In Proc. of the 13th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (ISORC), Carmona, Spain, May 2010.
Revised version.

Model-based Programming of Modular Robots

David Arney[*], Sebastian Fischmeister[§], Insup Lee[*], Yoshihito Takashima[‡], Mark Yim[*]

[*]University of Pennsylvania
{arney,lee}@cis.upenn.edu, yim@grasp.upenn.edu

[§]University of Waterloo
sfischme@uwaterloo.ca

[‡]Microsoft
yoshita@microsoft.com

Abstract

Modular robots are a powerful concept for robotics. A modular robot consists of many individual modules so it can adjust its configuration to the problem. However, the fact that a modular robot consists of many individual modules makes it a highly distributed, highly concurrent real-time system, which are notoriously hard to program. In this work, we present our programming framework for writing control applications for modular robots. The framework includes a toolset that allows a model-based programing approach for control application of modular robots with code generation and verification. The framework is characterized by the following three features. First, it provides a complex programming model that is based on standard finite state machines extended in syntax and semantics to support communication, variables, and actions. Second, the framework provides compositionality at the hardware and at the software level and allows building the modular robot and its control application from small building blocks. And third, the framework supports formal verification of the control application to aid the gait and task developer in identifying problems and bugs before the deployment and testing on the physical robot.

1 Introduction

Modular robots are a powerful concept for robotics, because they adjust the tool to the problem and not the problem to the tool. A modular robot consists of many, possibly thousands, of modules that arrange themselves to form different shapes and collaborate to solve a specific task. For example, when it needs to go through a pipe it can use a snake shape and when it needs to scan the environment it can form a small tower with the sensors at the top of the tower. Figure 1 shows a CAD drawing of a modular robot built from four CKBot modules [14].

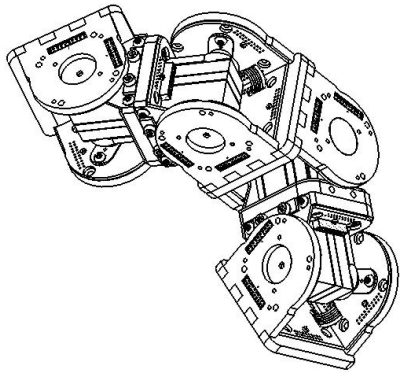


Figure 1: A robot built from four CKBot modules.

A modular robot is difficult to program because first, it needs to decide on what tool it wants to transform to—the tool means what shape and structure—and second, the modules must execute the right control program for the chosen tool. Writing the right control program for a single configuration is difficult by itself, because as one modular robot consists of possibly thousands of modules, together they form a massively distributed embedded real-time system. This work concentrates on the problem of how to program the control application given a robot configuration with its structure and its shape.

A number of related approaches exist for programming modular robots. The most basic form is a centralized control application executed on a workstation that remotely controls the modular robot as has been implemented in the early works of MTRAN [7] and PolyBot [3]. Gait tables [3] are two-dimensional tables in which each column represents a module state over time and each row represents a complete state of the modular robot with all its modules at a specific point in time. At run time, the modular robot executes the gait table one row after the other

and, for example, sets the motor angles on the individual modules accordingly. This behavior only allows programming stateless gaits, meaning that the gait cannot behave differently depending on some internal or external state, because the gait table offers no means of implementing branches. Phase Automata [19] and additional infrastructure [18] implement a finite state machine with phase offsets where each module executes the automaton using its offset but without verification support. Similar to Phase Automata are the mechanisms of specifying cyclic actions for each module [16] and hormone-based control [13]. Finally, TOTA [12] abstracts modules and programs at the level of context-aware tuples that can propagate through the network and can be applied to programming modular robots [11].

Our approach differs from related work in three key features. First, it provides a complex programming model based on standard finite state machines that have been extended in syntax and semantics to support synchronous and asynchronous communication, variables, and actions. Second, our approach provides compositionality at the hardware and at the software level. And third, our approach supports formal verification as a means to ease gait development. Once the developer specifies the state machines controlling the modular robot, he can use several verification tools to check the application for properties such as absence of deadlocks, continual physical movement, and motor position constraints in specific states.

The remainder of the paper is organized as follows: Section 2 introduces our model and our assumptions. Section 3 describes our programming framework with steps for the configuration design, control application development, the verification, the code generation, and the code integration with the runtime system. Section 4 evaluates our programming framework, and Section 5 closes the paper with our conclusions and future work.

