardware architectures, while integrating new software entities, allowing validation of reusable code for all supported architectures. The deployment to test hardware or engineering models is not separate from the deployment to models already deployed and functioning. This fact ensures that software differences between robotic hardware in the field and testing apparatus are drastically minimised if not removed entirely. This process also allows monitoring feedback from target hardware during test scenarios. Quantities such as power consumption, temperature, mechanical stress, and software status can be flagged and reported throughout the scenario, supporting requirements validation in real time. Such validation can also detect invalid software before deployment, creating a much more reliable codebase of validated software.

References

[1]. B. Selic, The pragmatics of model-driven development, *IEEE Software,* Vol. 20, Issue 5, 2003, pp. 19-25.

[2]. C. McColl, M. McColl, V. Estivill-Castro, R. Hexel, Decomposable and executable models for verification of real-time systems, in *Proceedings of the International Conference Model-Driven Engineering and Software Development (MODELSWARD'22),* 2022, pp. 135-156.

[3]. C. McColl, V. Estivill-Castro, M. McColl, R. Hexel, Verifiable executable models for decomposable real-time systems, in *Proceedings of the International Conference Model-Driven Engineering and Software Development (MODELSWARD'22)*, 2022, pp. 182-193

[4]. V. Estivill-Castro, R. Hexel, Simple, not simplistic the middleware of behaviour models, in *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'15)*, April 2015, pp. 189-196.

[5]. S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, Robot Operating System 2: Design, architecture, and uses in the wild, *Science Robotics,* Vol. 7, Issue 66, May 2022.

[6]. M. Fowler, J. Highsmith, The agile manifesto, *Software Development,* Vol. 9, Issue 8, 2001, pp. 28-35.

[7]. C. Zielinski, M. Figat, R. Hexel, communication within multi-FSM based robotic systems, *Journal of Intelligent & Robotic Systems,* Vol. 93, Issue 3, 2019, pp. 787-805.

[8]. M. Figat, C. Zieliński, R. Hexel, FSM based specification of robot control system activities, in *Proceedings of the 11th International Workshop on Robot Motion and Control (RoM oCo'17)*, 2017, pp. 193-198.

[9]. V. Estivill-Castro, R. Hexel, Module isolation for efficient model checking and its application to FMEA in model-driven engineering, in *Proceedings of the 8th Intl. Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'13)*, 2013, pp. 218-225.

[10]. H. Kopetz, R. Nossal, Temporal firewalls in large distributed real-time systems, in *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997, pp. 310-315.

[11]. D. Aceituna, Survey of concerns in embedded systems requirements engineering, *SAE International Journal of Passenger Cars-Electronic and Electrical Systems,* Vol. 7, Issue 1, 2013, pp. 1-13.

[12]. S. Yoo, A. A. Jerraya, Introduction to hardware abstraction layers for SoC, in Embedded Software for SoC (A. A. Jerraya, S. Yoo, D. Verkest, N. When, Eds.), *Springer US*, 2003, pp. 179-186.

[13]. R. Barry, FreeRTOS – Real-time operating system for microcontrollers, Real Time Engineers Ltd., https://www.freertos.org

[14]. CMSIS – Common Microcontroller Software Interface Standard, Arm, https://arm-software.github.io/CMSIS_6/

