A Case Study on Using Automata in Control Synthesis

Thomas Hune and Anders Sandholm

BRICS*, Department of Computer Science
University of Aarhus, Denmark
{baris,sandholm}@brics.dk

Abstract. We study a method for synthesizing control programs. The method merges an existing control program with a control automaton. We have used monadic second order logic over strings to specify the control automata. Specifications are translated into automata by the Mona tool. This yields a new control program restricting the behavior of the old control program such that the specifications are satisfied. The method is presented through a concrete example.

1 Introduction

In the following we will describe some practical experience on synthesizing programs for the LEGO® RCX™ system. The synthesis is based partly on an existing simple program and partly on an automaton generated by the tool Mona [7].

Writing control programs can often be an error prone task, especially if a number of special cases must be taken into account. Time is often spent on handling special case or failure situations rather than solving the actual problem at hand. A number of different methods and tools have been developed to ease this task. One well known method is based on a control automaton running in parallel with the actual program [12, 13]. The automaton controls the input and output events of the program. This is a way of restricting the sequences of I/O actions occurring.

The automata controlling the I/O actions can be specified in different ways, e.g. by specifying it directly in some suitable notation, or by a logical formula. We have chosen the latter approach. There are various logics which could be used as specification language. We have chosen to use monadic second order logic over strings (M2L) [2] for a number of reasons. First of all M2L has a number of nice properties such as being expressive and succinct. For instance, having second order quantification M2L is more expressive than LTL. Furthermore, there are succinct M2L-formulae of size n which have minimal corresponding automata of non-elementary size. Secondly, the tool Mona [7] implements a translation from M2L formulae to minimal deterministic automata (MDFA) accepting the language specified by the formula. The automata generated do not contain any

* Basic Research in Computer Science,
Center of the Danish National Research Foundation.

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 349–362, 2000.

acceptance condition for infinite executions so we will only be considering safety properties.

The method we study here is a variation of classical synthesis as described in e.g. [10, 12], in that the method is partly based on an existing control program. The aim of the synthesis described here is to restrict the behavior of an existing (hopefully very simple) control program such that it satisfies certain given properties. The executions of the existing control program are restricted by the control automaton having I/O events as alphabet. These events define the interface between the existing control program and the specification.

For studying the method we will look at a control program for a moving crane. We have implemented the method for this example in the LEGO® RCX™ system [9]. Using the LEGO® RCX™ system is interesting for at least two reasons. First of all the environment of the RCX™ system and especially the programming language is quite restricted, so it is not obvious that implementing the method is feasible at all. Secondly, using the LEGO® RCX™ system one can build actual physical systems for testing the control programs. We have built the crane and used it with different control programs.

The language running on the LEGO® RCX™ brick (RCX™ language) is an assembly-like language with a few high level features, like a notion of task or process. Programs are written on a PC and downloaded to the RCX™ brick where they are interpreted.

1.1 Related Work

The use of finite state automata for controlling systems is not novel. Ramadge and Wonham [12] give a survey of classic results.

The method used in this paper has been used successfully in `<bigwig>` [13], a tool for specifying and generating interactive Web services. Our method for control synthesis is used as an integral part of `<bigwig>` to define safety constraints. In fact, via use of a powerful macro mechanism [1] the method has been used to extend the Web programming language in `<bigwig>` with concepts and primitives for concurrency control, such as, semaphores and monitors.

1.2 Outline of the Paper

In the following section we will outline the method. A short presentation of the LEGO® system is given in Section 3. In Section 4 the crane example is presented. The logic-based specification language is presented in Section 5, and the merge of automata with the RCX™ code in Section 6. Finally, Section 7 rounds off with conclusions, and suggestions for future work.

2 Outline of the Method

The two main components of the synthesis is a basic control program and an automaton. From these two components we generate a control program which is ready for use. We do not have any special requirements to what a control program is, like no side effects, since in our case the control program is the only program running on the RCX™ brick. The interface between the two components is a predefined set of I/O actions. This will typically be all commands in the program for reading sensors or manipulating actuators.