2 Model and Assumptions

A modular robot consists of a number of individual modules which form the robot body. A robot configuration consists of a topology graph, the robots' body, and its structure. A topology graph is an undirected graph where the vertices are modules and the edges between vertices specify a physical and logical connection between two modules. The physical connection is established via screws that hold the two modules together. The logical connection is established via a bus connection that is shared by all linked modules and a point-to-point connection between two adjacent modules. The structure of a modular robot specifies for each connection the orientation of the modules and connections. The control application steering the motor is a distributed real-time program which consists of a set of periodic, preemptible tasks. Tasks communicate with other tasks locally via shared variables and remotely via channels, which have logical identifiers. All channels are mapped onto one shared communication medium. Tasks read and write their data from and to a shared variable space exclusively. Tasks run on processors, which are connected by a shared communication medium. Processors communicate with each other exclusively via the medium; particularly, we assume that there is no shared memory. We assume that time is given in discrete units. The communication medium provides a reliable broadcast service (a common assumption for such systems); therefore, either all processors receive a message or none of them do.

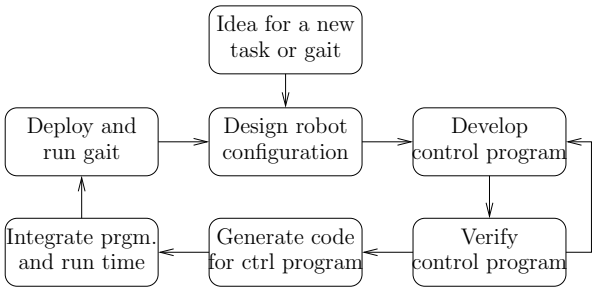


Figure 2: The development process for a new gait using our programming approach.

3 The Development Process

Figure 2 shows the development process for a new task or gait. The first step towards a new gait is for the developer to design a robot configuration that includes the topology graph, the body, and the structure. Then she develops the control program consisting of one or more state machines. The control program is already tailored to the robot configuration. Once the program is finished, the task developer can check the control program for properties such as deadlocks and continued motion. If the program fails the verification stage, then the developer will adapt the control application to make it pass. If the program passes the verification stage, then the developer can generate C code from the control application. The generated code is then integrated into the runtime systems executed by the modules, where one module can host one or more state machines. After the integration state, the compiled runtime systems are deployed on the modules and the developer can run the gait. If she is unsatisfied with the gait and wants to refine it, then she can return to the initial design phase.

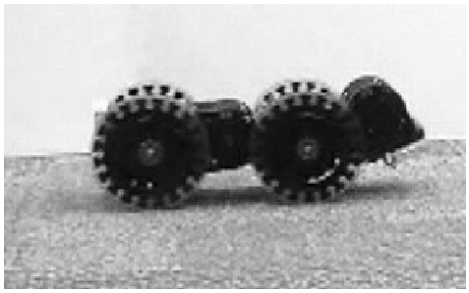


Figure 3: The configuration for the skate gait.

3.1 Robot Configuration Design

For our work, we use the CKBot modular robot developed at the University of Pennsylvania. Each module can be connected to another module at four locations (front, rear, right, left) with two different angles (straight, at zero degrees, and tilted at ninety degrees). An individual module hosts a microprocessor, a hobby servo, a bus communication interface, and an extension slot for additional sensor and actuators. The microprocessor is a PIC18F2680; the important specifications for this work are its 64KB program memory, about 3KB of RAM for data storage, 1KB of EEPROM, four timer modules (two 8bit and two 16bit), and its ECAN module. The chip is driven by a 20MHz oscillator, so the single instruction speed is 50ns which results in a computation power of 5MIPS. The hobby servo drives a rotary axis with 180 degrees of freedom. It has a duty cycle of 18ms. Modules can communicate via broadcast on the CAN bus. One of the modules will have a battery pack mounted on its connection interface to provide power to the whole modular robot. For further details on the hardware platform, see [14].

The running example for this paper is the skate gait. The idea is that the modular robot has a board with passive wheels and uses a foot formed by modules to push the board forward. The robot body consists of four modules (see Figure 3); two standard ones (labeled *Spine* and *Foot*) and two modules (labeled *AxisOne* and *AxisTwo*) with passive wheels mounted on each side. The robot topology is a graph with its vertex set containing all four modules and its edge set containing the edges $(AxisOne, Spine), (Spine, AxisTwo)$, and $(AxisTwo, Spine)$. Finally, the robot's structure specifies that all edges are connected at a zero degree angle. From left to right, the module *AxisOne* and *AxisTwo* have passive wheels mounted on each side. The second module from the left, *Spine*, acts as spacer between the two wheels and forms the robot's spine. Finally, the fourth module from the left, *Foot*, realizes the foot pushing the robot forward.

