Nordic Journal of Computing

CODE SYNTHESIS FOR TIMED AUTOMATA

TOBIAS AMNELL[1] ELENA FERSMAN[1] PAUL PETTERSSON[1]
HONGYAN SUN[2] WANG YI[1]

[1] *Uppsala University, Sweden.*
{tobiasa,elenaf,paupet,yi}@docs.uu.se.
[2] *Technical University of Denmark, Denmark.*
sun@imm.dtu.dk.

Abstract. We present a framework for the development of real-time embedded systems based on timed automata extended with a notion of real-time tasks. It has been shown previously that reachability and schedulability for such automata can be checked effectively using model checking techniques. In this paper, we propose to use the extended automata as design models. We describe how to compile design models to executable programs with predictable behaviours. The compiling procedure ensures that the execution of the generated code satisfies mixed timing, resource and logical constraints imposed on the design model. To demonstrate the applicability of the framework, a prototype C-code generator based on the legOS operating system has been implemented in the TIMES tool and applied to develop the control software for a production cell. The production cell has been built in LEGO® equipped with a Hitachi H8 based LEGO® Mindstorms control brick.

Key words: real-time, timed automata, code synthesis, schedulability, case study.

1. Introduction

A key facility of many commercial tools for development of embedded software is to automatically synthesise executable code (i.e. code generation) from design models. However, few of the existing tools are capable of producing software with *predictable* timing behaviours, i.e. code which is known a priori to satisfy given timing constraints. In fact, most of the commercial tools trust the software developers to resolve the timing issues e.g. whether all tasks will meet their deadlines, when the generated code is supposed to run on a specific target platform. In contrast, research tools such as UPPAAL [Larsen *et al.* 1997] and KRONOS [Bozga *et al.* 1998] are often dedicated to analysis and abstraction on high-level design descriptions. This has proven useful for finding errors and checking correctness properties in many case studies, e.g. [Bengtsson *et al.* 1996], [Lindahl *et al.* 1998], [Stauner *et al.* 1997] and [D'Argenio *et al.* 1997]. However, it provides little or no support for producing the actual program code to be executed in the final implementation.

In this work, we aim at synthesis of executable code with predictable behaviours, from high level design specifications. We combine results and ideas from model-checking, scheduling, and synchronous programming to develop a framework which

Received February 7, 2003.

supports, on one hand, formal specification, validation, analysis of design, and on the other hand, code generation which ensures that timing constraints are met when the generated code is executed on the target system. As a design language, we use timed automata extended with real-time tasks [Fersman *et al.* 2002]. In such extended automata, a transition is associated with a task (or several tasks) that is an executable program with given parameters, such as execution time, deadline, fixed priority etc. Intuitively, whenever an automaton takes a transition, its associated task is released for execution. Thus a discrete transition denotes an event releasing a task and the clock constraints on the transition (i.e. the guards) specify the possible arrival times of the associated task. When a task is released, it is inserted into the ready queue of the operating system and executed according to its scheduling policy.

It has been shown that the schedulability checking problem for the model of extended automata is decidable for both non-preemptive [Ericsson *et al.* 1999] and preemptive [Fersman *et al.* 2002] scheduling policies. An automaton is schedulable with a given scheduling policy if, for all possible sequences of events accepted by the automaton the released tasks can be computed within their deadlines. The check is performed by transforming the original scheduling problem to a reachability problem of timed automata extended with subtraction operations on clocks. This means that, given a design model of a timed system, described as a set of automata running in parallel, it can be checked whether it is schedulable prior to its implementation provided that the computing times on the target hardware for the associated tasks are known.

In this paper, we show how to compile a (verified) design model to an executable program preserving the mixed timing and logical constraints imposed on the model. Inspired by the design philosophy of synchronous languages e.g. Esterel [Berry and Gonthier 1992], we assume that the target real-time operating system and hardware ensure the *synchrony hypothesis*, i.e. the run-time of certain system functions is neglectable compared with the execution times and deadlines of the application tasks. Based on this assumption, we transform the control structure of an automaton to operating system functions and the associated tasks to light-weight threads. The compiling procedure should be applicable to a variety of target platforms that guarantee the assumption.

To demonstrate the applicability, we have developed a prototype C-code generator on the legOS operating system for the LEGO® Mindstorms control brick, equipped with an 8-bit Hitachi micro-processor. The code generator has been integrated in the Times tool [Amnell *et al.* 2002] and applied to develop the control software for a production cell built in LEGO® and controlled by LEGO® Mindstorms control bricks. The control program for the most central component, the two-armed robot, is designed, analysed and generated using the Times tool. The generated code has been compiled, downloaded, and executed on LEGO® Mindstorms bricks.

The rest of this article is organised as follows: Section 2 describes the syntax and semantics of design models that are timed automata extended with tasks. To extract executable behaviours from design models, we present a deterministic semantics for the extended automata. Section 3 shows how to implement the deterministic semantics on a small generic embedded operating system. Section 4 describes the modelling and design of a production cell using the extended automata. Section 5 is devoted to the analysis and synthesis of the control software for the production cell. Section 6 concludes the paper.

2. The Design Language

In [Ericsson *et al.* 1999] and [Fersman *et al.* 2002], an extended version of timed automata with real time tasks is presented. The idea is to annotate each transition of an automaton with a task (an executable program with computing time and deadline), that will be triggered when the transition is taken. The triggered tasks will be scheduled to run according to a given scheduling policy. This provides a general task model for real time systems, which can be used to solve scheduling problems.

In this paper, we adopt a more expressive version of timed automata with tasks. We allow that an automaton and the annotated tasks may have shared variables. The shared variables may be updated by the execution of a task (or the automaton) and their values may effect the behaviour of the automaton.

2.1 Syntax

Assume a set of variables $\mathcal{D}$ ranged by u, v. We assume that the variables take values from finite data domains. The variables may be updated by assignments in the form: $u := \mathcal{E}$ where $\mathcal{E}$ is a mathematical expression. We use $\mathcal{R}$ to denote the set of assignments (e.g. $u := u + 1$).