Given a basic control program and an implementation of the automaton we merge these. Each instruction in the basic control program using one of the I/O actions is transformed to a sequence of instructions first calling the automaton and based on the response from the automaton performing the action or not. Section 6.3 will discuss different approaches to what should happen, when a given action is not allowed by the automaton.

Since the automaton is invoked only when the basic control program is about to make an I/O action, it can only restrict the possible I/O behaviors of the control program, not add new I/O actions. Only looking at sequences of I/O actions the basic control program must therefore be able to generate all sequences present in the solution. Since the automaton will prune away unwanted sequences, the basic control program might also generate unwanted sequences. The basic control program should not implement any kind of priority scheme, unless one is sure that combining this with the safety specification will not lead to deadlocks.

The hope is that writing such basic control programs should be a simple task. In general the basic control program could be one always being able to read any sensor and give any command to the actuators. This amounts to generating the star operation of the input alphabet. However, there will often be a correspondence between input and output which it would be natural to have in a basic the control program. This is the case in the example shown later.

One could see the basic control program as implementing the method for controlling the sequences of I/O actions and the automaton defining the allowed policy for these. This suggests that with one implementation of a basic control program it is possible to test different specifications or strategies (policies) only by changing the control automaton. Therefore, a fast (automatic) way of getting an implementation of an automaton from a specification and merging this with the control program allows for testing different specifications fast.

3 The LEGO® System

The studies we have conducted are based on the LEGO® RCX™ system and the associated RCX™ language. The language is an assembly like language with some high level concepts like concurrent tasks. The language is restricted in a number of ways, e.g. it is possible to address only 32 integer variables and allows only ten tasks in a program. Furthermore, one cannot use symbolic names in programs. However, we have not encountered problems with the mentioned restrictions during our experiments.

A small operating system is running on the RCX™ with processes for handling I/O and one process running an interpreter for the RCX™ language. The RCX™

brick has three output ports (for motors and lights) and three input ports. Four kinds of sensors for the input ports are supplied by LEGO®: touch, temperature, rotation, and light.

3.1 The RCX™ Language

A program consists of a collection of at most ten tasks. There is no special way to communicate between tasks but all variables are shared, providing a way of communication. A task can start another task with the command `StartTask(i)` and stop it with the command `StopTask(i)`. Starting a task means restarting it from the beginning. That is, there is no command for resuming the execution of a task nor spawning an extra "instance" of a task.

The language has some commands for controlling the actuators, the main ones being `On(li)` and `Off(li)` where `li` is a list of ports. The commands `SetFwd(li)` and `SetRwd(li)` sets the direction of the ports in `li` to forward and reverse respectively. There are also a number of instructions for manipulating variables. All of these take three integer arguments. The first argument specifies the target variable, the second the type of the source, and the third the source. The most important types of sources are: variables (the third argument is then the number of the variable), constants (the third argument is then the value), and sensor readings (the third argument is then the number of the sensor). These types of sources can be used in the instruction `SetVar(i,j,k)` for assigning a value to a variable. In the instructions for calculating like `SumVar(i,j,k)`, `SubVar(i,j,k)`, and `MulVar(i,j,k)` sensor readings are not allowed.

Loops can be defined in two ways, either by the `Loop(j,k)` instruction or by the `While(j,k,l,m,n)` instruction. The arguments of the `Loop` indicates how many times the body should be iterated in the same way as the source of the instructions for calculating. The `While` loop is iterated as long as the condition specified by the arguments is satisfied. The first two and last two arguments specify the sources of a comparison as in an assignment and `l` specifies a relation from the set $\{=, <, >, \neq\}$.

There is also a conditional, `If(j,k,l,m,n)`, with the condition specified as in the `While` construct and an `Else` branch can be specified as well.

One can block a task for a given time using the `Wait(j,k)` statement. When the specified time has passed, execution of the task is resumed.

During execution a task is either enabled or blocked. A task can be blocked by a `StopTask(i)` instruction, by a `Wait(j,k)` instruction, or by finishing its execution (reaching the end of the code). Initially only task zero is enabled. The enabled tasks are executed in a round robin fashion, where each task executes one instruction and then leaves control for the next task.

The statements presented above constitute the part of the RCX™ language which we have used for implementing control automata.