The intuition for the gait is shown in three subfigures of Figure 4. When the *Spine* module contracts, then the *Foot* module touches the ground. After it touches the ground, the module *Foot* performs a kick that pushes the robot forward. Simultaneously, the *Spine* module stretches to allow the foot to prepare for the next kick. The passive wheels allow the robot to scoot forward once it gained momentum from the kick.

3.2 Control Application Development

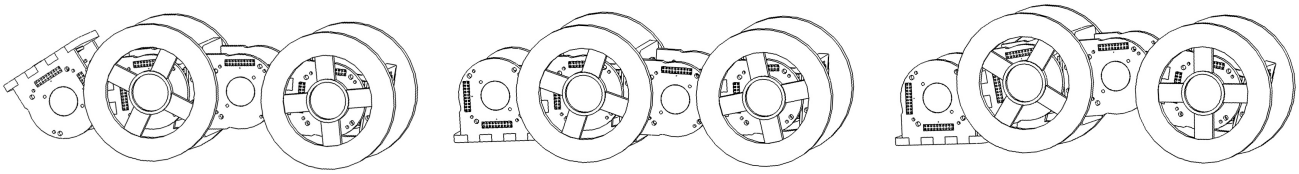
In the development stage, the task developer takes the robot configuration and specifies the control application for a gait using state machines. The language for specifying these state machines is EFSM (extended finite state machines). The EFSM language is designed to be a simple way of representing state machines. An EFSM consists of a set of states connected by transitions. Transitions must have a guard condition and may also be tagged with a set of actions which are performed when the transition is taken. Two or more state machines can communicate via communication channels. Communication channels in EFSM systems may be synchronous or asynchronous, with any finite size buffer, may pass values, and may be blocking or non-blocking. For complete details, including a grammar for the language and full description of the semantics, see [1].

The EFSM system for the skate gait is depicted in Figure 5. It consists of three state machines: gait controller, foot controller, and spine controller. The three machines communicate using the channels 'foot' and 'spine'. Each EFSM in the system has a number of states, shown as rounded boxes with the state's name inside. Transitions are represented as lines between states and are labeled with a guard, an arrow, and a (possibly empty) set of actions. EFSMs wait in a state until an outgoing transition becomes enabled, meaning that the guard of at least one transition becomes *true*. Then, they switch to the destination state of the transition and perform the transition's actions. The EFSMs represent the high-level behavior of system components. Low-level behavior is implemented in the runtime system presented in Section 3.6.1.

3.3 Verification of the Control Application

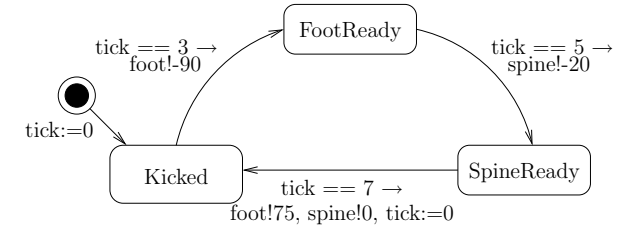
In the verification stage, the task developer takes the specified control application and checks it for properties such as deadlocks. This verification aids and speeds up the development, because it allows the developer to identify bugs in the control application before he generates the code, deploys it on the robot, and runs the gait.

For verification, the EFSM toolset supports automatic

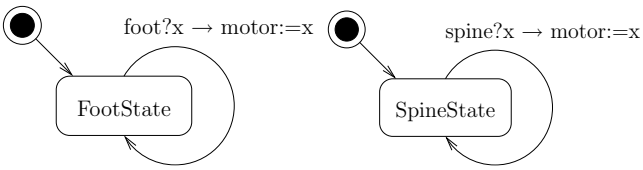


(a) *Kicked*: foot kicked, spine stretched. (b) *FootReady*: foot in position. (c) *SpineReady*: crouched, foot on ground.

Figure 4: The movement of the skate gait.



(a) The skate gait controller.



(b) The foot controller. (c) The spine controller.

Figure 5: The state machines for the skate gait.

translation into the input formats of Spin [6] and UPPAAL [9]. We also translate guards into DIMACS format so we can use boolean satisfiability checkers to test for determinism and totality.

3.3.1 Checking with SAT Solvers

Boolean satisfiability checkers, or SAT solvers, are tools which take a boolean formula over a set of variables and decide if there is some assignment of values to variables which makes the formula true. The EFSM toolset includes code from the SAT4J project[1], and supports transforming guards into boolean formulas it can check. This allows us to easily check EFSMs for nondeterminism and totality. The toolset also supports exporting the formulas in DIMACS format so that other SAT solvers can be used.

