Enabling Modern Application Development with Swift on the Nao/Pepper Robots

Callum McColl[1(✉)], Vladimir Estivill-Castro[2], Eugene Gilmore[1], Morgan McColl[1], and René Hexel[1]

[1] MiPal, Griffith University, Brisbane, QLD, Australia
callum.mccoll@griffithuni.edu.au
[2] MiPal, Universitat Pompeu Fabra, Barcelona, Spain

Abstract. We show the advantages of using `Swift` as the programming language for behaviours on the Pepper and Nao robots as used with the RoboCup Standard Platform League and the RoboCup@Home - Social Standard Platform. We show that `Swift` is not only incorporating modern features of object-oriented programming and functional programming, but is also now a stable systems programming language that enables both high-level development as well as fine hardware control. Deterministic memory management makes `Swift` suitable for real-time, embedded systems, and thus for robotic applications. Moreover, we show in this paper we can apply model-driven software-development by deploying behaviours coded as executable arrangements of logic-labelled finite-state machines (LLFSMs). We also show LLFSMs are not only suitable for reactive architectures, but also for deliberative architectures.

Keywords: Functional programming languages · Logical programming languages · Model-driven software development · Deliberative architectures

1 Introduction

While there have been long-ranging debates on the most relevant programming language for Software Engineering as well as robotics [29,31,38], there is agreement on important characteristics that a programming language for robotics and embedded systems needs to satisfy. Number one is to be Turing complete[1]. Importantly, it should be a systems programming language, able to provide fine grained, standardised control over devices, hardware drivers, and communication mechanisms, with predictable performance to enable integration across hardware, from sensors to actuators. The capability to have control over the temporal domain in execution, task control, and parallelism is crucially important.

[1] This formally means that it should be as expressive as a Turing machine, including sequencing, conditionals, and iteration found in most imperative languages [17].

R. Alami et al. (Eds.): RoboCup 2021, LNAI 13132, pp. 15–27, 2022.
https://doi.org/10.1007/978-3-030-98682-7_2

We argue here that such a language should also enable a seamless transition to higher behaviour-based descriptions and model-driven development.

We port `Swift` and its supporting environment to the virtual machine development environment associated with the SoftBank Pepper and Nao robots, as well as to the robots themselves. `Swift` has long produced fast and predictable executables for iOS and macOS, but is Open Source and features modern and safe language concepts. Complex data testbeds show `Swift` code to be about two times faster than Objective-C [39] and four to eight times faster than Python [1], while being similarly discoverable to programmers (e.g. through *Playgrounds* that facilitate interactivity during development and enable programmers to quickly test new algorithms). `Swift`'s modern features include closures, generics, and type inference, which have been shown to result in higher productivity and more adherence to common software patterns.

The clean syntax of `Swift` is a major factor for its higher readability [40] and thus maintainability, while its semantics make it harder to make mistakes common to systems programming languages and add a layer of quality control during development. Functional language features that provide referential transparency and avoid implicit state or mutable data allow writing performant code in more functional style [8]. This also facilitates pure functions and idempotence (i.e. lack of side-effects and the same output regardless how many times a function is called), enabling parallelisation as function calls become independent.

With `Swift` being Open Source, while being a relatively young language, and still somewhat of a moving target [33], it has become popular very quickly, including on the server side and cloud computing. The `Swift` language won first place for *Most Loved Programming Language* in the *Stack Overflow Developer Survey* 2015 and second place in 2016. Three years ago, `Swift` became the 12th most popular language, overtaking Objective-C, Go, Scala, and R. Now, in 2021, the reviews of the full-stack academy place `Swift` second just below JavaScript. For robotics, `Swift` provides the capability dynamic libraries and, vitally, a low memory footprint through value types and Automatic Reference Counting (ARC) that offers similar convenience as tracing garbage collectors, but vastly superior and deterministic performance. This is in contrast to languages such as Python, Java, C♯, or Go, whose lack of ability to bound memory usage and garbage collection performance implies unpredictability of CPU usage, suggesting that every single thread on board of the robot could be denied vital CPU time, potentially resulting in catastrophic consequences. For example, what would be the use of a Kalman filter if the time-stamp of the sensor reading was unpredictable or seriously jeopardised? The literature has plenty of discussion [3] why, e.g., the versions of Java for real-time and/or embedded systems are not completely satisfactory.