4 Example

As an example we will look at a crane which we will program in the RCX™ language. We have built the crane and tested it with different control programs. The crane is run by three motors connected to the RCX™. One motor is driving the wheels, one is turning the turret around, and one is moving the hook up and down. The input for the three motors are three touch sensors, which is all the RCX™ brick has room for. This means we can only turn motors on and off. Therefore the crane alternates between moving forward and backward each time the motor is turned on. The direction of turret and the hook is controlled in a similar way.

A very basic control program for the crane could consist of four tasks. One task for setting up the sensors and motors, and starting the other tasks. For each of the three inputs, one task for monitoring input and controlling the motor correspondingly. Task 1 for monitoring sensor 0 would then be

```
BeginOfTask 1
Loop 2, 0                   'An infinite loop
  SetFwd "0"                'Set direction of motor 0 to forward
  SetVar 1, SENSOR, 0       'Var1 := Sensor0
  While VAR, 1, 3, CONST, 1 'While Var1 != 1
    SetVar 1, SENSOR, 0
  EndWhile
  On "0"                    'Start motor 0
  Wait CONST, 100           'Wait
  SetVar 1, SENSOR, 0       'Var1 := Sensor0
  While VAR, 1, 3, CONST, 1 'While Var1 != 1
    SetVar 1, SENSOR, 0
  EndWhile
  Off "0"                   'Stop motor 0
  Wait CONST, 100               'Wait
  ... repeat the code replacing SetFwd "0" with SetRwd "0" ...
EndLoop
EndOfTask
```

The `Wait` statements ensures that one touch of the sensor is not read as two touches. We could of course have tested for this but for our example this will do. The two other tasks for controlling the remaining two motors look similar, only the numbers of variables, sensors and motors are different.

For the purpose of illustrating the presented method we choose to place the following constraints on the behavior of the crane. First of all we only want one thing happening at a time, so we will not allow for two motors to be turned on at the same time. Pressing the touch sensor could now be seen as a request which the control program may grant (and start the motor) when all the motors are stopped. Moreover, we want that moving the hook has higher priority than the wheels and the turret. Requests from the other two are handled in order of arrival. The first constraint on the behavior is basically mutual exclusion which is nontrivial to implement in the RCX™ language (this is an integrated part of the implementation of the automata-based approach described in Section 6). On top of this we have a mixed priority and queue scheme.

5 Logic-Based Specifications

Basically we could keep the initial simple control program if we had a way of pruning out some unwanted executions. To be able to implement the constraints we have to change the initial control program slightly. This is done by considering touching a sensor as a request. The motor can be turned on when the request is accepted. Even with these changes the program is still a simple to write. Execution of the program gives rise to a sequence of events. In our case we will consider input (requests), and two kinds of output (start and stop motor) as events. We then implement the automaton accepting the language over these events satisfying the introduced constraints. With this approach we can thus keep the control program simple.

Traditionally, control languages are described by automata which are in some cases a good formalism to work with. However, having experience in using logic for specifying properties, we will take that approach here. In this section we describe the use of a logic formalism from which we can automatically generate automata.

5.1 Terminology

An *automaton* is a structure $A = (Q, q^{\text{in}}, \Sigma, \rightarrow, F)$, where Q is a set of states with initial state $q^{\text{in}} \in Q$, Σ is a finite set of events, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ the set of acceptance states. We shall use $q_1 \xrightarrow{\sigma} q_2$ to denote $(q_1, \sigma, q_2) \in \rightarrow$. A sequence $w = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in \Sigma^*$ is said to be *accepted* by the automaton A if there exists a run of A which reads the sequence w and ends up in an accepting state q. So we have $q_1, \ldots, q_{n-1} \in Q$ and $q \in F$, such that $q^{\text{in}} \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-2}} q_{n-1} \xrightarrow{\sigma_{n-1}} q$. We shall denote by $L(A)$ the *language* recognized by an automaton, that is, $L(A) = \{\, w \in \Sigma^* \mid A \text{ accepts } w \,\}$.