Real Time Tasks. Let $\mathcal{P}$ ranged over by P, Q, R, denote a finite set of task types (executable programs written in a programming language). A task type may have different instances that are copies of the same program with different inputs. We assume that the *execution times* and *hard deadlines* of tasks in $\mathcal{P}$ are known[1].

Further assume that a task may update the data variables by the end of its computation using assignments in the form $u := \mathcal{E}$ (computed by the task and the value of $\mathcal{E}$ is returned when the task is finished). Thus, each task P is characterised as a triple denoted (C, D, A) with $C \leq D$, where C is the execution time of P, D is the relative deadline for P and A is the set of assignments updating data variables. The deadline D is a relative deadline meaning that when task P is released, it should finish within D time units. We shall use $C(P)$, $D(P)$ and $A(P)$ to denote the execution time, relative deadline and set of assignments of P respectively.

Timed Automata. Assume a finite set of actions *Act* and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b, τ etc to range over *Act*, where τ denotes a distinct internal action, and x_1, x_2 etc to range over $\mathcal{C}$. We use $\mathcal{B}(\mathcal{C})$ to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, =, \geq, >\}$, and C, D are natural numbers. We use $\mathcal{B}_I(\mathcal{C})$ for the subset of $\mathcal{B}(\mathcal{C})$ where all atomic constraints are of the form $x \prec C$ and $\prec \in \{<, \leq\}$. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*. These are the basic syntactical objects needed to define timed automata i.e. finite state automata extended with clocks.

We shall allow automata to test (or read) and update data variables. To read and test the values of data variables, we assume a set of predicates $\mathcal{B}(\mathcal{D})$ (e.g. $u \leq 10$). Let $\mathcal{B} = \mathcal{B}(\mathcal{D}) \cup \mathcal{B}(\mathcal{C})$ be ranged over by g called guards. Further let $\mathcal{A} = \mathcal{R} \cup \{x := 0 | x \in \mathcal{C}\}$ be called resets. We use r to stand for a subset of $\mathcal{A}$.

[1] Note that tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.

DEFINITION 1. *A timed automaton extended with tasks, over actions Act, clocks $\mathcal{C}$, data variables $\mathcal{D}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M\rangle$ where*

- ◦ *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations ranged over by l, m, n,*
  - *$l_0 \in N$ is the initial location,*
  - *$E \subseteq N \times \mathcal{B} \times Act \times 2^{\mathcal{A}} \times N$ is the set of edges, and*
  - *$I : N \mapsto \mathcal{B}_I(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- ◦ *$M : Act \hookrightarrow \mathcal{P}$ is a partial function assigning actions with tasks* [2].

Intuitively, a discrete transition in an automaton denotes an event triggering a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the event (or the associated task). Whenever a task P is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems). When the task is finished, the data variables will be updated by the assignments $A(P)$.

Note that a boolean can be used to denote the completion of a task. The arrivals of events may be guarded by the boolean, which implies that any task to be triggered by the events can not be started before the completion of this task. Thus, we can describe precedence constraints over tasks.

To handle concurrency and synchronisation, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [Larsen *et al.* 1995]) using the notion of synchronisation function [Hüttel and Larsen 1989]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronisable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronisation function [Hüttel and Larsen 1989] , the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

2.2 Operational Semantics

Semantically, an extended timed automaton may perform two types of transitions just as standard timed automata. But the difference is that delay transitions correspond to the execution of running tasks with the highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrival of new task instances.

We use valuation to denote the values of variables. Formally a valuation is a function mapping clock variables to the non–negative reals and data variables to the data domain. We denote by $\mathcal{V}$ the set of valuations ranged over by σ. Naturally, a semantic state of an automaton is a triple (l, σ, q) where l is the current control location, σ denotes the current values of variables, and q is the current task queue. We assume that the task queue takes the form: $[P_1(c_1, d_1), P_2(c_2, d_2)...P_n(c_n, d_n)]$

[2] Note that M is a partial function meaning that some of the actions may have no task. Note also that we may also associate an action with a set of tasks instead of a single one. It will not introduce technical difficulties.

where $P_i(c_i, d_i)$ denotes a released instance of task type P_i with remaining computing time c_i and relative deadline d_i.

Assume that there is a processor running the released task instances according to a certain scheduling strategy Sch e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever a new task arrives according to task parameters e.g. deadlines. In general, we assume that a scheduling strategy is a sorting function which may change the ordering of the queue elements only. Thus an action transition will result in a sorted queue including the newly released tasks by the transition. A delay transition with t time units is to execute the task in the first position of the queue for t time units. Thus the delay transition will decrease the computing time of the first task with t. If its computation time becomes 0, the task should be removed from the queue (shrinking).

- Sch is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\mathsf{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)]$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive.
- Run is a function which given a real number t and a task queue q returns the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor is available. Then the meaning of $\mathsf{Run}(q, t)$ should be obvious and it can be defined inductively. For example, let $q = [Q(4, 5), P(3, 10)]$. Then $\mathsf{Run}(q, 6) = [P(1, 4)]$ in which the first task is finished and the second has been executed for 2 time units.

Further, for a real number $t \in \mathbb{R}_{\geq 0}$, we use $\sigma + t$ to denote the valuation which updates each clock x with $\sigma(x) + t$, $\sigma \models g$ to denote that the valuation σ satisfies the guard g and $\sigma[r]$ to denote the valuation which maps each variable α to the value of $\mathcal{E}$ evaluated in σ if $\alpha := \mathcal{E} \in r$ (note that $\mathcal{E}$ is zero if α is a clock) and agrees with σ for the other variables. Now we are ready to present the operational semantics for extended timed automata by transition rules:

Definition 2. (Operational semantics) *Given a scheduling strategy* Sch [3]*, the semantics of an extended timed automaton* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0, σ_0, q_0) *is a transition system defined by the following rules:*

- $(l, \sigma, q) \xrightarrow{a}_{\mathsf{Sch}} (m, \sigma[r], \mathsf{Sch}(M(a) :: q))$ *if* $l \xrightarrow{g,a,r} m$ *and* $\sigma \models g$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma + t, \mathsf{Run}(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C(\mathsf{Hd}(q)) > t$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, (\sigma[A(\mathsf{Hd}(q))]) + t, \mathsf{Run}(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C(\mathsf{Hd}(q)) = t$

where $M(m) :: q$ *denotes the queue with* $M(m)$ *inserted in* q *and* $\mathsf{Hd}(q)$ *denotes the first element of* q.[4]

The first rule defines that a discrete transition can be taken if there exists an enabled edge, i.e. the guard of the edge is satisfied. When the transition is taken,

[3] Note that we fixed Run to be the function that represents a one-processor system.