While IDE integration is not as mature as with other languages and despite some blogs criticising `Swift`, suggesting that Objective-C can use C and C++ libraries more smoothly, we demonstrate that we can use in-memory middleware for module/package/node communication quite effectively, integrating applicable C and C++ libraries for robotics applications.

2 Cross Compiling

The goal of cross-compilation is to be able to build a program for a specific platform on a different platform. When compiling for the Nao and Pepper robots, SoftBank Robotics provides GNU tools commonly found in linux systems for C and C++ which have been specifically built to allow cross-compilation. The tools include `binutils` [15] which contains tools such as the linker (`ld`), `glibc` [16] which contains the C standard library, and the `gcc` project [14] which contains the C compiler (`gcc`) as well as other compilers such as the C++ compiler `g++`. This, for example, allows us to build Nao-robot applications from a 64-bit Linux or macOS. We note that the Nao and the Pepper use some version of the linux kernel (for us, we use version 2.6). The kernel is simply a C program, thus a C compiler is needed to compile the kernel.

When cross-compiling, two things are required on the host (the system executing the cross-compiler): the tools (the cross-compiler and linker), and, the `sysroot` folder containing all necessary files needed for compilation for the target platform, such as the Nao. The `sysroot` folder is usually setup to mimic the folder structure of the target and generally contains headers and libraries which are needed at runtime by the programs being compiled. When using the GNU tools (as Aldebaran did originally), a compiler must be provided for each host and target combinations. So in order to build for the Pepper and Nao robots on two separate hosts—for example macOS and linux—then you need a total of four compilers. Two compilers for each host which build for the Nao and Pepper respectively. However, we can use alternatives to this approach.

Our `Swift` project is built using LLVM [20], built on top of the GNU stack and provides compilers (`clang` for the C language and `clang++` for C++) that allow programs to be cross-compiled without having to build a separate cross-compiler. In other words, the same compiler that is built for the host is able to cross-compile for other targets (as already utilised, e.g., for iOS and tvOS applications). We potentiate this further here for `Swift`, which already leverages LLVM, by creating an environment that allows us to cross-compile for Nao robots using only one cross-compiler.

3 `Swift` for the Pepper

To enable `Swift` for a new host system and target system, such as the Pepper, we need to build the `Swift` compiler (and libraries), often passing platform-specific flags to the C/C++ compilers to fix compiler errors or warnings. To ensure consistency, alleviate setup-specific issues, and document the process, we provide a Docker container which builds the `Swift` project [27]. Docker allows OS-level virtualisation in what is called containers, similar to virtualisation; however, rather than a host OS emulating the hardware in favour of a guest OS [37], containers share the host kernel. Containers are now a de-facto standard to share the environment for a package of applications.

Another challenge when building for the Pepper is the fact the `Swift` project does not support compiling for 32-bit linux targets. That is, none of the build scripts that are provided by the `Swift` project will allow such cross-compilation. Instead, we release our own scripts that our Docker container [27] automatically downloads and uses.

To cross-compile `Swift` for the Pepper, we need to replace existing `binutils` with `gold` enabled `binutils` (`--enable-gold`), build the LLVM tools for the host and the target, and then build the `Swift` standard library for the target. This allows us to create a `swiftenv` toolchain that enables building vital infrastructure libraries for the target such as `libdispatch` and Foundation.

3.1 Replacing `Binutils` and Building the LLVM Toolchain

SoftBank's toolchain provides an older `ld` linker that does not have the capabilities required by more modern languages such as `Swift`. The `Swift` project requires the gold linker [41], `ld.gold`, a more modern alternative to `ld`. Our scripts download `binutils` and recompile them, enabling the gold linker and allowing it to be executed on the 64-bit host, but cross-compile for the 32-bit Pepper.

The `Swift` project uses a series of `git` repositories, utilising a consistent version tag, to distribute a specific version of the language and the tools that are used within the `Swift` toolchain. The version of `Swift` that we have built is version 5.1.4 (tagged as swift-5.1.4-RELEASE).

