Generating Arduino C Codes from *Mediator*

Yi Li and Meng Sun

LMAM and Department of Informatics, School of Mathematical Sciences,
Peking University, Beijing, China
`liyi_math@pku.edu.cn, sunmeng@math.pku.edu.cn`

Abstract. Manual encoding is exceedingly time consuming and error prone, and has become a huge obstacle between reliable software models and trustworthy computer programs. To deal with this problem, dozens of code generators are developed to automatically convert different models into executable codes. In this paper we present a new code generator for the component-based modeling language *Mediator*. It aims to generate platform-dependent (Arduino in this case) programs that can be directly downloaded to the hardware without any manual adaption. We also present a case study where we use *Mediator* to develop a wheeled-robot controller, generate the corresponding program through our code generator, which has been successfully executed on an Arduino-based robot platform.

Keywords: Code Generation, Mediator, Arduino, Component-based Modeling

1 Introduction

With IoT techniques sweeping around the world, software systems are becoming more complicated, distributed and safety-critical, and thus the development of such systems is becoming notoriously difficult. Small failures in daily-used software, such as smart house controllers, payment applications, etc., may lead to severe butterfly effect. Under such circumstances, various approaches have been proposed in the past decades to help software development, such as object-oriented programming [19], aspect-oriented programming [14], component-based modeling and development [20], and so on. Among these approaches, component-based modeling is extremely popular and helpful in the development of embedded systems [11] and service-oriented applications [3].

Mediator [15] is a new component-based modeling language that provides an automata-based formal semantics and supports hierarchical modeling. With the help of a full-featured type system and powerful coordination mechanisms, this language can be used by both domain-specific experts and software engineers to guarantee the reliability of software system models.

However, such powerful modeling languages can only help with the correctness of high-level models. In practice, lots of software errors are caused by the inconsistency between implementations and models. And it turns out that manual implementations do not always precisely follow abstract models and designs, especially for systems with high complexity. To address this problem, automatic code generation has been proposed to avoid errors caused by human activities in the implementation process [9]. In this paper, we present an algorithm that generates C code from *Mediator* models, which can be directly compiled and executed on Arduino [16], a popular open-source embedded platform.

Importance of code generation has been uncovered for a long time. As a result, a large number of formal and industrial code generation tools have been built for different target platforms. For example, Event-B [12] and SCADE [8] are very popular formal tools that can generate executable codes from abstract models. The code generator in SCADE is especially famous for its reliability, but its scalablity is restricted by the Esterel language it accepts. As a synchronous language, Esterel only formalizes a single embedded control loop, which makes it hard to model concurrent and timed behavior. Furthermore, both Event-B and SCADE code generators only aim at x86 platform, i.e. they cannot generate code that directly works on embedded systems.

On the other hand, Ptolemy[18], MATLAB Simulink Toolbox[13] and LabVIEW[17] are the most famous industrial modeling tools that support platform-dependent code generation. These tools have large number of libraries and plug-ins that almost cover all commonly-used embedded platforms and programming languages. Nevertheless, power of these tools also becomes limitation when we try to perform testing or verification techniques on their models. For example, Ptolemy uses standard JAVA as its semantics, and Simulink enables users to write components directly through MATLAB language. As far as we know, models directly written in such full-featured programming languages, with loop and dynamic memory allocation, are very hard to be verified.

The rest of this paper is structured as follows: Section 2 briefly introduces the *Mediator* language and the Arduino platform. Then in Section 3 we illustrate how the *Mediator*-Arduino code generator works. Section 4 presents the wheeled-robot controller as a case study. Finally, Section 5 concludes the paper.

2 Background

2.1 *Mediator*

Mediator is a component-based modeling language proposed in [15]. As a hierarchical modeling language, *Mediator* provides formalisms for both high-level *system* layouts and low-level *automata*-based behavior units.

As implied by the name, the language tries to provide a mediator through different language elements. For example, *systems* are designed for software engineers who may have no background about formal methods, which makes it easier to construct software models with reliable components and connectors. On the other hand, these reliable components and connectors are supposed to be built using *automata* by researchers who know formal methods well.