[4] In case the task queue is empty we interpret the conditions $C(\mathsf{Hd}(q)) > t$ and $C(\mathsf{Hd}(q)) = t$ as true. Hence the two last transition rules becomes equal and corresponds to a delay transition in ordinary timed automata.

the variables are updated according to the resets of the edge, and the tasks of the action (if any) are inserted into the queue.

There are two kinds of delay transitions as defined in the second and the third rules. In the second rule, the transition corresponds to that the first task in the queue is executed. The transition is enabled when the invariants are satisfied and the task has not completed the execution during the delay. The transition results in updates of the clock variables and a decrease of the remaining execution time and deadline time of the task at the head of the queue.

In the third rule, the transition corresponds to completing the execution of the task at the head of the queue. The delay is the same as the remaining execution time. When the transition is taken the variables are updated according to the assignment of the task. Note that we assume a fixed execution time $C(P)$ for a task. It is possible to extend the model and the analysis so that the execution times vary in an interval between best and worst case execution time.

We shall omit Sch from the transition relation whenever it is understood from the context.

2.3 Analysis of Design Model

It has been shown that reachability and schedulability are decidable problems for the class of timed automata extended with tasks both for non-preemptive [Ericsson *et al.* 1999] and preemptive scheduling policies [Fersman *et al.* 2002]. In this section we give the definitions of these and define the boundedness problem, which should also be checked prior to code-synthesis. For a more detailed description on how to check schedulability of task extended timed automata we refer the reader to [Fersman *et al.* 2002] and [Fersman *et al.* 2003].

We use the same notion of reachability as for ordinary timed automata:

DEFINITION 3. (REACHABILITY) *We shall write* $(l, \sigma, q) \longrightarrow (l', \sigma', q')$ *if* $(l, \sigma, q) \xrightarrow{a} (l', \sigma', q')$ *for an action* a *or* $(l, \sigma, q) \xrightarrow{t} (l', \sigma', q')$ *for a delay* t. *For an automaton with initial state* (l_0, σ_0, q_0), (l, σ, q) *is reachable iff* $(l_0, \sigma_0, q_0) \longrightarrow^* (l, \sigma, q)$.

Thus a state (l, σ, q) is reachable if there is a sequence of transitions starting in the initial state (l_0, σ_0, q_0) and ending in (l, σ, q). We shall use reachability analysis to check safety properties of design models to conclude that undesired situations (states) are not reachable. For example, we may check that the size of the task queue for all reachable states is bounded. Note that the boundedness is a useful property, which may be used for estimation of memory consumption. Further, we may use reachability analysis to check the schedulability of a design model prior to the final implementation.

DEFINITION 4. (SCHEDULABILITY) *A state* (l, σ, q) *where* $q = [P_1(c_1, d_1), \ldots, P_n(c_n, d_n)]$ *is a failure denoted* $(l, \sigma, \mathsf{Error})$ *if there exists* i $(i \in [1..n])$ *such that* $c_i \geq 0$ *and* $d_i < 0$, *that is, a task failed in meeting its deadline. An automaton* A *with initial state* (l_0, σ_0, q_0) *is non-schedulable with scheduling policy* Sch *if and only if* $(l_0, \sigma_0, q_0) \longrightarrow^*_{\mathsf{Sch}} (l, \sigma, \mathsf{Error})$ *for some* l *and* σ. *Otherwise, we say that* A *is schedulable with* Sch.

Intuitively, a system is schedulable if for all reachable states, all tasks are guaranteed to meet their deadlines. It should be noticed that schedulability implies boundedness but not the other way around, as clearly many bounded ready queues are not schedulable.

In order to check the above properties of a system design, given as an extended timed automaton C (possibly a parallel composition $C_1 \parallel \ldots \parallel C_n$), we will need to take the behaviour of its environment into consideration. Recall that in the extended model of timed automata adopted in this paper, new task instances are released at the action transitions, corresponding to input signals received from the environment of the controller C. To model the environment, we compose in parallel with C timed automata models $E_1 \parallel \ldots \parallel E_m$ of its environment, and analyse properties of the complete system design $S^{Design} = C \parallel E_1 \parallel \ldots \parallel E_m$.

We shall see in the next section that the code-synthesis is guaranteed to preserve safety, schedulability and boundedness properties. This means that these properties of a system design can be checked prior to its implementation.

2.4 Deterministic and Executable Semantics

We shall consider S^{Design} as a design model. Code synthesis is to generate executable code from the design model that implements the controller. In general the behaviour of the design model according to the operational semantics (cf. Definition 2) is non-deterministic. Our goal is to extract deterministic behaviour for the controller, preserving safety properties satisfied by the design model. Thus, our problem is essentially to resolve non-determinism.

The sources of non-determinism in the operational semantics are time-delays and external actions. We use *time non-determinism* to mean that an enabled transition can be taken at any time-point within the time zone, while we use *external non-determinism* to mean that several actions may be simultaneously present from the environment which results in several enabled transitions. We say that a transition $e = l \xrightarrow{g,a,r} m$ is enabled in state $\mathsf{s} = (l, \sigma, q)$, denoted $\mathsf{Enabled}(e, \mathsf{s})$, when its guard is true i.e. $\sigma \models g$ and the environment is ready to synchronise on the action a.