We then build the LLVM project, creating C/C++ compilers that are capable of running on the host, but also cross-compile for the Pepper. We recommend building the C/C++ compilers using our `Swift` project since subsequent cross-compilation of robotic applications with `Swift` is simpler and more convenient. In particular, fewer flags are necessary for cross-compilation than with the GNU cross-compilers provided by the original SoftBank toolchain.

Using the host tools built in the previous step, a second build cross-compiles the LLVM toolchain for the Pepper delivering pre-requisites for the `Swift` cross-compiler and standard library. This requires several flags that inform the host C/C++ compiler where the shared objects, headers and C standard library exist within the Pepper toolchain. This is necessary to redirect the compiler away from the usual locations in the 64-bit host environment.

3.2 Building the `Swift` Environment for the Pepper

As we cross compile, we can skip building a `Swift` compiler to run on the Pepper, but, the `Swift` standard library must still be built, which is now possible with our cross-compilers. However, The `Swift` standard library consists of several modules written in different languages, some are C/C++ while others (such as the main standard library) are written in `Swift` itself and needs to be compiled and installed into the `sysroot`. Importantly here, since the standard library is written in `Swift`, a `Swift` compiler with a specific version is needed within the Docker container. We use `swiftenv` [13] in order to download the appropriate compiler for the version of `Swift` that we are compiling, to ensure that the host

Swift compiler and all the other tools that exist within the toolchain match the versions of the tools that we are building for the Pepper.

The Swift project follows a folder structure where the actual Swift standard library folders are placed within a Swift folder under the installation directories lib folder (typically /usr/lib for a normal installation). The installation folder that our scripts use is $SYSROOT/home/nao/swift-tc where $SYSROOT represents the absolute path of the sysroot directory on the build system. The reason why we use this installation prefix is to follow Pepper's security policy where the only folder with write access is the /home/nao folder. This swift directory follows the following format:

```
$SYSROOT/home/nao/swift-tc/lib/swift
└── <os>
    ├── <arch>
    │   ├── Swift.swiftdoc
    │   ├── Swift.swiftinterface
    │   ├── Swift.swiftmodule
    │   ├── glibc.modulemap
    │   └── ...
    ├── libswiftCore.so
    ├── libswiftGlibc.so
    └── ...
```

In this folder structure <os> and <arch> represent the target operating system and architecture (e.g. linux and i686 for the Pepper). These folders contain the Swift standard library files, consisting of the shared objects or dynamic libraries required by the compiler. The glibc.modulemap file includes the C standard library which allows developers to import into their programs by a simple import statement, for example (import Glibc).

This folder structure is important because when building the Pepper Swift toolchain, some of these files need to change. Recall that we leverage swiftenv to install a version of Swift that matches the version of Swift that we are building for the host. This makes it so that we can use the host Swift compiler to cross-compile Swift programs. However, the host Swift toolchain contains the standard library for the host 64-bit linux. We copy this toolchain into a new Pepper toolchain folder, and replace the Swift folder contents with the cross-compiled toolchain that was installed into the sysroot, replacing the 64-bit standard library with the 32-bit standard library built for the Pepper. Then we patch the glibc.modulemap file to correct the hardcoded paths to point to the Pepper C standard library within sysroot.

3.3 Building Libdispatch and Foundation for the Pepper

Once the Pepper swiftenv toolchain has been created, we can use it to build the remaining projects that make up the required Swift infrastructure. The libdispatch project is a C project, but the Swift variant contains wrappers that enable simple access to libdispatch from Swift programs. The Foundation framework was once written in Objective-C, but has since been rewritten in pure Swift. This allows it to be deployed to linux platforms, adding extra functionality on top of the Swift standard library. By using the swiftenv, we are able to compile these components and install them within the sysroot.

4 Swift on Legacy Robots

Older versions of the Nao (version 5 or earlier) use older versions of the C compiler and standard library that fail to meet the minimum system requirements for cross-compiling the swift toolchain. To overcome this issue, we need to take the extra step of installing an entirely separate *root* directory within the home directory of the Nao that will contain all files required for a newer linux system containing a suitable version of `glibc`. However, we avoid installing an entire linux system. Instead, we install the minimum amount of software that `Swift` needs in order to minimise disk space.