Syntax tree of a typical *Mediator* program is defined as follows.

$$\boxed{\langle \textit{program} \rangle ::= (\langle \textit{typedef} \rangle \mid \langle \textit{function} \rangle \mid \langle \textit{automaton} \rangle \mid \langle \textit{system} \rangle)^*}$$

*Typedef*s are aliases for types. *Function*s are definitions of custom (or *native*, which will be interpreted later) functions. *Automaton*s and *system*s are the core modeling elements in *Mediator*. They are also called *entities* since they share the same declaration form.

Type System. *Mediator* provides various data types that are widely used in different formal modeling languages and programming languages. These data types are categorized as *primitive types* and *composite types*, which are presented in Table 1 and 2, respectively. [1]

Table 1. Primitive Data Types

Name	Declaration	Term Example
Integer	`int`	`-1,0,1`
Bounded Integer	`int lowerBound .. upperBound`	`-1,0,1`
Double	`double`	`0.1, 1E-3`
Boolean	`bool`	`true, false`
Character	`char`	`'a', 'b'`
Enumeration	`enum` $\texttt{item}_1$`, ..., `$\texttt{item}_n$	`enumname.item`

Table 1 shows the primitive types supported by *Mediator*: *integers and bounded integers, double values, boolean values, single characters* and *finite enumerations.*

Table 2. Composite Data Types (`T` denotes an arbitrary data type)

Name	Declaration
Tuple	$(\texttt{T}_1,\ldots,\texttt{T}_n)$
Union	$\texttt{T}_1 \mid \ldots \mid \texttt{T}_n$
Array	`T [length]`
List	`T []`
Map	`map` $[\texttt{T}_{key}]$ $\texttt{T}_{value}$
Struct	`struct` $\{$ $\texttt{field}_1$:$\texttt{T}_1$,..., $\texttt{field}_n$:$\texttt{T}_n$ $\}$
Initialized	$\texttt{T}_{base}$ `init term`

Composite types are used to construct complex data types from simpler ones. Such composite patterns in Table 2 are interpreted as follows:

- *Tuple*. The *tuple* operator ',' can be used to construct a finite tuple type with several base types.
- *Union*. The *union* operator '|' is designed to combine different types as a more complicated one.

[1] Some notations in Table 1 and 2 are slightly different from the original language proposal in [15] since both the tool and language are still being frequently updated.

– *Array* and *List.* An *array* $T[n]$ is a finite ordered collection containing exactly n elements of type T. Moreover, a *list* is an array of which the capacity is not specified, i.e. a list is a dynamic array.
– *Map.* A *map* $[T_{key}]\ T_{val}$ is a dictionary that maps a key of type T_{key} to a value of type T_{val}.
– *Struct.* A *struct* $\{field_1 : T_1, \cdots, field_n : T_n\}$ contains a finite number of fields, each has a unique identifier $field_i$ and a particular type T_i.
– *Initialized.* An initialized type is used to specify the default value of a type T_{base} with `term`.

Parameter Types. In some situations we may hope to reuse an automaton or a system with the help of generalization. For example, an encrypted communication system supports different encryption algorithms encapsulated as parameter functions or components. Parameter types make it possible to take functions and entity interfaces as template parameters. *Mediator* supports two parameter types:

1. *Interface*, denoted by $\texttt{interface (port}_1\texttt{:T}_1\texttt{,}\cdots\texttt{,port}_n\texttt{:T}_n\texttt{)}$, defines a parameter that could be any *automaton* or *system* with exactly the same interface (i.e. number, types and directions of the ports perfectly match the declaration). Interfaces are only used in templates of *systems*.
2. *Function Type*, denoted by $\texttt{func (arg}_1\texttt{:T}_1\texttt{,}\cdots\texttt{,arg}_n\texttt{:T}_n\texttt{):T}$, defines a function that has the argument types $\texttt{T}_1, \cdots, \texttt{T}_n$ and result type `T`. Functions are permitted to appear in templates of *other functions*, *automata* and *systems*.

Parameter types can only be used in template parameters. It is impossible to declare a function or an interface as a local variable.