[15]. R. Hexel, C. McColl, M. McColl, G. Tuxworth, Embedded Cross Framework, https://github.com/mipalgu/embedded-cross-framework
[16]. M. Banzi, D. Cuartielles, T. Igoe, M. G., D. Mellis, Arduino IDE, https://www.arduino.cc/en/software
[17]. R. Stallman, GCC, the GNU Compiler Collection, Free Software Foundation Inc., https://gcc.gnu.org
[18]. I. Kravets, PlatformIO – Your Gateway to Embedded Software Development Excellence, PlatformIO Labs, https://platformio.org
[19]. PlatformIO Integrations, https://platformio.org/install/integration
[20]. Visual Studio Code. Microsoft, https://code.visualstudio.com
[21]. J. Wiegand, Eclipse: A platform for integrating development tools, *IBM Systems Journal,* Vol. 43, Issue 2, 2004, pp. 371-383.
[22]. Qt Creator, https://doc.qt.io/qtcreator/
[23]. Vim – the ubiquitous text editor, http://www.vim.org
[24]. M. Banzi, D. Cuartielles, T. Igoe, M. G., D. Mellis, Arduino Language Reference. Arduino, https://www.arduino.cc/reference/en/
[25]. MISRA C:2023 – Guidelines for the Use of the C Language in Critical Systems, Second Revision of the Third Ed., *The MISRA Consortium Limited*, 2023.
[26]. Robotic Operating System, Open Robotics, https://www.ros.org
[27]. T. Azumi, Y. Maruyama, S. Kato, ROS-lite: ROS framework for NoC-based embedded many-core platform, in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'20)*, 2020, pp. 4375-4382.
[28]. N. Hauser. MODM: a barebone embedded library generator, https://modm.io
[29]. K. H. Martin, Mastering CMake, 7th Ed., *BPB Publications*, 2019.
[30]. A. Canis, *et al.*, LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems, *ACM Transactions on Embedded Computing Systems (TECS),* Vol. 13, Issue 2, 2013, pp. 1-27.
[31]. C. Baaij, CλasH: from Haskell to hardware, MSC Thesis, *University of Twente*, 2009.
[32]. A. Bukowiec, J. Tkacz, M. Adamski, Transition based synthesis with modular encoding of Petri nets into FPGAs, *Advances in Electrical and Electronic Engineering,* Vol. 12, Issue 5, 2014, pp. 435-442.
[33]. S. Ostroumov, L. Tsiopoulos, VHDL code generation from formal event-B models, , in *Proceedings of the 14th Euromicro Conference on Digital System Design*, Aug. 2011, pp. 127-134.
[34]. L. Gomes, A. Costa, J. P. Barros, P. Lima, From Petri net models to VHDL implementation of digital controllers, in *Proceedings of the 33rd Annual Conference of the IEEE Industrial Electronics Society (IECON'07)*, 2007, pp. 94-99.
[35]. G. Łabiak, G. Borowik, Statechart-based controllers synthesis in FPGA structures with embedded array blocks, *International Journal of Electronics and Telecommunications,* Vol. 56, Issue 1, 2010, pp. 13-24.
[36]. W. E. McUmber, B. H. C. Cheng, UML-based analysis of embedded systems using a mapping to VHDL, in *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, 1999, pp. 56-63.
[37]. E. Soto, M. Pereira, Implementing a Petri net specification in a FPGA using VHDL, in Design of Embedded Control Systems, *Springer US*, 2005, pp. 167-174.
[38]. S. K. Wood, D. H. Akehurst, O. Uzenkov, W. G. J. Howells, K. D. McDonald-Maier, A model-driven development approach to mapping UML state diagrams to synthesizable VHDL, *IEEE Transactions on Computers,* Vol. 57, Issue 10, 2008, pp. 1357-1371.
[39]. H. Kopetz, Sparse time versus dense time in distributed real-time systems, in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992.
[40]. R. Alur, D. Dill, The theory of timed automata, in *Proceedings of the Real-Time: Theory in Practice Conference*, 1992.
[41]. S. P. Miller, M. W. Whalen, D. D. Cofer, Software model checking takes off, *Communications of the ACM,* Vol. 53, Issue 2, 2010, pp. 58-64.
[42]. D. Harel, Statecharts: a visual formalism for complex systems, *Science of Computer Programming,* Vol. 8, Issue 3, 1987, pp. 231-274.
[43]. V. Hadžić, R. Primas, R. Bloem, Proving SIFA protection of masked redundant circuits, *Innovations in Systems and Software Engineering,* Vol. 18, Issue 3, 2022, pp. 471-481.
[44]. D. Allen, Automatic one-hot re-encoding for FPGAs, in *Proceedings of the Field-Programmable Gate Arrays: Architecture and Tools for Rapid Prototyping Conference*, 1993.
[45]. A. Aurandt, P. Jones, K. Rozier, Runtime verification triggers real-time, autonomous fault recovery on the CySat-I, in *Proceedings of the NASA Formal Methods Conference (NFM'22)*, 2022.
[46]. M. Leucker, C. Schallhart, A brief account of runtime verification, *The Journal of Logic and Algebraic Programming,* Vol. 78, Issue 5, 2009, pp. 293-303.
[47]. V. Estivill-Castro, R. Hexel, M. McColl, High-level executable models of reactive real-time systems with logic-labelled finite-state machines and FPGAs, in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'18)*, 2018, pp. 1-8.
[48]. V. Estivill-Castro, R. Hexel, Logic labelled finite-state machines and control/status pull technology for model-driven engineering of robotic behaviours, in *Proceedings of the 26th International Conference on Software & Systems Engineering and Their Applications*, 2015.
[49]. M. McColl, C. McColl, R. Hexel, Automatic verification of high-level executable models running on FPGAs, in *Proceedings of the Automated Technology for Verification and Analysis Conference (ATVA'23)*, 2023.
[50]. M. McColl, C. McColl, A. Pereira, P. de Souza, G. Tuxworth, R. Hexel, Continuous formal verification for aerospace applications, in *Proceedings of the IEEE Aerospace Conference (AeroConf'24)*, 2024.
[51]. R. Hexel, C. McColl, M. McColl, G. Tuxworth, Highly flexible and scalable software architectures for robotic applications, in *Proceedings of the 4th IFSA Winter Conference on Automation, Robotics & Communications for Industry 4.0 / 5.0 (ARCI'24)*, Innsbruck, Austria, February 2024, pp. 151-156.
[52]. CMake Presets, https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html
[53]. C. Lattner, Clang: a C language family frontend for LLVM, https://clang.llvm.org
[54]. Swift Compiler, https://www.swift.org/swift-compiler/
[55]. L. Torvalds, J. Hamando, e. al., Git – fast version control, https://git-scm.com