Installing a newer version of `glibc` is particularly challenging, it is an integral part of the system, i.e., there is a mutual inter-dependence between the kernel and the GNU C compiler (`gcc`) on the one hand and `glibc` on the other hand. This circular dependency creates close coupling, and if the newer version of `glibc` breaks ABI (application binary interface) stability (meaning that symbols within the library have changed), programs that have already been compiled will stop working. Since the linux kernel is itself a C program which links against `glibc`, this means that nothing will work. Upgrading an existing `glibc` to a newer version is therefore only possible if the new `glibc` version does not break ABI stability. In our illustration (Nao V5), the minimum version of `glibc` supported by the swift toolchain does break ABI stability.

An additional challenge is that the `glibc` version that is needed for the swift toolchain does not compile under the Nao kernel. For our approach of a parallel system root, the new toolchain must not depend on anything outside of this parallel system root. If we were to simply compile `glibc` using the existing `gcc` compiler, then the new `glibc` would link against `libgcc.so` which is outside of the system root directory. Hence, we needed to compile an entirely new linux systems from source, following [2]. This book provides a comprehensive guide for creating new Linux distributions without any prior knowledge, particularly, the details for compiling a self-contained system root [2]. Akin to this approach, we created a parallel system root containing the new `glibc`, kernel and `gcc`, taking the opportunity to upgrade the `gcc` compiler to a more modern version. From this point, the same steps as in Sect. 3 apply.

5 Efficient Robot Software Architectures Through Swift

5.1 Behaviour-Based Architectures

Finite-State Machines are ubiquitous models of behaviour. In robotics, they have been used in several forms, perhaps most notably with the introduction of the subsumption architecture [4] and therefore, in behaviour-based control by the seminal description of Toto [22–24]. Finite-state machines are typically associated with reactive software architectures for robotics, because of the prevalence of the state-chart format introduced by Harel [18]. However, Harel's semantics of finite-state machines is event-driven, implying that the driver of any activity is the environment who generates events to state-charts waiting and expecting

events [7]. Harel's vision was the creation of mechanisms to compose more sophisticated behaviours by hierarchies of state-charts as nesting one sub-machine in a parent sub-machine created an appealing mechanism nesting behaviours from other behaviours. This idea gained acceptance in OMT [35] and latter became the dominant semantics of UML's model for describing behaviour [32]: that is the semantics of *run until completion* [7,36].

However, the original time-augmented finite-state machines of the subsumption architecture were essentially logic-labelled; that is, what labels a transition between two states is a Boolean expression. In such models, the machine runs at its own pace, evaluating the expressions of the transitions and performing a transition only when the expression evaluates to `true`. Expressions labelling transitions can be as sophisticate as decision trees [21,34]. Logic-Labelled Finite State Machines (LLFSMs) use this alternative paradigm of execution enabling much safer elaboration of behaviours [30]. Rather than using nesting, LLFSMs can use stacking like the subsumption architecture [4] or create versatile dynamic executions by `suspend`, `resume` and `restart` calls. Concurrency of arrangements of LLFSMs is achieved by a sequential pre-defined schedule of execution, enabling smaller state-spaces for formal verification and model checking [12].

5.2 Formal Verification

Implementations of LLFSMs were based on interpreters until `clfsm` [12] offered their compilation into loadable libraries in `C++`. However, doing so left them unverifiable as the C++ language does not posses any means of being able to query the state of the variables making up the LLFSM at run time. Our porting of `Swift` to the SoftBank robots enables efficient execution on board of the robot of compiled `swift` LLFSMs while still enabling formal verification. Here we argue for the use of `swiftfsm` [25,26]: a scheduler for LLFSMs enabling formal verification.