Functions. *Mediator* supports two types of functions, *common* functions and *native* functions. The syntax of functions is shown as follows.

$$
\begin{aligned}
\langle \textit{funcDecl} \rangle &::= \texttt{native}^{?}\ \texttt{function}\ \langle \textit{template} \rangle^{?}\ \langle \textit{identifier} \rangle\ \langle \textit{funcInterface} \rangle\ \{ \\
&\quad\ (\ \texttt{variables}\ \{\ \langle \textit{varDecl} \rangle^{*}\ \}\ )^{?} \\
&\quad\ \texttt{statements}\ \{\ \langle \textit{assignStmt} \rangle^{*}\ \langle \textit{returnStmt} \rangle\ \} \\
\langle \textit{funcInterface} \rangle &::= \texttt{(}\ (\ \langle \textit{identifier} \rangle\ \texttt{:}\ \langle \textit{type} \rangle\ )^{*}\ \texttt{)}\ \texttt{:}\ \langle \textit{type} \rangle \\
\langle \textit{assignStmt} \rangle &::= \langle \textit{term} \rangle\ (\ \texttt{,}\ \langle \textit{term} \rangle\ )^{*}\ \texttt{:=}\ \langle \textit{term} \rangle\ (\ \texttt{,}\ \langle \textit{term} \rangle\ )^{*} \\
\langle \textit{iteStmt} \rangle &::= \langle \textit{if} \rangle\ \texttt{(}\ \langle \textit{term} \rangle\ \texttt{)}\ \langle \textit{stmt} \rangle^{+}\ (\ \texttt{else}\ \langle \textit{stmt} \rangle^{+}\ )^{?} \\
\langle \textit{returnStmt} \rangle &::= \texttt{return}\ \langle \textit{term} \rangle \\
\langle \textit{varDecl} \rangle &::= \langle \textit{identifier} \rangle\ \texttt{:}\ \langle \textit{type} \rangle\ (\ \texttt{init}\ \langle \textit{term} \rangle\ )^{?}
\end{aligned}
$$

Common functions are composed of *function interfaces* and *function bodies.* Function interfaces describe the input variables and return type of functions. Function bodies, including local variables and statements, specify the behavior of the functions. All user-defined functions are common functions.

Native functions, on the other hand, have no function bodies but only function interfaces. Similar to the function declarations in other programming languages like C headers, native functions are part of the *Mediator* plugins where the behavior of the functions cannot be described through *Mediator* statements directly. For example, hardware-relevant operations, complex loops, etc. Such behavior can be captured by native codes in plugins. More discussions and examples of native functions will be presented in Section 3.

Entities. Both automata and systems are called *entities* in *Mediator*. All *Mediator* entities have their own templates and interfaces. However, ways to formalize their behavior are complete different.

Automata. Syntax tree of automata is shown as follows.

$$\begin{array}{rl}
\langle \textit{automaton} \rangle & ::= \texttt{automaton}\ \langle \textit{template} \rangle^{?}\ \langle \textit{identifier} \rangle\ \texttt{(}\ \langle \textit{port} \rangle^{*}\ \texttt{)}\ \{ \\
& (\ \texttt{variables}\ \{\ \langle \textit{varDecl} \rangle^{*}\ \}\ )^{?} \\
& \texttt{transitions}\ \{\ \langle \textit{transition} \rangle^{*}\ \}\ \} \\
\langle \textit{port} \rangle & ::= \langle \textit{identifier} \rangle\ \texttt{:}\ (\ \texttt{in} \mid \texttt{out}\ )\ \langle \textit{type} \rangle \\
\langle \textit{transition} \rangle & ::= \langle \textit{guardedStmt} \rangle \mid \texttt{group}\ \{\ \langle \textit{guardedStmt} \rangle^{*}\ \} \\
\langle \textit{guardedStmt} \rangle & ::= \langle \textit{term} \rangle\ \texttt{->}\ (\ \langle \textit{stmt} \rangle \mid \{\ \langle \textit{stmt} \rangle^{*}\ \}\ ) \\
\langle \textit{stmt} \rangle & ::= \langle \textit{assignStmt} \rangle \mid \langle \textit{iteStmt} \rangle \mid \texttt{sync}\ \langle \textit{identifier} \rangle^{+}
\end{array}$$

As the basic behavior unit in *Mediator*, *automaton* consists of four parts: *templates*, *interfaces*, *local variables* and *transitions*, which are interpreted respectively as follows.