We shall refine the design model for the controller as follows:

- External non-determinism is resolved by assigning priorities to transitions in the controller. Let $\mathsf{Pr} : E \mapsto \mathbb{Z}$ be a function assigning a unique priority to each edge, which defines an order that the guards of edges are evaluated in. If several transitions are enabled in a state, they should be taken in priority order.
- Time non-determinism is resolved by implementing the so-called maximal-progress assumption [Yi 1991]. Maximal-progress means that the controller should take all enabled action transitions until the system stabilises, i.e. no more action transitions are enabled. Similar ideas have been adopted in the *asynchronous time model* in the Statemate semantics of Statecharts [Harel and Naamad 1996] and the *run-to-completion* step of UML statecharts [Lilius and Paltor 1999].

We can now give a refined (deterministic) version of the operational semantics for extended timed automata:

Definition 5. (Deterministic semantics) *Let $A = \langle N, l_0, E, I, M\rangle$ be an extended timed automaton. Given a scheduling strategy* Sch *and a function* Pr *that assign priorities to edges the deterministic semantics is a labelled transition system defined by the rules:*

(1) $(l,\sigma,q)\xrightarrow{a}(m,\sigma[r],\mathsf{Sch}(M(a)::q))$ *if* $\mathsf{Enabled}(l\xrightarrow{g,a,r}m,(l,\sigma,q))$ *and there is no* $e\in E$ *such that* $\mathsf{Pr}(e)>\mathsf{Pr}(l\xrightarrow{g,a,r}m)$ *and* $\mathsf{Enabled}(e,(l,\sigma,q))$

(2) $(l,\sigma,q)\xrightarrow{t}(l,\sigma+t,\mathsf{Run}(q,t))$ *if* $(\sigma+t)\models I(l)$ *and* $C(\mathsf{Hd}(q))>t$ *and for all* $e\in E$ *and* $d<t$ *such that* $\neg\mathsf{Enabled}(e,(l,\sigma+d,q))$

(3) $(l,\sigma,q)\xrightarrow{t}(l,(\sigma[A(\mathsf{Hd}(q))])+t,\mathsf{Run}(q,t))$ *if* $(\sigma+t)\models I(l)$ *and* $C(\mathsf{Hd}(q))=t$ *and for all* $e\in E$ *and* $d<t$ *such that* $\neg\mathsf{Enabled}(e,(l,\sigma+d,q))$

Clearly the behaviour of the controller according to the refined semantics is deterministic. Safety properties in the design model are preserved since the behaviour defined by deterministic semantics is included in the behaviour defined by the operational semantics. That is, all sequences of transitions (i.e. executions) of a design model for a controller according to the deterministic semantics can also be taken according to the operational semantics.

Note that according to the semantics, we may have automata that exhibit zeno-behaviours, that is, infinite sequences of action transitions within a finite time delay. Such automata correspond to non implementable design models, and they should be discovered by schedulability analysis as non schedulable. To simplify the analysis, we adopt the following syntactical restriction. We shall consider only automata in which each cycle contains at least one edge associated with a task.

3. Code Synthesis

Now we show how to transform a design model to an executable program according to its deterministic semantics. We assume a generic target platform which guarantees the synchronous hypothesis and on which the associated tasks will consume their given computing times to execute. As our transformation preserves the deterministic semantics, the schedulability of a design model will be preserved by the generated code when it is executed on the target platform.

We assume that the target OS provides the following generic features:

- threads with unique priority levels,
- a scheduler based on fixed priority assignment,
- interrupt handling routines.

The first two requirements are needed to map the notion of tasks from the design model to the generated code. Tasks have unique priorities in the model. Thus when they are mapped to threads in the implementation the threads must also have unique priorities. The last requirement is needed to encode control automata, which is done by interrupts using interrupt handing routines.

The intended target platform for the generated code is a single embedded processor, but the organisation of the code is more easily illustrated by assuming two processors as shown in Fig. 1. The code of the controller and the tasks execute

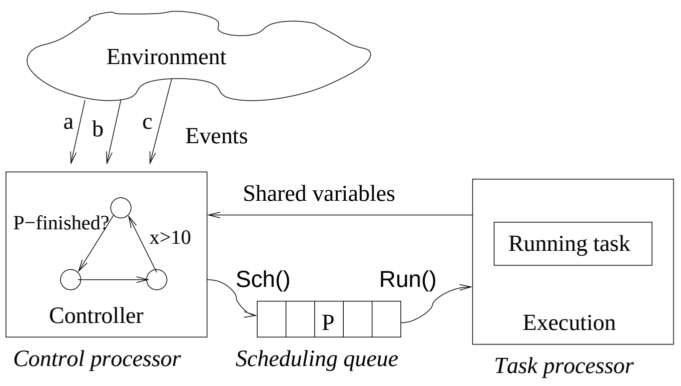


Fig. 1: The logical view of the executing system, with a *control processor* handling events from the environment and a *task processor* executing task code. Communication between the processors is limited to the task queue and shared variables.

on two separate logical processors, a *control processor* and a *task processor*. The interaction between the two is limited to the scheduling queue and shared variables. The control processor receives events from the environment and inserts tasks into the scheduling queue. The task processor executes the task at the head of the scheduling queue and updates the shared variables when a task is done.

To realise the execution model on a single processor, we execute the code of the controller automata as a separate thread with a priority higher than all the other threads.

3.1 Handling Tasks and Variables

Tasks are executed in threads, one thread for each task type, with priorities lower than the controller thread. Scheduling of task threads and management of the ready queue are handled by the target operating system. We assume boundedness of the queue length, meaning that the memory allocated for the task queue can be fixed at compile-time and no exception handling for queue overflow is needed at run time.

Data variables in the design model are mapped to global integer variables in the generated code. To encode clocks, let sc be a global system clock. For each clock x in the timed automata, let x_{reset} be an integer variable holding the system time of the last clock reset. The value of the clock is then $(sc - x_{\mathrm{reset}})$, and a reset can be performed as $x_{\mathrm{reset}} := sc$.