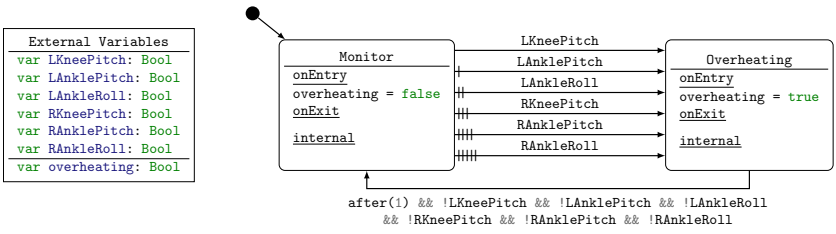


Fig. 1. The temperature monitor LLFSM

We now present a small illustration of the efficacy and usefulness of our `Swift` toolchain. We use `swiftfsm` to perform verification of a small piece of infrastructure: a module which monitors the temperature levels of the motors on the robot. The corresponding LLFSM is presented in Fig. 1. For the sake of

this example, we represent the temperature sensors as Boolean values. This is to minimise the resulting Kripke Structure from the verification for this simple example. Each Boolean variable in the external variables represents whether or not that particular sensors is overheating. If the sensor is overheating, then the value is `true`, otherwise the value is `false`. The `overheating` external variable represents a control message which notifies any other modules that a joint is overheating. Thus, this LLFSM simplifies the process of checking whether the robot is overheating by distilling the sensors into one overheating message.

The machine starts in the Monitor state indicated by the initial pseudo-state icon. The Monitor sets `overheating` to `false` and contains a number of transitions to the Overheating state. We have labelled each of these transitions with a check for each external variable. Therefore, if any of these external variables evaluate to `true`, then the machine will transition to the Overheating state. Our machine takes advantage of the LLFSM semantics [10] implemented by `swiftfsm` to continuously poll the external variables throughout its execution. This simple fact also demonstrates the advantages of `swiftfsm` in handling issues of concurrency without any overhead created by the user. Lastly, if the machine transitions to the Overheating state, then the overheating external variable is set to `true`. After a second and when all sensors are not overheating, the machine transitions back to the Monitor state thus setting `overheating` back to `false`.

One of the advantages of using `Swift` and `swiftfsm` is that `swiftfsm` utilises an extensive set of protocols to enforce its semantics. A protocol in `Swift` is a construct similar to an interface in Java, but more powerful. This *protocol oriented design* (a paradigm widely used in the `Swift` community) establishes the desired semantics of the scheduler which enables formal verification. Only by enforcing LLFSMs to follow these semantics—through the use of protocols—is a verification possible [28]. Performing the actual verification is only possible because of the reflection capabilities of the language. As stated earlier, the C++ variants of LLFSMs were only able to verify LLFSMs through the implementation of an interpreter. With `Swift`, we are able to verify LLFSMs natively without having to implement a custom interpreter. Not only does the temperature monitor LLFSM demonstrate these features of `Swift`, it also demonstrates one of the necessary features for robotics: interoperability with C.

In the temperature monitor, each external variable maps to a message within a middleware implemented in C: the `gusimplewhiteboard` [12]. Through the use of this middleware, any `Swift` LLFSM (and by extension any `Swift` program) is able to communicate with any other modules within the system. Many languages support C bindings making the use of a middleware a key step in being able to communicate between modules written in different languages including C++. The temperature monitor demonstrates that the use of `Swift` does not demand that existing modules must be rewritten. `Swift` modules can add extra functionality and coexist with existing modules written in different languages. Other LLFSMs written in C++ or `Swift` can now use the output of the `Swift` machine in Fig. 1 through the use of the same middleware. Moreover, using LLFSMs enables Model-Driven Software Development. That is, the LLFSMs

can be defined using a language agnostic meta-model and editor. Then, model-to-text transformation produces the equivalent behaviour for C++, `Swift`, and LISP [5]. More importantly for formal verification, a model-to-text transformation produces the equivalent NuSMV model (see coming up example in Fig. 3b) which can be co-simulated by the model-checker (that is, by NuSMV) as well as verifying properties against it.