1. *Templates.* Templates of an automaton include a set of parameter declarations. A parameter can be either a type (common type or parameter type) or a value. Concrete values or types are supposed to be provided when the automaton is instantiated (i.e. declared in systems).
2. *Interfaces.* Interfaces consist of directed ports and describe how automata interact with their contexts. Ports can be regarded as structures with three fields: `value`, `reqRead` and `reqWrite`, which correspondingly denote the values of parts, status of reading requests and status of writing requests.
3. *Local Variables.* Each automaton contains a set of local variables. Types of these variables are supposed to be *initialized.* We use the evaluations of local variables to represent states of an automaton.
4. *Transitions.* Behavior of an automaton is depicted by guarded transitions. Each transition consists of a boolean term *guard* and a sequence of statements. Transitions are ordered by their priority. For example, if multiple transitions are activated at the same time, the one that has highest priority will be fired. On the other hand, non-deterministic firing is also supported by encapsulating part of the transitions through `group`.

Currently, *Mediator* supports three types of statements:

1. *Assignment* statements, each including an expression and an optional assignment target, evaluate the expression and assign the result to its target if possible.
2. *Ite* (if-then-else) statements act as conditional choice statements in other languages.
3. *Synchronizing* statements, labelled with `sync`, are the flags requiring synchronized communication with other entities.

According to the existence of synchronizing statement (i.e. external communication through ports), transitions are classified as either *internal* transitions or *external* ones.

Compared with automata models being widely-used in other formal tools (e.g. UPPAAL[4], Simulink/Stateflow[13]), an automata in *Mediator* has no explicitly declared locations. Instead, it uses the evaluation of local variables to represent its states. An example of *Mediator* automaton can be found in Example **??**, Section 4, where automata are used to model drivers of motors.

$$\begin{array}{rl}
\langle \mathit{system} \rangle & ::= \texttt{system}\ \langle \mathit{template} \rangle^{?}\ \langle \mathit{identifier} \rangle\ \texttt{(}\ \langle \mathit{port} \rangle^{*}\ \texttt{)}\ \{ \\
 & (\ \texttt{internals}\ \langle \mathit{identifier} \rangle^{+})^{?} \\
 & (\ \texttt{components}\ \{\ \langle \mathit{componentDecl} \rangle^{*}\ \})^{?} \\
 & \texttt{connections}\ \{\ \langle \mathit{connectionDecl} \rangle^{*}\ \}\ \} \\
\langle \mathit{componentDecl} \rangle & ::= \langle \mathit{identifier} \rangle^{+}\ \texttt{:}\ \langle \mathit{systemType} \rangle \\
\langle \mathit{connectionDecl} \rangle & ::= \langle \mathit{systemType} \rangle\ \langle \mathit{params} \rangle\ \texttt{(}\ \langle \mathit{portName} \rangle^{+}\ \texttt{)}
\end{array}$$

Systems. As the textual representation of hierarchical entities to organize sub-entities (automata and simpler systems), *systems* with the above syntax tree are composed of:

1. *Components.* Entities can be placed and instantiated in systems as components. Each component is considered as a unique instance and executed in parallel with other components and connections. Ports of a component can be referenced by `identifier.portName` once the component is declared.
2. *Connections.* Connections are used to connect *a) the ports of the system itself, b) the ports of its components, and c) the internal nodes.* Inspired by the Reo project[5–7], complex connection behavior can also be determined by other entities.
3. *Internals.* Sometimes we need to combine multiple connections to perform more complex coordination behavior. Internal nodes, declared in `internals` segments, are untyped identifiers which are capable to weld two ports with consistent data-flow direction.

Systems also have *templates* and *interfaces* which have exactly the same forms as in automata. An example of a *Mediator* system is presented later in Section 4.

2.2 Arduino

Arduino[16] is an open-source electronics project that aims to build easy-to-use hardware and software. Arduino boards support various models of single-board micro-controllers, properly encapsulate them and expose a set of simple APIs to users. Here we give a brief introduction on program structure of Arduino C and hardware resources on a typical Arduino motherboard.

Program Structure. The Arduino community has developed a simple IDE that uses a dialect of C as its programming language. A typical Arduino C program describes its behavior through a `setup` function and a `loop` function.

– `setup()`: This function is called once when a sketch starts after power-up or reset. It is used to initialize variables, input and output pin modes, and other libraries needed in the sketch.
– `loop()`: After `setup` has been called, function `loop` is executed repeatedly in the main program. It controls the board until the board is powered off or is reset.