An EFSM is nondeterministic if there exists some state with two or more outgoing transitions which may be enabled at the same time. We check for nondeterminism by finding the set of guard conditions $g_1, g_2, \ldots, g_n$ for the transitions from each state, and then checking that the expression $g_i \wedge g_j$ is unsatisfiable for any $i < j <= n$. The check is performed by feeding the expression into a boolean satisfiability checker, for instance ZCHAFF. The input format for ZCHAFF (and most other SAT solvers) is DIMACS, which is a syntax for expressions in conjunctive normal form (CNF). In order to use ZCHAFF to check these expressions, we must first convert the guards into CNF. For full details of how the conversion is done, see [1].

Another property we can check for is totality, or completeness. This is a way of saying that the system does not get stuck. That is, at each state S with outgoing transitions $t_1, t_2, \ldots, t_n$ and corresponding guards $g_1, g_2, \ldots, g_n$, the property $(g_1 \vee g_2 \vee \ldots \vee g_n) \rightarrow T$ holds.

3.3.2 Translating the Model for Spin

Spin is a popular model checking program which uses the input language Promela. The translator allows the user to convert an EFSM system into Promela code. We use Spin to check some properties which we cannot check using UPPAAL such as "The module *Foot* will always move". The translator currently supports EFSM systems which use synchronization channels only. Spin supports all of the operators used in guards and actions, so translating these is simply a matter of making slight changes to syntax.

3.3.3 Translating the Model for UPPAAL

The EFSM toolset also contains a translator which can convert EFSM systems into UPPAAL. UPPAAL supports only synchronous channels, so any value passing must take place through shared variables.

The UPPAAL translation of the skate gait system must use shared variables to pass values instead of using value passing channels. The semantics of UPPAAL's shared variables, without any synchronization, match the semantics of communications channels in the implementation. In the UPPAAL model for the skate gait system, the channels *foot* and *spine* become shared variables with the same name.

[1] Available from www.sat4j.org

3.4 Verifying the Skate Gait

We check a variety of properties of the EFSM system of the skate gait. Many of these, such as deadlock, could be checked using either Spin or UPPAAL. Other properties cannot be stated in CTL and thus require Spin.

For our skate gait, we check the following properties:

- **Determinism and totality.** We used the checks described above to check that the EFSMs were deterministic and total.
- **Absence of dead locks.** The absence of deadlocks can be shown with either tool. We used UPPAAL to check the property AG(!deadlock).
- **Absence of livelocks.** Livelocks [5] occur when the system has an infinite execution path where it never waits for input. This system trivially does not have any livelocks since every transition checks whether the tick variable is equal to a different number. In general, checking for livelocks will require a set of formulas tailored to the specific state machine being considered. Spin allows marking states with acceptance labels to assert that they may not be part of an infinite loop, which is useful for this check.
- **All states are reachable.** Every state in the model should be reachable. Unreachable states usually indicate an error in programming. These can be found by checking that $EF($ gait.*stateName* $)$ for each state in the system.
- **Foot will always move.** The module *Foot* should always move back and forth. Using Spin, we can check this with the property AF foot $== -90$ && AF foot $== 75$. We also check that the variable *foot* is -90 until it becomes 75 and vice versa $AG(foot == -90 \text{ U } foot == 75)$. Together, these properties ensure that the variable *foot* will change between these two values (and only these two values) infinitely often, and as the module *Foot* uses the variable *foot* as its motor value, it will always move back and forth.
- **Spine will always move.** This is very similar to checking that the foot will always move, and the formulae are essentially the same.
- **Foot does not brake.** We want to ensure that the module *Foot* will not drag while the robot is rolling forward, as this would quickly stop it. The gait controller state machine waits in state *Kicked* for three seconds while the robot rolls. If the module *Foot* is always raised (i.e., the angular value is greater than a threshold) while the gait controller is in this state, then the foot never drags. We check this property using UPPAAL and AG(gait.kicked imply gait.foot $== 75$).
- **Are foot and spine always ready before the kick?** We want to ensure that the modules *Foot* and *Spine* are both in the correct position before the kick happens. This can be done in UPPAAL using the property AG(gait.spineready imply gait.foot $== -90$ and gait.spine $== -20$).
- **The robot will always try to move forward.** Since we have shown that the Foot and Spine will always move, if we show that the movements are coordinated in the right way then we can claim that the robot will always try to move forward. We used the following three formulas in UPPAAL: AG(gait.kicked imply (gait.foot $== 75$ and gait.spine $== 0$)) AG(gait.footready imply (gait.foot $== -90$ and gait.spine$== 0$)) AG(gait.spineready imply (gait.foot $== -90$ and gait.spine$== -20$))

3.5 Code Generation

Our toolset currently supports generating Java and C code, but for the modular robots we only use C code. During code generation, the programmer has the opportunity to link input and output variables to functions defined in a code library. These functions are written by the programmer and do low-level tasks. This allows the EFSM to treat low-level tasks as abstract statements which can be filled in by the code generator.