In order to be able to define the notion of a legal control language, one partitions the event set Σ into *uncontrollable* and *controllable* events: $\Sigma = \Sigma_u \cup \Sigma_c$. The controllable events can be disabled by the control automaton at any time, whereas the uncontrollable ones are performed autonomously by the system without any possible interference by the control automaton, which merely has to accept the fact that the particular uncontrollable event has occurred. Thus a control language must in some sense, which is defined precisely below, respect the uncontrollableness of certain events. Furthermore, since our method only allows restrictions concerning safety properties, it does not make sense to have non-prefix-closed languages as control languages. That is, we define the notion of control language as follows.

Let $pre(L)$ denote the prefix closure of a language L, and let $unc(L)$ denote closure of L under concatenation of uncontrollable events. That is, let

$$pre(L) = \{ v \in \Sigma^* \mid \exists w \in \Sigma^* : vw \in L \} \text{ and}$$
$$unc(L) = \{ vw \in \Sigma^* \mid v \in L \wedge w \in \Sigma_u^* \}.$$

A language, L over $\Sigma = \Sigma_u \cup \Sigma_c$ is called a *control language* if it satisfies the two properties $pre(L) = L$ and $unc(L) = L$.

When using deterministic finite state automata to specify sets of sequences, checking for prefix closedness is easy. One just has to make sure that all transitions from non-accepting states go to non-accepting states. Similarly, checking closure under concatenation of uncontrollable events is straightforward for deterministic automata.

What is new here, in comparison to the use of our method in [13], apart from the new domain of LEGO® RCX™ robots, is the partition into controllable and uncontrollable events and the resulting additional restrictions and computations.

5.2 Specification Logic

It would be nice if instead of converting the informal requirement in Section 4 into an automaton, one could write it formally in a specification formalism closer to natural language. That is, we would like to be able to write something like the following.

– Only one motor can be turned on at a time;
– If the wheels get turned on, then the hook must not be requesting and the wheels must have been the first to make a request; and
– If the turret gets turned on, then the hook must not be requesting and the wheels must have been the first to make a request.

We therefore turn to a formalism that is as expressive as finite state automata and yet still allows for separation of the declaratively specified requirements (previously our control automaton) and the operational part of the control program (the existing RCX™ program).

Experience has shown that logic is a suitable specification formalism for control languages. For the purpose of defining controllers for LEGO® RCX™ robots, we have chosen to use M2L. One might argue in favor of other specification formalisms such as high-level Petri Nets [6] or Message Sequence Charts [11]. Being a logic formalism, however, M2L has the advantage that specifications can be developed iteratively, that is, one can easily add, delete, and modify parts of a specification. It also has a readable textual format. Moreover, the formalism in use should be simple enough that a runtime checker, such as an automaton, can actually be calculated and downloaded to the RCX™ brick. Thus, M2L is powerful and yet just simple enough to actually subject it to automated computation.

Experience in using M2L as a language for defining control requirements has shown that only the first-order fraction of the logic is used in practice [1, 13]. We shall thus consider only first order quantifications, though second-order quantifications could be added at no extra cost.

The abstract syntax of the logic is given by the following grammar:

$$\phi ::= \exists p : \phi' \mid \forall p : \phi' \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' \vee \phi'' \mid \phi' \Rightarrow \phi'' \mid \sigma(t) \mid t < t'$$
$$t ::= p \mid t + 1$$

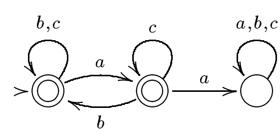
That is, M2L has the constructs: universal and existential quantifications over first order variables (ranging over positions in the sequence of events), standard boolean connectives such as negation, conjunction, disjunction, and implication, and basic formulae, $\sigma(t)$, to test whether an event σ can be found at position t, and $t < t'$, to test whether position t is before position t'. It also has operations on terms, such as, given a position t one can point out its successor $(t + 1)$, and simple term variables (p).

A formula ϕ in M2L over the event set Σ will – when interpreted over a finite sequence of events w – either evaluate to true or to false and we shall write this as $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is $L(\phi) = \{ w \in \Sigma^* \mid w \models \phi \}$. The language associated with an M2L formula is guaranteed to be regular. In fact, it has been known since the sixties that M2L characterizes regularity [2, 4].