Hardware Resources. Pins are the most import hardware resources of Arduino boards. Through them the motherboard communicates with its accessories, e.g. sensors, motors and other devices. Numbers and types of pins vary a lot between different Arduino motherboards. Here we take Arduino Uno, one of the most popular Arduino motherboards, as an example to introduce types of pins. This motherboard is also used for the case study in Section 4.

1. *Analog Pins*: 6 analog pins named `A0 .. A5` are provided on Arduino Uno to perform analog signal transmission. Resolution of analog pins is 10 bits, in other words, value of an analog pin varies from 0 to 1023. Reading and writing operations on analogs pins are performed through builtin functions `analogRead(pin)` and `analogWrite(pin,value)`.
2. *Digital Pins*: Digital pins have only two possible values 0 and 1, or `LOW` and `HIGH`. Builtin functions `digialRead` and `digitalWrite` are used to read values from and write values to digital pins. Moreover, part of the digital pins provide Pulse-Width Modulation (PWM, [2]) feature to transfer analog value through binary encoding. In this case we are supposed to use `analogRead` and `analogWrite` instead.

An Arduino pin can be in either *INPUT* mode or *OUTPUT* mode. Modes of pins, no matter whether they are analog or digital, are configured through the builtin function `pinMode(pin, mode)`.

3 Code Generation

In this section we introduce the Arduino code generator for *Mediator*. The code generator mainly consists of a native function library and a set of generators for different *Mediator* language elements, e.g. types, functions and entities.

The Arduino code generator is implemented as a plugin of the *Mediator* project [1], which is written in Java and based on Maven framework. Executable Jar packages and help documents can be found in this repository.

3.1 Native Functions

Native functions are the bridges between software controllers and hardware resources in the *Mediator* framework. It tells *Mediator* which and how the hardware resources can be operated. Similar to C/C++ header files, these native functions are declared in a *Mediator* source file as a library. But their corresponding hardware behavior is defined by the code generator plugin.

The Arduino code generator supports the following native functions:

```
native function digitalRead (pin: int) : int 0..1;
native function digitalWrite (pin: int, val:int 0..1);
native function analogWrite (pin: int) : int 0..1023;
native function analogWrite (pin: int, val:int 0..1023);
native function delay (milliseconds: int);
```

- *DigitalRead* reads binary signals from a digital pin.
- *DigitalWrite* writes binary signals to either a digital or an analog pin. When it is performed on an analog pin, it configures the analog electronic level to either 1023 (`HIGH`) or 0 (`LOW`).
- *AnalogRead* reads analog signals from an analog pin.
- *AnalogWrite* writes analog signals to either an analog pin or a digital pin that supports PWM encoding.
- *Delay* forces the Arduino processor to suspend for a certain time delay.

3.2 Type Generator

Arduino C naturally supports most of the primitive types in Table 1 and part of the composite types in Table 2: unbounded integers `int`, float point numbers `double`, characters `char`, boolean values `bool` (as zero and non-zero integers), enumeration `enum`, union `union`, structure `struct` and finite arrays.

For the other types that are not directly supported by Arduino C, we use an attached runtime library to simulate their behavior. Such types include:

- *Bounded Integer*. In *Mediator*, bounded integers are mainly used to avoid overflow and unexpected values. For example, an Arduino analog signal varies between 0 and 1023, writing any other integers to an analog pin may lead to unknown behavior. Bounded integers are not supported by C. Moreover, according to its widely use we may suffer from performance degradation if we use complex structures to represent them. So we choose to generate assertions every time when a variable of bounded integer type is assigned. For example,

```
int a = 0; // type of a is int 0 .. 1 init 0
void loop() {
    // ...
    a = 1 - a;
    assert (a >= 0 && a <= 1);
    // ...
}
```

– *List.* C supports unbounded lists by pointers. However, C does not care about the capacity and consumption of them, which frequently lead to memory overflow and invalid dereference. In the *Mediator* runtime library, we encapsulate a void pointer to represent an unbounded list, and two integer fields to denote its capacity and the number of items existing in this list.

```
struct __MR_List {
    void * list;
    int capacity;
    int num_items;
    int item_size;
};
typedef struct __MR_List MR_List;

void init_empty_list (MR_List list, int item_size);
void list_add (MR_List list, void * item);
void * list_get (MR_List list, int index);
void list_del (MR_List list, int index);
```

The definition of unbounded list, as shown here, is type-independent. In other words, when storing items to or obtaining items from an unbounded list, type casting is unavoidable. In this case the *Mediator* syntax checker is responsible to guarantee the type consistency.