5.3 Deliberative Architectures

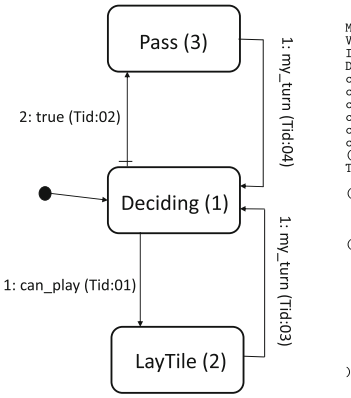
Importantly, LLFSMs enable the ability to create declarative and deliberative architectures, as the expression labelling the transitions can be a query to a reasoning agent [11] or a task planning system [9].

We now illustrate LLFSMs capability to describe behaviours beyond a reactive architecture. Our example is part of an application of a social robot that plays the game of Spanish dominoes with a human partner against another pair of players [19]. Figure 2 shows a small `Prolog` program that defines whether a player can play one of the tiles in their hand or must pass. Figure 3a shows an executable (compiled, not interpreted) of deliberative LLFSMs of a player that plays with the most naive strategy, play the first tile playable in the hand or pass. A more sophisticated player can be produced by sophistication of the `Prolog` program of Fig. 2. Nevertheless, the current version is sufficient to implement the server of the game that monitors players following the rules to play valid tiles and not revoke (claim to pass when it is possible to play).

```
% Return what tile a player can play
% Can play on end with value X the tile [X,Y]
tile_playable_on_end(X,[X,Y]).
tile_playable_on_end(Y,[X,Y]).
% Can play on end with value X with a tile on
%the list that starts with a tile the given tile
can_play_on_end(X,[T|R],T) :- tile_playable_on_end(X,T).
can_play_on_end(X,[T|R],O) :- can_play_on_end(X,R,O).
%Can the player with hold H play on a chain with ends X and Y the tile T
can_play_low_end(X,Y,H,T) :- can_play_on_end(X,H,T).
can_play_high_end(Y,X,H,T) :- can_play_on_end(X,H,T).
can_play(X,Y,H T) :- can_play_low_end(X,Y,H,T).
can_play(X,Y,H,T) :- can_play_high_end(X,Y,H,T).
```

Fig. 2. Determine a tile to play in the players hand.

The experience of enabling `Swift` is crucial to port GNU-Prolog [6] to the robots. Porting the interpreter `gprolog` and the compiler `gplc` for interfacing with C amounts to downloading and patching the sources of a suitable version and using out C/C++ cross-compilers. However, calling prolog from `Swift` LLFSMs requires cross-compiling a C/C++ wrapper. In our illustration we wrap the querying whether a player holding a specific hand can lay a tile on the board and must pass. This wrapper is now a C-function that is in itself wrapped and linked by `Swift` and thus, we can invoked in the label of a transition of the LLFSM defining the player's behaviour. However, this is easier said than done. While



(a) Executable behaviour.

```
MODULE DominoPlayer(executing my_trun)
VAR pcDominoPlayer : 0..3; can_play : boolean;
INIT (pcDominoPlayer = 0)
DEFINE
condT00:= (((pcDominoPlayer=0)) & TRUE);
condT01:= (((pcDominoPlayer=1)) & (can_play));
condT02:= (((pcDominoPlayer=1))& TRUE);
condT03:= (((pcDominoPlayer=2)) & (my_trun));
condT04:= (((pcDominoPlayer=3)) & (my_trun));
condDefault0:= ((!condT00) & (!condT01) & (!condT02) & (!condT03) &
(!condT04));
TRANS
  (!executing -> ((next(pcDominoPlayer)=pcDominoPlayer) &
(next(can_play)=can_play)))
   & (executing ->
    ( ( condT00 & (next(pcDominoPlayer)=1) & (next(can_play)=can_play))
    | ((!condT00) & condT01 & (next(pcDominoPlayer) = 2) &
(next(can_play)=can_play))
    | ((!condT00) & (!condT01) & condT02 &
      (next(pcDominoPlayer)=3) & (next(can_play)=can_play))
    | ((!condT00) & (!condT01) & (!condT02) & condT03 &
      (next(pcDominoPlayer)=1) & (next(can_play)=can_play))
    | ((!condT00) & (!condT01) & (!condT02) & (!condT03) & condT04 &
      (next(pcDominoPlayer)=1) & (next(can_play)=can_play))
    | (condDefault0 &
      (next(pcDominoPlayer)=pcDominoPlayer) & (next(can_play)=can_play))
    )
)
```

(b) Corresponding `NuSMV` code.