The Mona tool implements the constructive proof of the fact that for each M2L formula there is a minimal deterministic finite state automaton accepting the language of the formula. That is, Mona translates a particular M2L formulae, ϕ, into its corresponding minimal deterministic finite state automata (MDFA), A, such that $L(\phi) = L(A)$.

Example 1. Let $\Sigma = \{a, b, c\}$. The M2L formula to left

$$\begin{array}{l}\forall p, p'' : (p < p'' \wedge a(p) \wedge a(p'')) \\ \quad \Longrightarrow \exists p' : p < p' < p'' \wedge b(p')\end{array}$$



is true for sequences in which any two occurrences of a will have an occurrence of b in between. Using Mona, one can compute the automaton corresponding to the formula above. The resulting automaton appears to the right.

Example 2. With this logic-based specification language in place, we can write a specification of the requirements given in the example. The logic-based specification looks quite complex at first. However, because of it's modular structure we find it easier to handle than the automaton. The basic formulae for the elements of the alphabet are `req1(t)`, `req2(t)`, `req3(t)`, `turnon1(t)`, `turnon2(t)`, `turnon3(t)`, `turnoff1(t)`, `turnoff2(t)`, and `turnoff1(t)`. The first three are uncontrollable events of the alphabet and the rest are controllable events of the alphabet. A predicate is true if the event at position t is the mentioned event. Using these basic predicates we can define some basic predicates like all motors are stopped by:

$$\begin{aligned}\text{off1}(t) = {} & (\forall t' : t' < t \Rightarrow \neg\texttt{turnon1}(t')) \vee \\ & (\forall t' : (t' < t \wedge \texttt{turnon1}(t')) \Rightarrow \\ & \exists t'' : t' < t'' \wedge t'' < t \wedge \texttt{turnoff1}(t''))\end{aligned}$$

Similarly, we define predicates `off2(t)` and `off3(t)` and using these we can define a predicate, `alloff(t)`, specifying that all the motors are turned off.

$$\text{alloff}(t) = \text{off1}(t) \wedge \text{off2}(t) \wedge \text{off3}(t)$$

We can specify that motor 1 has been requested to be turned but has not yet been turned on by the following predicate:

$$\text{request1}(t) = \exists t' : t' < t \wedge \texttt{req1}(t') \wedge (\forall t'' : (t' < t'' \wedge t'' < t) \Rightarrow \neg\texttt{turnon1}(t''))$$

Predicates `request2(t)` and `request3(t)` are specified similarly. Using this we can define a predicate specifying that the first request which has not been acknowledged is one.

$$\begin{aligned}\text{req1first}(t) = {} & \text{req1}(t) \wedge \forall t' : (t' < t \wedge \texttt{request1}(t') \wedge \\ & \forall t'' : (t' < t'' \wedge t'' < t) \Rightarrow \neg\texttt{request1}(t'')) \Rightarrow \\ & ((\text{req2}(t') \Rightarrow \exists t'' : t' < t'' \wedge t'' < t \wedge \neg\text{req2}(t'')) \wedge \\ & (\text{req3}(t') \Rightarrow \exists t'' : t' < t'' \wedge t'' < t \wedge \neg\text{req3}(t'')))\end{aligned}$$

Again, predicates `req2first(t)` and `req3first(t)` are specified similarly. With these basic predicates as building blocks we can give a specification closely related to the informal requirements of the example.

$$\begin{aligned}& \forall t : (\texttt{turnon1}(t) \vee \texttt{turnon2}(t) \vee \texttt{turnon3}(t)) \Rightarrow \text{alloff}(t) \wedge \\ & \forall t : \texttt{turnon2}(t) \Rightarrow (\neg\text{req1}(t) \wedge \text{req2first}(t)) \wedge \\ & \forall t : \texttt{turnon3}(t) \Rightarrow (\neg\text{req1}(t) \wedge \text{req3first}(t)),\end{aligned}$$

An informal specification for the control of the crane containing three properties was given in Section 5.2. Each of these properties corresponds to one of the lines in the predicate above. For instance the first line of the predicate specifies that if a motor is turned on then all the motors are turned off. This corresponds to the first property that only one motor can be turned on at a time. Similar for the remaining two lines and properties.