– *Map.* Similar to the *list*, *Mediator map* in Arduino C is also type-independent. A map uses two unbounded lists to store keys and values, respectively.

```
struct __MR_Map {
    MR_List keys;
    MR_List values;
};
typedef struct __MR_Map MR_Map;

void init_empty_map (MR_Map map, int key_size, int value_size);
void map_put (MR_Map map, void * key, void * value);
void * map_get (MR_Map map, void * key);
void map_del (MR_Map map, void * key);
```

– *Initialized.* In an Arduino program, default value of a type is only used in the `setup` function to initialize the corresponding variable. As a result, we do not have to use initialized type in Arduino C explicitly. For example, `int init 0` will be simply replaced by `int` when the Arduino C code is generated.

3.3 Function Generator

As mentioned before, there are two types of functions to be considered here: common functions and native functions.

Common Functions. When designing *Mediator*, we deliberately restrict the expressiveness of common functions (transitions as well) so that they are easier to be verified formally. As a result, common functions in *Mediator* are very easy to be encoded in C.

Native Functions. When a native function is called in a transition, the code generator needs to replace the function with corresponding native API in Arduino C. According to Section 2.2, all native functions supported in this plugin can be mapped to an Arduino API with the same name.

3.4 Entity Generator

All Arduino motherboards are equipped with only one processor and there is no time-sharing operating system support. They do not support parallel execution. Consequently, typical *Mediator* systems including a set of parallel components and connections can not be directly encoded in Arduino C.

Fortunately, a scheduling algorithm that flats a hierarchical system into a single automaton has been introduced in [15]. The algorithm guarantees that its resulting automaton is always canonical, i.e., the automaton contains exactly one transition group, in which all transitions are also canonical. With help of this algorithm, all we need to do is to encode a single *Mediator* automata in Arduino C.

The following steps illustrate the sketch of the encoding process.

1. *Template and Interface.* When generating Arduino codes, we always assume that the source automaton has NO template parameters and NO ports. Ports are special elements in *Mediator* that are used to react with a *Mediator* context. Behavior of ports are undeclared in the Arduino C context.
2. *Local variables* are declared as global variables in Arduino C. According to [15], types of all local variables should be initialized, i.e., they are declared with default values. And these default values will be assigned to the global variables in the `setup` function.
3. *Statements.* Transitions are composed of sequences of statements. After being scheduled and canonicalized, an automaton contains only *assignment statements* and *ite statements* (which are inherently supported in C). Since we assume that no port exists in the automaton's interface, *synchronizing statements* can be simply omitted.
4. *Transitions.* Transitions are *activated* if their guards are satisfied by the current evaluation of the local variables. Since in a canonical automaton all transitions are encapsulated by a `group`, the transition selection process is fully non-deterministic, i.e. the transition to fire is randomly selected from all *activated* transitions. In our approach, we use the following three steps to perform transition selection and firing:

– Step 1. *Activation checking.* At the beginning of each `loop`, we use a set of `if` statements to check which transitions are activated under the current evaluation of local variables. We use an array `cmd_activated` to store indexes of all transitions being activated.
– Step 2. *Random selection.* With the help of the `random` function in Arduino, it is easy to pick up a random index number from `cmd_activated`.
– Step 3. *Transition firing.* Another set of `if` blocks are used to encode the statements in transitions. Conditions of these blocks are used to check whether index of this transition is equal to the selected index.

Example 1. Consider a *Mediator* automaton `test` with one local variable x (initialized by 0) and two transitions: increasing x by 1 if x is less than zero, or decreasing x by 1 otherwise. The generated Arduino C code is as follows.

```
int test_x;
int cmd; // stores the index of selected transition
int cmd_activated[1]; // the capacity depends on number of
    transitions that belongs to the automaton

void setup() { test_x = 0; }

void loop() {
    // STEP 1 collect activated transitions
    cmd_activated_counter = 0; // the stack pointer of cmd_activated
    if (test_x < 0) {
        cmd_activated[cmd_activated_counter] = 0;
        cmd_activated_counter ++;
    }
    if (test_x >= 0) {
        cmd_activated[cmd_activated_counter] = 1;
        cmd_activated_counter ++;
    }

    // STEP 2 pick up a transition randomly
    cmd = cmd_activated[random(cmd_activated_counter)];

    // STEP 3 fire the selected transition
    if (cmd == 0) test_x = test_x + 1;
    if (cmd == 1) test_x = test_x - 1;
}
```

The code generating process is summarized in Algorithm 1.