[56]. S. Chacon, B. Straub, Git tools – submodules, in Pro Git, 2nd Ed., *Springer Nature*, 2014, pp. 298-318.
[57]. V. Estivill-Castro, R. Hexel, C. Lusty, High performance relaying of C++11 objects across processes and logic-labeled finite-state machines, in *Proceedings of the 4th International Conference Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR'14)*, 2014.
[58]. J. Paternoster, CANopenNode – CANopen protocol stack, https://canopennode.github.io/CANopenNode/index.html
[59]. S. Abdulrasool. µSwift[Core], https://github.com/compnerd/uswift
[60]. GitHub, https://github.com
[61]. GitLab, https://gitlab.com
[62]. E. Sweet, CMake Presets integration in Visual Studio and Visual Studio Code, https://devblogs.microsoft.com/cppblog/cmake-presets-integration-in-visual-studio-and-visual-studio-code/
[63]. V. Estivill-Castro, R. Hexel, Arrangements of finite-state machines-semantics, simulation, and model checking, in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'13)*, 2013.
[64]. M. McColl, Modelling complex real-time systems on field-programmable gate arrays using logic-labelled finite-state machines, Honours Thesis, *Griffith University*, 2018.
[65]. V. Estivill-Castro, R. Hexel, Correctness by construction with logic-labeled finite-state machines – comparison with event-B, in *Proceedings of the 23rd Australasian Software Engineering Conference (ASWEC'14)*, 2014, pp. 38-47.
[66]. C. McColl, V. Estivill-Castro, R. Hexel, Versatile but precise semantics for logic-labelled finite state machines, *International Journal on Advances in Software*, Vol. 11, Issue 3, 2018, pp. 227-238.
[67]. M. McColl, LLFSMGenerate, https://github.com/CPSLabGU/LLFSMGenerate.
[68]. M. McColl, C. McColl, Editor, https://github.com/CPSLabGU/editor
[69]. C. McColl, Leveraging Decomposition for the modelling, implementation and verification of complex dependable real-time systems, PhD Thesis, *Griffith University*, Australia, 2023.
[70]. G-TTCan, https://cpslabgu.github.io/gttcan/
[71]. D. Billington, V. Estivill-Castro, R. Hexel, A. Rock, Modelling behaviour requirements for automatic interpretation, simulation and deployment, in Simulation, Modeling, and Programming for Autonomous Robots, *Springer*, Berlin, Heidelberg, pp. 204-216.
[72]. V. Estivill-Castro, R. Hexel, Verifiable parameterised behaviour models, in *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, Madeira, Portugal, January 2018, pp. 364-371.