3.2 Encoding and Executing the Controller Automata

In the generated program, the controller automata are encoded as four look-up tables and two functions, as shown in Fig. 2. Three of the look-up tables are static and used to encode the edges, the locations, and the synchronisations respectively. The fourth table is dynamic and used to hold the set of currently active edges, i.e. edges leaving the current control location (or locations, if the controller consists of parallel automata). We use ACTIVE to denote the list of active edges.

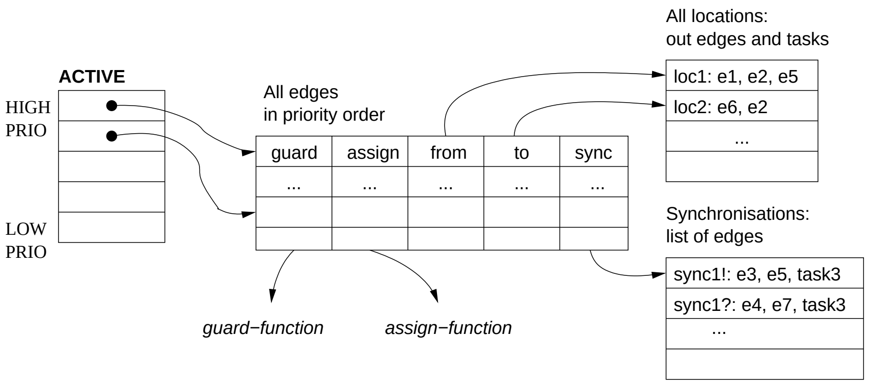


Fig. 2: Encoding of controller automaton in look-up tables.

The table containing all edges is sorted in priority order. For each edge there are five fields: *guard*, *assign*, *from*, *to* and *sync*. The *guard* and *assign* fields are references to code that when invoked will evaluate the guard and perform the assignments of the edge. The fields *from* and *to* are references into the table of locations. The field *sync* is either empty if the edge has no synchronisation label, or a reference to a list of edges having the complementary synchronisation label.

In the table containing all locations, we store for each location a list of all edges leaving the location. In the table containing synchronisation labels, we store references to all edges at which the label occurs, and also the set of tasks associated with the label. Note that some of the information stored in the tables (such as the synchronisation label of an edge) is intensionally duplicated to improve efficiency.

The list of active edges, ACTIVE, is managed by the procedure shown in Table I where we use $\overline{a}$ to denote the complementary synchronisation label of a, e.g. *sync*! for *sync*?. We use *aut* to denote a function that takes a location as parameter and returns an unique identifier for the automaton that the location belongs to. We use *sort* to denote a sorting function that takes as input two parameters: a list of edges, and a function assigning unique priorities to edges, and returns as output a list of edges sorted according to the assigned priorities.

The procedure in Table I is executed by the controller thread whenever an event (such as timeout or arrival of an external event) has occurred. When executing, no new events are processed and the timers are not updated, i.e. the whole procedure is executed in a critical region. Note that this means that the tasks will not be able to update shared variables while the procedure is executing.

Initially the list of active edges consists of all edges leaving the initial locations. The procedure scans ACTIVE in priority order and evaluates the corresponding guards. If a guard is found to be satisfied, there are two cases:

- no synchronisation - the assignment is performed and the information in the table of locations is used to update the list of active edges and to release any tasks associated with the edge.
- synchronisation - (i.e. the edge is labelled with an action label) the information in the table of synchronisations is used to find an active edge belonging to another control automata with complementary action label and satisfied

TABLE I: Procedure, in pseudo-code, that executes the encoded controller.

```
Initially:
$\mathsf{ACTIVE} := \{l \xrightarrow{g,a,r} l' | l = l_0\}$
$\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$
```

```
Algorithm:
START:
for each $l \xrightarrow{g,a,r} l'$ in $\mathsf{ACTIVE}$ do
    if $\sigma$ satisfies $g$ then
        if $a = \tau$ then
            $\sigma := \sigma[r]$
            remove $\{n \longrightarrow n' | n = l\}$ from $\mathsf{ACTIVE}$
            add $\{n \longrightarrow n' | n = l'\}$ to $\mathsf{ACTIVE}$
            $\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$
            $q := \mathsf{Sch}(M(a) :: q)$
            goto START
        else
            if exists $m \xrightarrow{g',\overline{a},r'} m'$ in $\mathsf{ACTIVE}$ s.t. $\sigma$ satisfies $g'$ and
                    $aut(m) \neq aut(l)$ then
                if $a$ is sending then
                    $\sigma := \sigma[r]$; $\sigma := \sigma[r']$
                else
                    $\sigma := \sigma[r']$; $\sigma := \sigma[r]$
                remove $\{n \longrightarrow n' | n = l \vee n = m\}$ from $\mathsf{ACTIVE}$
                add $\{n \longrightarrow n' | n = l' \vee n = m'\}$ to $\mathsf{ACTIVE}$
                $\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$
                $q := \mathsf{Sch}(M(a) :: M(\overline{a}) :: q)$
                goto START
            fi
        fi
    fi
od
```

guard. If such an edge is found, the compound transition is performed i.e. the assignments of the two edges are performed, ACTIVE is updated and the tasks associated with action label are released.

When a transition has been executed the procedure returns to `START` and re-examines the (updated) list of active edges to check for another transition to be taken. The procedure will continue to do so as long as enabled edges exist in ACTIVE. When there are no more enabled edges, the procedure will terminate and let the OS execute task threads. As an effect, the procedure implements a so-called *run-to-completion* step ensuring that the generated code has the maximal-progress behaviour assumed by the deterministic semantics (see Section 2.4). Note that since ACTIVE is always kept sorted and the procedure always scans from the head of the list, the implementation is also deterministic with respect to external actions.