4 Experiment

In this section, we show how to model a wheeled-robot controller in *Mediator* and generate Arduino C code through our code generator. The generated platform-dependent code has been directly compiled and flashed to the motherboard, without any manual modification.

Algorithm 1 Generate Codes for a Specified Entity E in a Program P

```
Require: A program $P = \langle Typedefs, Functions, Automata, Systems \rangle$, an entity $E$
Ensure: Arduino C codes
 1: $global, setup, loop \leftarrow$ “ ”
 2: if $E \in Automata$ then
 3:     $A \leftarrow$ Canonicalize($E$)
 4: else
 5:     $A \leftarrow$ Schedule($E$)
 6: end if
 7: if $A.Ports \neq \varnothing$ then
 8:     return NULL
 9: end if
10: for $var \in$ {local variables of $A$} do
11:     add variable declaration of $var$ to $global$ with the generated type
12:     add variable initialization of $var$ to $setup$
13: end for
14: for $t = guard \rightarrow statements \in$ {transitions of $A$} do
15:     add activation checking of $guard$ to $loop$
16: end for
17: add $random\ index\ selection$ to $loop$
18: for $t = guard \rightarrow statements \in$ {transitions of $A$} do
19:     add the generated $statements$ to $loop$
20:     if pin $pin$ is involved in $statements$ then
21:         add pinMode to $setup$ to configure $pin$ correctly
22:     end if
23: end for
24: $setup \leftarrow$ “void setup(){” + $setup$ + “}”
25: $loop \leftarrow$ “void loop() {” + $loop$ + “}”
26: return $global + setup + loop$
```

The hardware platform being used is based on an Arduino Uno motherboard, which consists of 4 motors (divided into two group *left and* right) and an ultrasonic distance sensor, as shown in Fig. 1.

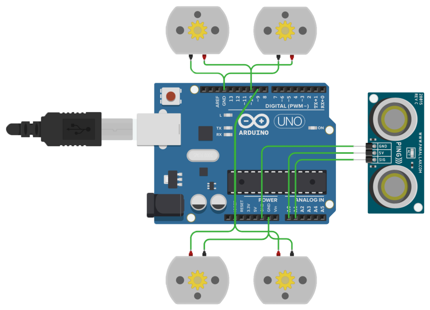

Fig. 1. Hardware Architecture of the Experiment Platform

The *Mediator* model of the wheeled-robot controller as shown in Figure 2 contains the following parts:

- *UltraSonic*. This sensor detects the distance from nearest obstacles and sends the distance information to the controller.
- *Controller*. The core algorithm of this robot is encapsulated in the controller. It reads distance information from the ultrasonic sensor, and gives an abstract command (e.g. forward, backward, turn, stop) to the *speeder* according to the distance information.
- *Speeder*. This automaton updates the speed of two motor groups according to the abstract command it receives from the *controller*.
- *Motors*. 4 motors, divided into two groups, are equipped in this small robot. Each motor has two control pins, one for direction and the other for speed. The *Mediator* automaton shown in Example **??** is the driver for motors. It receives a single control signal *speed* and updates the electronic level of control pins correspondingly.

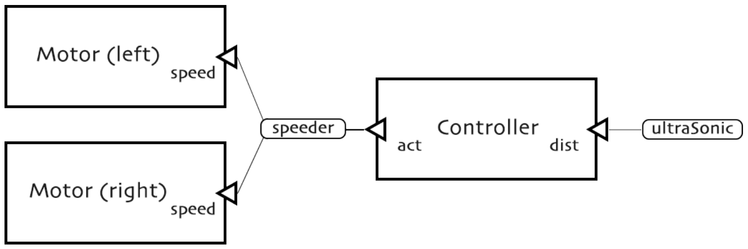

Fig. 2. Mediator Model of the Robot Controller

The controller shown in Figure 2 is captured by the following *Mediator* code, where the motors and the controller are defined as components, and the speeder and ultrasonic distance sensor are defined as connections.

```
system robot () {
    components {
        left_motor : motor<8, 9>;
        right_motor : motor<11, 10>;
        c  : controller;
    }
    connections {
        speeder(c.act, left_motor.speed, right_motor.speed);
        ultraSonicDist<6,7>(c.dist);
    }
}
```

Four *Mediator* automata are specified in this controller model: *motor*, *controller*, *speeder* and *ultraSonicDist*. Here we only show the definition of *motor*, further details can be found at [1].