Code generation results in one program per EFSM. Each program is split into a header file and a code file. The header file contains all the global channel definitions used for communication between multiple state machines. We also generate a code segment from the EFSM. The generated source code is split into two parts. In the first part, the code declares all necessary variables for the state machine. The variables may have local or global scope. Variables with global scope are, for example, the current state or variables declared in the state machine. Variables with local scope are basically auxiliary variables such as *transition_taken*.

In the second part, the code realizes the state machine which takes the form of a series of if statements, surrounded by an outside while loop. We chose to implement run-to-completion semantics, which means that the state machine is allowed to continue taking transitions as long as it can, until it reaches a state where no transitions are enabled. Listing 1 shows the generated code for the gait controller specified in Figure 5. The first part declares the local variables *currentState*, *tick*, and *transition_taken*. This is followed by the code for the state ma-

chine where the while loop realizes the run-to-completion semantics with the variable *transition_taken*. Each time a transition is taken, this variable is set to one and the loop runs again. Inside a transition, the generated code implements what has been specified in the EFSM. For example between Lines 11 and 13, the state machine switches from state *FootReady* to state *SpineReady* and transmits a message with the contents 20 on channel *SPINE*. It also updates the variable *currentState*, and declares that a transition has been taken.

If no transition is taken, then the while loop exists and the program will wait for the next OS *tick*. At this time, the program will rerun the generated code and take transitions.

```
1 // local variable declaration
  static INT8U currentState = 2;
  static INT8U tick = 0;
  auto INT8U transition_taken=1;

6 // state machine
  while (transition_taken) {
     transition_taken = 0;
      if ((currentState == 2) && (tick == 5)) {
        PRETRANSITIONHOOK;
11      sendMessage( SPINE, -20, sizeof(INT16S) );
        currentState = 3;
        transition_taken = 1;
        POSTTRANSITIONHOOK;
      }
16    if ((currentState == 3) && (tick == 7)) {
        PRETRANSITIONHOOK;
        sendMessage( FOOT, 75, sizeof(INT16S) );
        sendMessage( SPINE, 0, sizeof(INT16S) );
        tick = 0;
21      currentState = 1;
        transition_taken = 1;
        POSTTRANSITIONHOOK;
      }
      if ((currentState == 1) && (tick == 3)) {
26      PRETRANSITIONHOOK;
        sendMessage( FOOT, -90, sizeof(INT16S) );
        currentState = 2;
        transition_taken = 1;
        POSTTRANSITIONHOOK;
31    }
  }
```

Listing 1: C code generated from *gait.efsm*.

There are two ways that implementation-specific details are introduced into generated code. These are through mapping from variables to library code functions as mentioned before, and through the transition hooks. The system uses two transition hooks: *PRETRANSITIONHOOK* and *POSTTRANSITIONHOOK*. These are pieces of code which are defined in the code library and inlined by the compiler. *PRETRANSITIONHOOK* is inserted into each if statement after the if condition has been met but before any actions are done. *POSTTRANSITIONHOOK* is placed after all of the actions have finished and the current state has been updated. These hooks can be used, for example, to change the run-to-completion semantics to one-transition-per-tick semantics

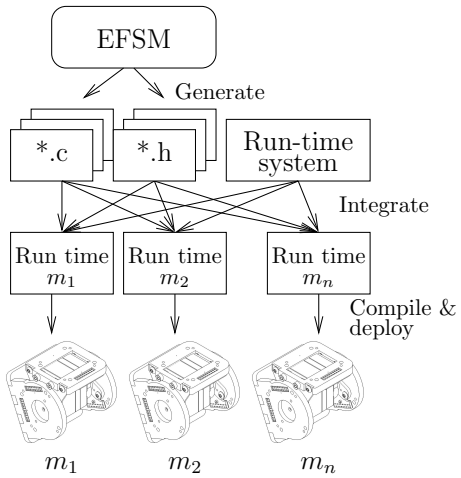


Figure 6: Overview of the integration process.

where the *POSTTRANSITIONHOOK* introduces a *break* statement. Another frequent usage is to introduce debug code that takes a snapshot of important system variables and stores them in the EEPROM.