3.3 Correctness

In this section we argue for the correctness of the code synthesis by emphasising how the deterministic semantics (c.f. Definition 5) of timed automata with tasks is realised in the generated code. The three components of a semantic state (l, σ, q) are mapped to the following parts of the synthesised code:

- l: the source locations of the edges in ACTIVE (the list of active edges),
- σ: integer variables for the data variables in σ, and the difference between reset-time and system time for the clocks in σ,
- q: the scheduling queue of the operating system.

Note that there is no explicit representation of control location l in the synthesised code, instead the current control location is implicitly represented by the edges in ACTIVE.

Initially, when the synthesised code starts to execute, the semantic state (l_0, σ_0, q_0) is represented in the code as:

- l_0: ACTIVE is initialised with the edges leading out from the initial locations,
- σ_0: data variables are initialised to their initial values and the reset-time of clocks are initialised to the system time at startup,
- q_0: the scheduling queue is empty.

The transition rules defined in the deterministic semantics (Definition 5) are mapped in the following way:

(1) Discrete transition - $(l, \sigma, q) \xrightarrow{a} (l', \sigma', q')$ is handled by the procedure in Table I. All edges that lead out from the source location are removed from ACTIVE and all edges that lead out from the target location are added to ACTIVE. Data variables are updated according to the assignments. Clocks are reset by assigning their reset-time to the current value of the system clock.

(2) Delay transition without task termination - $(l, \sigma, q) \xrightarrow{t} (l, \sigma', q')$ corresponds to the execution of a task. The location component of the state does not change during task execution. Data variables are not updated. Clocks are updated by the delay t during task execution since the difference between the last reset-time and the system clock will increase. In the queue q, the remaining execution time of the task at the head of the task queue, as well as the deadline of all tasks, will be t time units shorter.

(3) Delay transition with task termination - $(l, \sigma, q) \xrightarrow{t} (l, \sigma', q')$ corresponds to execution of a task. The location component of the state does not change during task execution. Data variables are updated by the assignment of the task. Clocks are updated by the delay t during task execution since the difference between the last reset-time and the system clock will increase. The queue is reduced by removing the task at the head of the queue, and the relative deadline of the other tasks will be t time units shorter.

The refinements introduced to resolve non-determinism are implemented as follows:

Priority - Recall that all edges are assigned unique priorities. The controller thread evaluates the guards of the active edges in priority order from high to low. The first transition whose guard is satisfied is taken. In the case of synchronisation the complementary transition with the highest priority whose guard is satisfied is taken. The order of the active edges is maintained when replacing outgoing edges of the source location with those of the target location by resorting before proceeding.

Maximal Progress - The code has maximal-progress behaviour since, when a discrete transition has been taken the list of active edges is checked again from the beginning. Only when no edge has a satisfied guard the controller will suspend and let other threads with lower priority (i.e. tasks) execute.

3.4 Prototype for legOS

The code synthesis described above has been implemented in a version of the TIMES tool [Amnell *et al.* 2002] to generate code for the legOS [LegOS-team 2002] operating system — a small open source operating system for the Hitachi H8 processor — embedded into the LEGO® Mindstorms RCX control brick. The OS is implemented in C and for simplicity we use C as implementation language for the tasks as well.

The Hitachi H8 processor in the RCX unit is equipped with 32 kB of RAM, a typical setup for the type of embedded systems that our code generation is intended for. The I/O interface consists of three sensor inputs and three actuator outputs.

In legOS threads are separate control flows that are scheduled on the CPU by the operating system. The scheduler implements preemptive fixed priority scheduling and handles up to 20 priority levels. Threads with equal priorities are executed in a round robin fashion. This limits the number of task types in the design to at most 19, since each task type needs a unique priority level and the highest priority level is reserved for the controller thread.

Another limitation in the current version of legOS is in the interrupt handling. The OS provides so-called wake-up functions to program event handling. A wake-up function is a general mechanism provided by legOS that lets a thread wait for some condition (such as the release of a semaphore, a timeout, a key press etc.). A thread registers a boolean function that the scheduler will execute when it looks for the next thread to run, which it will do when the current thread is suspended or periodically every 20 ms [5].

To make sure that the control procedure of Table I is executed at every event handling, we encode the entire control procedure as a wake-up function that always returns false, and is associated to a task at highest priority level. In this way, the control procedure is ensured to be executed by the operating system (atomically and at OS priority level) just after every event handling, and just before the operating system determines the next task thread to be executed.

For each task type a corresponding thread is created at boot time. The task threads contains a loop repeating a suspend call, and the task body to be executed. The suspend call registers a wake-up function that checks for a non-zero value of an element in the integer array *release list.* This array contains one element for each task type. When the controller releases a task the corresponding element is incremented,

[5] The period of the scheduler (the time-slice) can be modified by recompiling the OS.

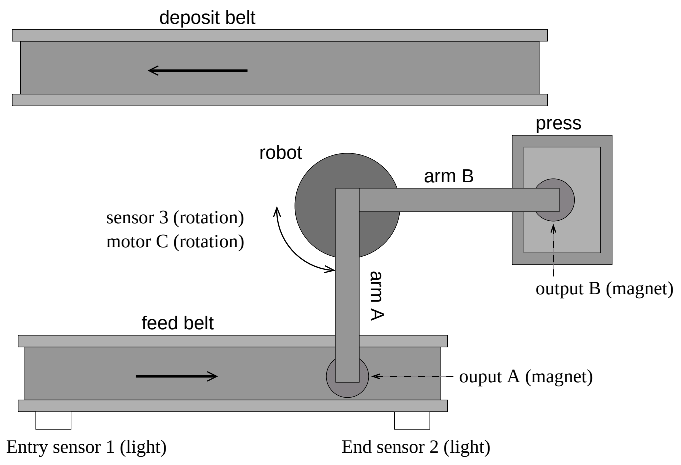


Fig. 3: The LEGO® Production Cell.

so that the wake-up function of the task thread will return and resume the task thread. When the task body has executed the corresponding element in the release list is decremented, and the thread is suspended again at the start of the loop. Note that since priorities are fixed, the release list can be used as a representation of the current state of the ready queue.