Fig. 3. Small example of an LLFSMs that enables deliberation. LLFSMs use a cross-action language semantics that enables automatic translation to `NuSMV`.

our earlier infrastructure enables to install `gprolog` on the host (the docker container) and to cross-compile `gprolog` for the Pepper, the `gplc` compiler offers no options to specify target architectures. Thus, although we can invoke our C cross-compiler for the `.c` files in the prolog project, we need to compile the `.pl` (the pure `Prolog` files) on the Pepper .s assembly files:

1. `pl2wam -o step.wam step.pl`
2. `wam2ma -o step.ma step.wam`
3. `ma2asm -o step.s step.ma`

We then copy the `.s` files back to the Docker host and continue with the cross compilation of the `.s` files to `.o` (object) files using the Pepper cross-assembly compiler. Now all files are linked in the host using our cross-linker and we have an executable (binary) that once copied to the Pepper executes as expected.

6 Conclusions

Model-driven software development promises high levels of abstraction. In this paper we have shown how to incorporate a modern systems programming language, `Swift`, with the most advanced features of object orientation that enables complete execution time control. We incorporated features of functional and logic programming. All this enables a high-level of abstraction that define executable behaviour models. Moreover, we can automatically produce input code for a model-checker for formal verification. We believe that these paradigms significantly improve the quality of the robotic behaviours and the facility to produce correct code both in the value domain as well as the time domain. With the combination of paradigms, we are not advocating just a choice of programming language, we are enabling safety through adherence to widely accepted

design principles and philosophies. This is more important than the specific language; however, we believe the experience reported in this paper for enabling `Swift` as well as our Docker container constitute immediate, valuable contributions and pieces of knowledge for broadening the available tools in the RoboCup community.

References

1. The computer language benchmarks game, April 2018. https://benchmarksgame-team.pages.debian.net/benchmarksgame/compare/swift-gcc.html
2. Beekmans, G., Burgess, M., Dubbs, B.: Linux from scratch, April 2021. http://www.linuxfromscratch.org/lfs/
3. Bouyssounouse, B., Sifakis, J.: Programming languages for real-time systems. In: Bouyssounouse, B., Sifakis, J. (eds.) Embedded Systems Design. LNCS, vol. 3436, pp. 338–351. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31973-3_25
4. Brooks, R.: A robust layered control system for a mobile robot. IEEE J. Robot. Autom. **2**(1), 14–23 (1986)
5. Carrillo, M., Estivill-Castro, V., Rosenblueth, D.A.: Verification and simulation of time-domain properties for models of behaviour. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) MODELSWARD 2020. CCIS, vol. 1361, pp. 225–249. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67445-8_10
6. Diaz, D., Codognet, P.: The GNU prolog system and its implementation. In: ACM Symposium on Applied Computing, SAC 2000, NY, USA, vol. 2, pp. 728–732 (2000). https://doi.org/10.1145/338407.338553
7. Drusinsky, D.: Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking. Newnes (2006)
8. Eidhof, C., Kugler, F., Swierstra, W.: Functional Programming in Swift. Florian Kugler (2014)
9. Estivill-Castro, V., Ferrer-Mestres, J.: Path-finding in dynamic environments with PDDL-planners. In: 16th International Conference on Advanced Robotics, ICAR, pp. 1–7. IEEE (2013)
10. Estivill-Castro, V., Hexel, R.: Arrangements of finite-state machines-semantics, simulation, and model checking. In: International Conference on Model-Driven Engineering and Software Development, pp. 182–189. SCITEPRESS (2013)
11. Estivill-Castro, V., Hexel, R., Ramirez Regalado, A.: Architecture for logic programing with arrangements of finite-state machines. In: 1st CPS Week Workshop on Declarative Cyber-Physical Systems, DCPS, pp. 1–8. IEEE (2016)
12. Estivill-Castro, V., Hexel, R., Lusty, C.: High performance relaying of C++11 objects across processes and logic-labeled finite-state machines. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 182–194. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_16
13. Fuller, K.: `swiftenv` documentation – release 1.4.0, 10 September 2018. http://buildmedia.readthedocs.org/media/pdf/swiftenv/latest/swiftenv.pdf
14. GNU Project: GCC, the GNU Compiler Collection. https://gcc.gnu.org/
15. GNU Project: GNU Binutils. https://www.gnu.org/software/binutils/
16. GNU Project: GNU C Library (glibc). https://www.gnu.org/software/libc/