3.6 Code Integration

In the code integration stage, the task developer integrates the code segments generated from the EFSM system into the runtime system and thereby produces runnable programs for each module. Figure 6 provides an overview of the integration stage. First, the developer generates code from the EFSM system. This results in many header and code files. Second, the developer integrates state machines and the runtime system into one specific runtime system per module. And finally, the developer compiles the run times and deploys them on the modules.

3.6.1 Runtime System

The robot control application is a concurrent, highly distributed real-time application with the following requirements for the runtime system.

- **Precise temporal control.** The robot's actuators need to be controlled precisely so dynamic gaits such as rolling or swinging motion can be realized accurately. Also, the robot's sensors need to be sampled and converted precisely so the developer can predict the robot's behavior as he programs it.
- **Preserve state machine semantics.** The task developer specifies a set of state machines that describe the robot's behavior. These state machines have specific semantics as described in Section 3.2. When the

run time executes the code that is generated from these state machines, it must run the code in a way that preserves the state machine's semantics. Otherwise, the implementation might have different properties than the specification and this renders the verification useless.

- **Compositionality.** A task developer can assign more than one state machine to a single module, meaning that that module's runtime is composed of multiple state machines that execute concurrently. As this module executes these state machines, it must preserve the original properties of each state machine, otherwise it again renders the verification useless.

Our runtime system for modular robots meets all three requirements. For practical reasons, we check the whole system for schedulability once using rate monotonic scheduling after it has been composed, instead of using a compositional approach as suggested in [15]. The system consists of four major components: the module control application implemented as a set of tasks, the variable space, the RoboVM, and the transceiver. The *module control program* exists on each module, and it can behave differently on each of them. The program consists of multiple tasks that run concurrently.

The *variable space* represents a variable storage that can be accessed via a set/get interface. The variable space stores for each variable a set of tuples $\langle x, val, ts\rangle$ where x identifies the variable, val specifies the numerical value assigned to the variable, and ts is a time stamp specifying the point in time at which the particular value becomes valid. The variable space can store multiples of those tuples for a single variable. Whenever it handles a request that reads a variable value, it updates the tuples and removes those that are dispensable. A tuple $\langle x, val, ts\rangle$ becomes dispensable if there exists another tuple $\langle x, val', ts'\rangle$ where $ts' > ts$ and $now > ts'$ where now represent the current clock value. The set function adds a tuple $\langle x, val, ts\rangle$ to the storage and has the signature $set(id, val, ts)$ where id identifies the variable, val is the new value, and ts is a time stamp specifying the point in time when the value becomes valid. The get function returns the value that most recently became valid given a particular time stamp and has the signature $get(id, ts)$ where id identifies the variable and ts specifies a time stamp. If no value is valid at the timestamp ts, then a default value is returned. We assume that an invocation of the *get* function never reads an already dispensed value, meaning that the parameter ts is never smaller than the current clock value. And we assume that all stored tuples $\langle x, val, ts\rangle$ and $\langle x, val', ts\rangle$ with the same variable identifier x and the same validity time stamp ts also have the same value, thus, $val = val'$. These two assumptions do not restrict the system in any way, because the tasks access the variables using a macro that fills in the local clock value for ts, which monotonically increases over time, and at most one task writes a variable while several other tasks may read it. The variable space is similar to the operation of the variable space described in [2, 8].

The *RoboVM* coordinates the tasks' release schedule and handles channel communication. Tasks are recurrent and they are released in periodic intervals. The RoboVM releases the tasks based on their specified frequency. This frequency is the *tick* as shown in Section 3.5. Tasks can use channels to communicate with tasks running on remote modules. To transmit a message on the channel, the task executes the function *sendMessage(channelId, value, size)* where *channelID* identifies the channel, *value* gives the value to be transmitted, and *size* gives the storage size of the value. The RoboVM receives the message request, annotates the message with a validity time stamp, and passes it to the CAN transceiver. It calculates the validity time stamp based on the execution frequency of the task sending the message. The validity time stamp is stored as a relative time. On the receiving side, the RoboVM unpacks the message, translates the relative time into a local absolute time, and invokes the set function in the variable space. The sending RoboVM also adds a storage tuple to the local variable space similar to the receiving RoboVM.

The *Transceiver* handles message transmission and reception. It implements an event triggered approach, where whenever a new CAN message becomes available, it hands the message to the RoboVM for further processing. Whenever the RoboVM wants to transmit a message, it queues this message in the transceiver's output queue from which the transceiver continually dequeues messages and places them in the CAN send buffer as this buffer becomes empty.