Since our basic control program specifies the order of the events in the individual tasks (first `req`, then `turnon`, and then `turnoff`), this specification will define the wanted behavior.

From this specification Mona generates the minimal deterministic automaton which has 46 states and 276 transitions (six from each state).

Should we want to change the control language of our example in such a way that all three tasks have equal priority, the overall structure of the control automaton would change. As the following example will show, modifying the logical formula is indeed quite comprehensible in the case of the LEGO® crane requirements.

Example 3. Say that we would like to change the requirements such that all motors are given equal priority, that is, they will be turned on in a first come first served manner. Using the logic-based specification, all we have to do is to change the last two lines of our specification slightly resulting in the following fifo requirement.

$$\begin{aligned}
&\forall t : (\texttt{turnon1}(t) \vee \texttt{turnon2}(t) \vee \texttt{turnon3}(t)) \Rightarrow \text{alloff}(t) \wedge \\
&\forall t : \texttt{turnon1}(t) \Rightarrow \text{req1first}(t) \wedge \\
&\forall t : \texttt{turnon2}(t) \Rightarrow \text{req2first}(t) \wedge \\
&\forall t : \texttt{turnon3}(t) \Rightarrow \text{req3first}(t).
\end{aligned}$$

Note that the sub-formulae, such as, `alloff()` and `req2first()` are reused from the previous specification. As we can see it is relatively easy to change the specification using the previously defined primitives.

From this specification Mona generates an automaton with 79 states and 474 transitions.

6 Merging Automata and RCX™ Code

Given a control automaton and the basic control program, one can synthesize the complete control program. In this section we describe how to translate an automaton into RCX™ code and how this is merged with the existing RCX™ control program. For our example we have done this by hand. It should be clear from this section that only standard techniques are used and these can easily be carried out automatically.

6.1 Wrapping the RCX™ Code

The execution of the basic control program is restricted to sequences allowed by a control automaton as follows. Firstly, RCX™ code is generated for the control automaton and then this code is merged with the existing RCX™ code. Merging RCX™ code with an automaton can in some sense be considered a program transformation. Each statement involving a request or writing to an output port is replaced by a block of code that tests whether the operation is legal according to the control automaton. For our example an action should be delayed if the control automaton does not allow it. Transforming the code for turning motor 0 on, will lead to the following piece of code.

```
While VAR, 4, 3, CONST, 1 'While automaton has not accepted the command
  SetVar 31, CONST, 1     'Arg := on0, the argument for the automaton
  GoSub 0                 'Run the automaton
  SetVar 4, VAR, 22       'Local success:= global success
EndWhile
On "0"                    'Execution of the actual command
```

We have chosen to implement the automaton as a subroutine. Since arguments for subroutines are not allowed in the language, passing an argument to the automaton has to be done via a global variable. Similarly, since a subroutine cannot return a value, return values are also placed in global variables for the process to read. The while loop delays the action until the automaton accepts execution of it.

6.2 Implementing Mutual Exclusion and Automata

However, the idea described above is not sufficient since we will not allow more tasks to use the automaton simultaneously. In the RCX™ language the problem is obvious since we are using shared variables for passing arguments and results. In general, we also need exclusive access to the automaton since the outcome of the automaton depends on its state when execution begins. If a process accesses the automaton while it is used by another process, the state variable might be corrupted. Therefore we must have exclusive access to the automaton.

In our implementation we have used Dijkstra's algorithm to implement mutual exclusion between several processes [3]. But any correct mutual exclusion algorithm would of course do. The algorithm uses some shared variables but this is no problem in the RCX™ language since all variables are shared. There are no gotos in the RCX™ language. Therefore, we have used an extra while loop and a success variable for each task. Except from these details, the algorithm is followed directly.

An automaton is implemented in the standard way by representing the transition relation as nested conditionals of depth two branching on the current state and the input symbol respectively. The current state and the input symbol is represented by one variable each. This gives us a way to combine the run of an automaton with the execution of standard RCX™ code with wrapped input/output statements.