A typical driver of motors with two control signals is defined as an automaton in *Mediator*. The simple automaton contains no local variable, one internal transition and one external transition. The internal transition updates the status of port `speed`, which is supposed to be ready to accept control commands at anytime. And the external transition receives target speed from the `speed` port and gives orders to the hardware correspondingly. The two template parameters describe where Arduino pins the motor is connected.

```
automaton <pinDirection,pinSpeed:int> motor (speed:in signedPWM) {
    variables {}
    transitions {
        !speed.reqRead -> speed.reqRead = true; // internal
        speed.reqRead && speed.reqWrite -> {
            sync speed; // external communication flag
            if (speed.value > 0) {
                digitalWrite(pinDirection, 1);
                analogWrite(pinSpeed, speed.value);
            } else {
                digitalWrite(pinDirection, 0);
                analogWrite(pinSpeed, -speed.value);
            }
        }
    }
}
```

Due to the length limitation, the generated program is omitted here and can be found at https://github.com/mediator-team/codegen-proposal/experiment.

5 Conclusion and Future Work

In this paper, we developed a fully-automatic code generator that converts *Mediator* models to executable Arduino C programs. Compared with plain Arduino C code, component-based *Mediator* models are more intuitive, easier to understand and construct. With the help of this code generator, engineers are able to build and review their models in *Mediator*, and generate Arduino C code automatically to avoid errors caused by manual encoding.

In the future we plan to use program verification tools to guarantee the reliability of generate codes. Due to the complexity, it is hard to formally prove the correctness of the code generator itself. So we plan to generate assertions automatically and insert them into the target code for formal verification. Hopefully they can be verified through program verification tools, e.g. CBMC[10]. Providing support for more hardware platforms and languages, e.g. Verilog/VHDL, System C, etc., are in our scope as well.

Acknowledgement

The work was partially supported by the National Natural Science Foundation of China under grant no. 61772038, 61532019, 61202069 and 61272160.

References

1. Mediator github repo, https://github.com/mediator-team
2. Wikipedia page of pulse-width modulation, https://en.wikipedia.org/wiki/Pulse-width_modulation
3. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer-Verlag (2004)
4. Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Pettersson, P., Weise, C., Yi, W.: UPPAAL - Now, Next, and Future. In: Cassez, F., Jard, C., Brigitte, R., Ryan, M.D. (eds.) Proceedings of MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer (2001)
5. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
6. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and temporal logical specifications for timed component connectors. Software and Systems Modeling 6(1), 59–82 (2007)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61(2), 75–113 (2006)
8. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
9. Budinsky, F., Finnie, M., Vlissides, J., Yu, P.: Automatic code generation from design patterns. IBM systems Journal 35(2), 151–171 (1996)
10. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proceedings of TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer (2004)
11. Cmkovic, I.: Component-based Software Engineering for Embedded Systems. In: Roman, G.C., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of ICSE 2005. pp. 712–713. ACM (2005)
12. Fisher, K.: EventB2Java: A Code Generator for Event-B. In: Proceedings of NFM 2016. LNCS, vol. 9690, pp. 166–171. Springer (2016)
13. Hahn, B., Valentine, D.T.: SIMULINK Toolbox. In: Essential MATLAB for engineers and scientists, pp. 341–356. Academic Press (2016)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.m., Irwin, J., Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.m., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) Proceedings of ECOOP'97. LNCS, vol. 1241, pp. 220–242. Springer (1997)
15. Li, Y., Sun, M.: Component-based modeling in Mediator. In: Proceedings of FACS 2017. LNCS, vol. 10487, pp. 1–19. Springer (2017)
16. Margolis, M.: Arduino cookbook. O'Reilly Media, Inc., Sebastopol, USA (2011)
17. National Instruments: Labview, http://www.ni.com/zh-cn/shop/labview.html
18. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
19. Sebesta, R.W.: Concepts of Programming Languages. Pearson, 10th edn. (2012)
20. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming, Second Edition. Publishing House of Electronics Industry (2003)