3.6.2 Single and Compositional Integration

An EFSM system consists of potentially many state machines and during the code generation process each state machine is translated into one header and one code segment. To build a functioning system, these segments must be integrated, made an executable, and then deployed on the robot hardware modules. There are two interesting cases to be considered for the integration: running exactly one state machine on one module and running more than one state machine on one module. We first consider the first case and then elaborate on the second one.

Integrating a single state machine for a module is straightforward. The runtime portion provides task skele-

tons which are tasks without any functionality in them. The generated code segments are ANSI-C compliant, so they can be inserted into the task skeleton via a preprocessor directive and without any manual intervention. The generated header segment is included right before the task skeleton, and the code segment is included inside the task skeleton. The task has to be registered in the RoboVM with its execution frequency, that is the *tick* time has to be declared.

Integrating multiple state machines into the run time for one module is also straightforward, but requires an additional step. Each state machine uses its own task skeleton, and the generated code segments for one state machine are integrated as mentioned above. Then each task is individually registered in the RoboVM and each task declares its execution frequency. Additionally, the developer has to assign a priority to each task. We use rate monotonic scheduling, meaning that the tasks with the highest execution frequency get the highest priority.

To guarantee compositionality, we have to consider compositionality in the value and the resource domain. In the value domain, the variable space provides compositionality, because even if a task finishes early, its output values do not become valid until its deadline, which equals its next release. It therefore enforces semantics that have been shown to provide compositionality in [4]. In the resource domain, for simplicity reasons, we check the composed systems once and not as we compose it. For each state machine in the composed run time, we extract the longest possible path and get the execution time of that path using the Microchip simulator. Then, we plug the values into the schedulability condition for rate-monotonic scheduling [10] $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n * (\sqrt[n]{2} - 1)$, and check whether the system is schedulable. However, at the current level of complexity of our programmed tasks, it is unlikely to run into a schedulability problem. The execution time for the state machines has to be slow enough for the motor to stabilize at the new angle value, which is about 700ms between motor angle updates. See Section 4 for more details on this.

4 Evaluation

Evaluating a programming framework is difficult per se, because only its usage over time shows whether the framework is really successful. However, we are convinced that we set the basis for a successful programming framework for the following reasons. First, the toolset and the runtime system are fully implemented and run on the target platform. We can easily program tasks for the modular robot, develop new gaits, and in most cases create the executable for a module by pushing a few buttons.

Second, the programming framework requires little understanding of the hardware platform as it provides high-level abstractions. For example, a task developer programming a new gait uses channels to model communication in the state machines. The code generator then creates code for these channels that handles the CAN communication registers, and receives and sends messages.

Third, the overhead introduced by the runtime system in terms of memory and execution time is sufficiently low to run complex state machines with potentially many, computationally intensive transitions per tick and many variables. We measured the computational overhead using the stop watch and the execution cycle counter built into the MPLAB simulator. The computation time of switching from one task to another is 999.2us which result from 503.2us in the runtime system, 230us from the operating system kernel, and two context switches with each 133us. Theoretically, the fastest execution rate is 1KHz, however, such a task may not have more than sixteen assembly instructions and all maskable interrupts should be turned off. Otherwise, this task will still be running at its next release. For more details on the measurements see [17]. This computation overhead is acceptable, because tasks in the modular robot are much slower that 1KHz. For example the motor driver runs at a frequency of 55Hz, and tasks executing a state machine may not execute faster than with a frequency of 1.4Hz, because it takes the motor about 700ms to stabilize after a 180 degree turn. Regarding memory overhead, the runtime system uses 15.745 bytes of program memory and 1.376 bytes of data memory. The data memory is mainly for the large stacks (256 bytes) for the four tasks: RoboVM, transceiver, and two task skeletons for state machines. We require large task stacks, because our CAN library stores messages contents in the current active task's stack space. So, the data memory overhead introduced by the runtime without the stacks is 376 bytes. Regardless, whether with or without large stack sizes, the runtime system fits conveniently on the microprocessor and leaves enough memory for the control application.

5 Conclusion

A modular robot consists of many individual modules that communicate, interact, and together solve a problem. The control application for such a robot is a highly distributed, highly concurrent real-time application, because of the high number of modules, the number of concurrent processes in the system, and the physical interaction through actuators and sensors.