17. Harel, D.: On folk theorems. Commun. ACM **23**(7), 379–389 (1980)
18. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill, New York (1998)
19. Javaid, M., Estivill-Castro, V.: Explanations from a robotic partner build trust on the robot's decisions for collaborative human-humanoid interaction. Robotics **10**(1), 51 (2021)
20. The LLVM Project: The LLVM Compiler Infrastructure. https://llvm.org/
21. Lötzsch, M., Bach, J., Burkhard, H.-D., Jüngel, M.: Designing agent behavior with the extensible agent behavior specification language XABSL. In: Polani, D., Browning, B., Bonarini, A., Yoshida, K. (eds.) RoboCup 2003. LNCS (LNAI), vol. 3020, pp. 114–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25940-4_10
22. Mataric, M.J.: Behavior-based control: examples from navigation, learning, and group behavior. J. Exp. Theor. Artif. Intell. **9**, 323–336 (1997)
23. Mataric, M.J.: The Robotics Primer. MIT Press, Cambridge (2007)
24. Mataric, M.: Integration of representation into goal-driven behavior-based robots. IEEE Trans. Robot. Autom. **8**(3), 304–312 (1992)
25. McColl, C., Estivill-Castro, V. Hexel, R.: An OO and functional framework for versatile semantics of logic-labelled finite state machines. In: The 12th International Conference on Software Engineering Advances, ICSEA, pp. 238–243 (2017)
26. McColl, C.: SwiftFSM - a finite state machines scheduler. Honours thesis (2016)
27. McColl, C., Gilmore, E.: Swift on Pepper. https://github.com/mipalgu/SwiftOnPepper
28. McColl, C., Estivill-Castro, V., Hexel, R.: Versatile but precise semantics for logic-labelled finite state machines. Int. J. Adv. Softw. **11**(3 & 4), 227–238 (2018)
29. Nicolescu, M.: Lecture 6: Lecture notes autonomous mobile robots CPE 470/670 (2016). http://slideplayer.com/slide/5382727/
30. Nicolescu, M.N., Mataric, M.J.: Deriving and using abstract representation in behavior-based systems. In: The 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence, p. 1087. AAAI Press (2000)
31. Owen-Hill, A.: What is the best programming language for robotics? (2016). https://blog.robotiq.com/what-is-the-best-programming-language-for-robotics-0
32. Pilone, D., Pitman, N.: UML 2.0 in a Nutshell. O'Reilly Media, Inc. (2005)
33. Rebouças, M., Pinto, G., Ebert, F., Torres, W., Serebrenik, A., Castor, F.: An empirical study on the usage of the swift programming language. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 634–638 (2016)
34. Risler, M., von Stryk, O.: Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In: AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems, Estoril, Portugal (2008)
35. Rumbaugh, J., Blaha, M.R., Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modelling and Design. Prentice-Hall, Englewood Cliffs (1991)
36. Samek, M.: Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems, 2nd edn. Newnes, Newton (2008)
37. Scheepers, T.: Virtualization and containerization of application infrastructure: a comparison. In: 21st Twente Student Conference on IT (2014)
38. Sheu, P.C.Y., Xue, Q.: Intelligent Robotic Planning Systems. World Scientific Publishing, River Edge (1993)
39. Singh, H.: Speed performance between swift and objective-C. Int. J. Eng. Appl. Sci. Technol. **1**(10), 185–189 (2016). http://www.ijeast.com

40. Solt, P.: Swift vs. Objective-C: 10 reasons the future favors Swift. InfoWorld (2015). https://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html
41. Taylor, I.L.: A new elf linker. In: Proceedings of the GCC Developers' Summit, pp. 29–36 (2008). http://ols.fedoraproject.org/GCC/Reprints-2008/taylor-reprint.pdf