Sensor readings in legOS are available to the generated program as variables which are updated independently by the hardware. Sensor variables are either read by the task code, or used directly in guards of the control automata. This means that checking if a transition is enabled becomes a condition only involving variables (external and other), i.e. essentially the same as a guard.

In the current prototype we associate tasks with locations instead of edges. The tasks are released when the location is entered. Such design models can easily be transformed into models with tasks on edges by associating the task of a location with all edges leading to the location. Furthermore we impose the syntactic restriction that a control automaton may not use both a sending and a receiving action label with the same name (e.g. not both *sync*! and *sync*? in the same automaton).

4. Modelling and Design of a Production Cell

In this section, we show how to use our design language to model and design the control software for an industrial robot. The production cell is a unit in a metal plate processing plant in Karlsruhe. The model has been developed by FZI in Karlsruhe [Lewerentz and Lindner 1995] as a benchmark example of concurrent and safety-critical system software development. In this paper, we use a LEGO® model of the production cell based on the model developed by FZI but with some simplifications. The LEGO® production cell system contains four subsystems, the *feed belt*, the *robot*, the *press* and the *deposit belt*, corresponding to the physical components as shown in Fig. 3. Each of the subsystems performs a specific operation during the plate processing:

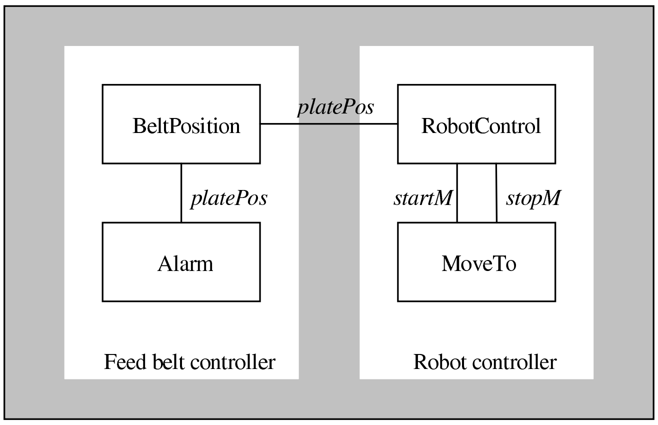


Fig. 4: The overall control structure.

The *feed belt* transfers a plate from the entry position to the end position where the robot arm named *arm A* can pick up the plate. The *press* forges the plate delivered by *arm A* of the robot, and the forged plate is unloaded from the *press* by *arm B* of the robot. The *robot* moves to the position where *arm A* points to the *feed belt* and picks up a plate. It then rotates until *arm A* points to the *press* where *arm A* delivers a plate. When a plate has been forged in the *press*, it moves to the position where *arm B* points to the *press* and picks up the forged plate. Afterwards, the *robot* rotates until *arm B* points to the *deposit belt* where it delivers the plate. The *deposit belt* receives a forged plate from *arm B* of the *robot.*

4.1 Overall Control Structure

Each subsystem of the *feed belt* and the *robot* comprises sensors and actuators for the physical component in the subsystem plus a controller (i.e. a control program) for controlling the sensors and actuators. In our version of the production cell, both the *press* and the *deposit belt* contain only the physical components (without any sensor and actuator attached) as media for the *robot* to convey a plate. The status of a plate in both components are modelled in the robot controller by means of internal variables. The overall control structure of the production cell is sketched as in Fig. 4.

Robot Controller. The robot controller is composed of two processes (or sub-controllers), RobotControl and MoveTo as shown in Fig. 4. The RobotControl process controls the overall motions of the robot, and communicates with both the feed belt controller and the local process MoveTo. The MoveTo process controls the robot movements according to the rotation sensor (i.e. sensor 3 in Fig. 3) information and the requests from RobotControl.

The RobotControl process communicates with the feed belt controller by means of the shared variable `platePos` which indicates whether there is a plate on the feed belt. It communicates with MoveTo by means of communication channels i.e. `startM` to start the robot movement and `stopM` to stop the robot movement.

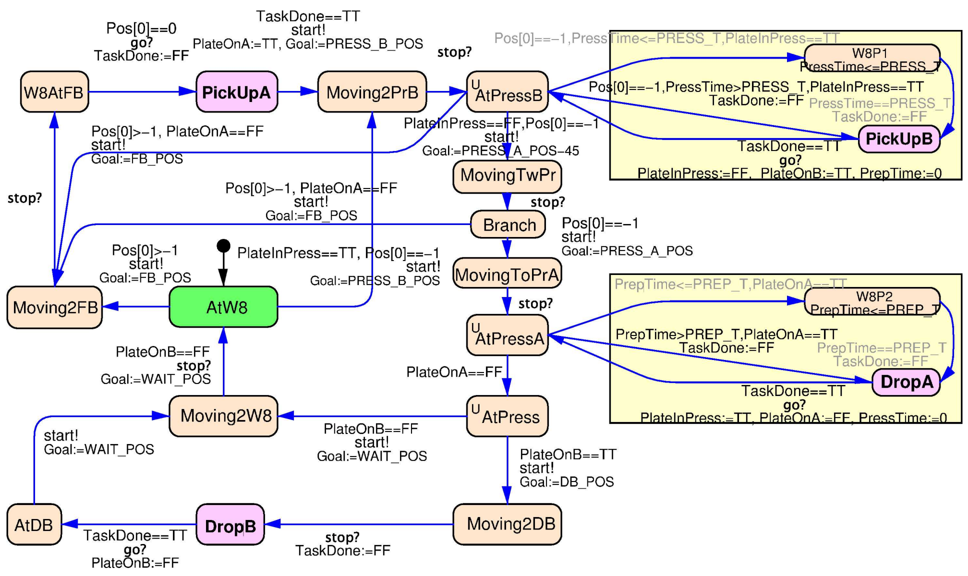


Fig. 5: The automaton RobotControl. Labels in bold font indicate associated tasks.