In this work, we presented a model-based approach for writing control applications for modular robots. The center piece is our programming framework that is character-

ized by three features. First, it provides an intuitive programming model based on extended finite state machines. It supports standard state machines and also synchronous and asynchronous communication, variables, and actions. Second, the framework provides compositionality at the software and at the hardware level, which allows one to write complex control applications composed from small building blocks. And third, the framework supports formal verification to aid the developer and speed up the development process, because certain bugs can be detected early in the development cycle.

The aim for the work was to develop an easy-to-use, tool-oriented programming framework for writing control applications for modular robots. The toolset and the runtime system have been implemented and the whole system works and runs. We believe that our system meets the goal for the following reasons: First, we can program complex control application using the well known notion of state machines, which make it easy to learn and to become a task developer. And second, the runtime system introduces a low enough overhead to allow the developer to program complex state machines with many transitions and many local variables.

The presented programming framework is still an initial step towards better high-level language support for programming modular robots. We plan to explore more higher-level languages from which we can generate state machines as input to our framework.

Acknowledgements

We would like to thank Jimmy Sastra for the initial idea of the skate gait, Mustafa Karabas for designing the basic CAD image of one module, and Thomas Mather and Michael Park for their crucial EE support during the development of the runtime system.

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, NSF CNS-0721541, NSF CNS-0720703, and NSF CNS-0720518.

References

[1] A. Easwaran, D. Arney, and I. Lee. CEFSM (Communicating Extended Finite State Machine). www.cis.upenn.edu/hasten/CEFSM.ps.

[2] S. Fischmeister, O. Sokolsky, and I. Lee. Network-Code Machine: Programmable Real-Time Communication Schedules. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 311–324, San Jose, United States, October 2006.

[3] A. Golovinsky, M. Yim, Y. Zhang, C. Eldershaw, and D. Duff. PolyBot and PolyKinetic System: A Modular Robotic Platform for Education. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, pages 1381–1386. IEEE Press, April 2004.

[4] T. A. Henzinger, C. M. Kirsch, and B. Horowitz. Giotto: A Time-triggered Language for Embedded Programming. In T. A. Henzinger and C. M. Kirsch, editors, *Proc. of the 1st International Workshop on Embedded Software (EMSOFT)*, number 2211 in LNCS. Springer, October 2001.

[5] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[6] G.J. Holzmann. Proving properties of concurrent systems with spin. In *Proc. CONCUR95, 6th Intern. Conf. on Concurrency Theory*, Philadelphia, PA., August 1995. LNCS 962, Springer-Verlag.

[7] Akiya Kamimura, Haruhisa Kurokawa, Eiichi Yoshida, Satoshi Murata, Kohji Tomita, and Shigeru Kokaji. Automatic Locomotion Design and Experiments for a Modular Robotic System. *IEEE/ASME Transactions on Mechatronics*, 10(3):314–325, 2005.

[8] Gregor König. Using Interpreters for Scheduling Network Communication in Distributed Real-Time Systems. Master's thesis, Salzburg University, Jakob-Haringer-Str. 2, 5020 Salzburg, Austria, March 2005.

[9] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[10] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] Marco Mamei and Franco Zambonelli. Programming modular robots with the TOTA middleware. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 485–487, New York, NY, USA, 2006. ACM.

[12] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):1–56, 2009.

[13] Behnam Salemi, Wei-Min Shen, and Peter Will. Hormone-Controlled Metamorphic Robots. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, pages 4194–4199, 2001.

[14] Jimmy Sastra, Sachin Chitta, and Mark Yim. Dynamic Rolling for a Modular Loop Robot. In *Proc. of the Int. Symp. on Experimental Robotics*, Rio de Janeiro, June 2006.

[15] I. Shin and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[16] Kasper Sty, Wei-Min Shen, and Peter Will. Global locomotion from local interaction in self-reconfigurable robots. In *Proc. of the 7th Intl. Conf. on Intelligent Autonomous Systems (IAS-7)*, pages 309–316, 2002.

[17] Yashihito Takashima. Towards Improving Time and Value Determinism in Distributed Real-Time Systems. Master's thesis, University of Pennsylvania, 2006.

[18] Y. Zhang, M. Yim, K. Roufas, C. Eldershaw, and D. Duff. Attribute/Service Model: Design Patterns for Efficient Coordination of Distributed Sensors, Actuators and Tasks in Embedded Systems. In *Proc. of the Workshop on Embedded System Codesign (ESCODES)*, 2002.

[19] Ying Zhang, Mark Yim, Craig Eldershaw, Dave Duff, and Kimon Roufas. Phase automata: a programming model of locomotion gaits for scalable chain-type modular robots. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2442–2447, Las Vegas, NV, October 2003.