6.3 Variations of the Method

In the example an action is delayed if the control automaton does not grant permission at once. Depending on the problem to be solved the action taken when permission is not granted can vary. That is, there are various ways of handling this temporary lack of controller acknowledgment:

- as in the above example where the task is *busy wait* asking the controller over and over whether its label had been enabled; but
- one could also simply *cancel* or ignore the statement requesting permission and continue execution. This could be done by replacing the busy waiting while loop by an if statement.

The former would often be the preferred approach in cases where the internal state of the code is important, such as, in our example, or in a train gate controller. The latter would be a good choice in cases where the code is written in a reactive style, constantly changing output action based on newly read input, e.g. in autonomous robots.

The property implemented by the automaton in the example was specific to the problem. One could also imagine using the method for general properties e.g. for protecting hardware against malicious sequences of actions. This leaves at least two options of where to put the automaton:

– as in the example above where the automaton was put *alongside the wrapped RCX™ code*. Placing the code implementing the automaton at this level seems a natural choice when dealing with properties about the behavior of a specific RCX™ program solving a particular problem.
– If the property is of a more general kind, one should rather place the automaton *at the level of the operating system*.

So far we have only considered untimed properties. One could easily imagine using automata with discrete time as control automata. This would open for a whole new range of properties to be specified, e.g. a minimum delay between two actions. In the example it would be possible to specify properties like that a minimum time of 5 seconds should pass between stopping the crane and starting to move the hook.

On the RCX™ this could be realized by having a variable representing the discrete time. This variable could be updated by a task consisting of an infinite loop waiting for one time unit and then updating the variable. Assuming variable number zero represents the time, it could be updated by:

```
SetVar 0, CONST, 0        'Initialize the timer
Loop CONST, 0             'An infinite loop
  Wait CONST, 10          'Wait for 1 sec.
  SumVar 0, CONST, 1      'Update the timer
EndLoop
```

7 Conclusion

We have used control automata in conjunction with basic control programs for synthesizing complete control programs. Using this method one can add to a basic control program a control automaton which will ensure certain safety properties are satisfied. We have used M2L to specify the control automata and the Mona tool to translate formulae into automata.

The approach has been implemented in the setting of the LEGO® RCX™ system. This has allowed for the possibility of testing the implementations on real physical systems.

Based on our experiments we find the method well suited for synthesis of programs ensuring safety properties like the ones we have used. We find the main advantage of the method is the ease of testing different specifications. The separation of the active control program and the restricting automaton also allows for ensuring (new) safety specifications to existing control programs. In critical systems one might consider the automaton only for monitoring actions, not restricting these, to avoid deadlocks.

The main disadvantage of the method is the restriction to safety properties. Since the all concurrent tasks must access the automaton there is a danger of this becoming bottleneck.

Future work There is an overhead connected with gaining exclusive access to the automaton and running it. How much time is spent on gaining access to the automaton of course depends on the arrival of input events. It would be interesting to calculate some specific times for this given some input sequences. A tool for translating RCX™ programs to timed models supported by Uppaal [8] exists [5]. Using Uppaal one can "measure" the time spent by a program from an input is read until the response arrives.

The example presented in this paper only has one component (one crane) and the control restrictions are consequently imposed on that particular component only. One could easily imagine having several components in a distributed environment working to achieve a common goal. By use of modular synthesis and distributed control [12] via independence analysis [13] one can statically infer information about which constraints to put locally on the components and which to put on the (most often necessary) central controller.

References

[1] Claus Brabrand. Synthesizing safety controllers for interactive Web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/`.

[2] J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[3] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[4] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[5] T. Hune. Modelling a real-time language. In *Proceedings of Fourth International Workshop on Formal Methods for Industrial Critical Systems*, 1999.

[6] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets – Theory and Application*. Springer-Verlag, 1991.

[7] N. Klarlund and A. Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998.

[8] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[9] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

[10] Z. Manna and A. Pnueli. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[11] S. Mauw and M. A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[12] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[13] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering, FASE'98*, Lecture Notes in Computer Science, LNCS 1382, pages 270–284. Springer-Verlag, March/April 1998. Also available as BRICS Technical Report RS-97-47.