Feed Belt Controller. The feed belt controller is composed of the BeltPosition process and the Alarm process. The BeltPosition process updates the positions of the plates on the feed belt periodically after they are detected by the light sensor. The Alarm process handles the light sensor (i.e. sensor 1 in Fig. 3) at the entry position of the feed belt. The light sensor senses the arrival of a plate on the feed belt. The communication between BeltPosition and Alarm is through the shared variable `platePos` which indicates the position of a plate on the belt. Note that the feed belt controller cannot stop the belt. The bricks will be picked up "on-the-fly" by the magnet on *arm A*.

4.2 Robot Controller Model

The robot controller is modelled as two extended timed automata, named Robot-Control and MoveTo. The shared variable `platePos` is modelled by an integer array `Pos`, and the communication channels `startM` and `stopM` are respectively modelled by synchronisation channels `start` and `stop`, together with a shared integer variable `Goal` that sets the goal position that the robot should reach.

Automaton RobotControl. Fig. 5 shows the concrete model of the automaton RobotControl. Recall that in our prototype implementation tasks are associated with locations. In the automata shown in this section and in Appendix A, tasks are indicated as labels with bold font in the locations.

In Fig. 5, `Pos[0]` is the first element in array `Pos`. `Pos[0]>-1` indicates that there is a plate on the feed belt, `Pos[0]==-1` indicates that there is no plate on the feed belt, and `Pos[0]==0` indicates that the plate is at the position where *arm A* of the robot can pick it up.

The boolean variable `PlateInPress` is used to model the status of the press (`TT` means that there is a plate on the press and `FF` means no plate on the press).

The constant `PRESS_T` is used to model the time needed for the press to forge a plate, and the constant `PREPARE_T` is used to model the time needed for the press to be ready to process a new plate. The clocks `PressTime` and `PrepareTime` are respectively used to measure whether `PRESS_T` and `PREPARE_T` are reached.

The boolean variables `PlateOnA` and `PlateOnB` are used respectively to indicate the status of *arm A* and *arm B*. `PlateOnA==TT` denotes that *arm A* of the robot holds a plate, and `PlateOnA==FF` denotes that *arm A* is empty. `PlateOnB==TT` denotes that *arm B* of the robot holds a plate, and `PlateOnB==FF` denotes that *arm B* is empty. The boolean variable `TaskDone` is shared with the tasks released by the process. `TaskDone==TT` denotes that the current task e.g. the pickup task of *arm A* is finished, and `TaskDone=FF` denotes that the current task has not finished yet.

The constants `WAIT_POS`, `FB_POS`, `PRESS_A_POS`, `PRESS_B_POS` and `DB_POS` are five goal positions for the robot to reach in the robot working space.

The automaton is initially at location AtW8, corresponding to that the robot is waiting for the control commands (or control events). The rest of the locations can be classified into three groups corresponding to the robot actions:

- The robot is moving from one position to the other, e.g. the locations Moving2FB and Moving2PrB. At the former location the robot is rotating so that *arm A* points to the feed belt, and at the latter the robot is rotating so that *arm B* points to the press. While waiting in these locations MoveTo control the movement of the robot.
- The robot is waiting for an event, e.g. the locations W8AtFB and AtPressB. The former represents that the robot stops at the position where *arm A* points to the feed belt and waits for the event `Pos[0]==0`, and the latter that the robot stops at the position where *arm B* points to the press and waits for the event `PlateInPress==TT`.
- The robot is carrying out the pickup or drop tasks, e.g. the locations PickUpA and DropA. At the former location *arm A* is picking up a plate from the feed belt, and the latter *arm A* is dropping the plate onto the press.

On each of the edges to PickUpA, PickUpB, DropA and DropB, there is a executable task associated. For example, on the edge to PickUpA the task PickUpA is released for *arm A* to pick up a plate from the feed belt, while the edge to DropA is associated with the task DropA by which *arm A* drops a plate onto the press.

Automaton MoveTo. The automaton MoveTo is shown in Fig. 6. The variable `RobotAngle` stores the sensor information about the current robot position. The value in `ROTATION_1` stores the updated robot position while the robot is moving. Clock `x` is used to guarantee that `RobotAngle` is updated before it is used.

The automaton is initially in location Entry corresponding to that the robot stops and waits for the signal from RobotControl over channel `start`. Having received the signal the task RdAngSen is released that reads the sensor information about the current robot position into the variable `RobotAngle` and the automaton moves to location Read. The task RdAngSen has deadline 16, see Appendix A, so the invariant $\mathtt{x} \leq 30$ together with the guards $\mathtt{x} = 30$ on the outgoing edges will guarantee that `RobotAngle` is updated before the location is left.

Based on the value in `RobotAngle`, the automaton moves either to location MR if $\mathtt{RobotAngle} \leq \mathtt{Goal}$ or to location ML if $\mathtt{RobotAngle} \geq \mathtt{Goal}$, or to location Entry if $\mathtt{RobotAngle} = \mathtt{Goal}$.

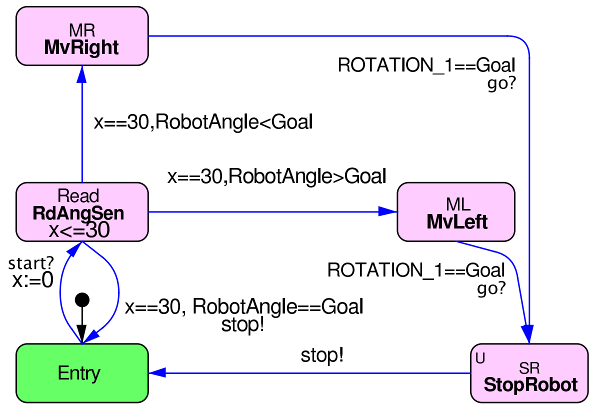


Fig. 6: The automaton MoveTo.

On the edge to location MR, the associated task MvRight commands the motor to rotate clockwise the robot. When the robot is in the desired position, i.e. `ROTATION_1` = `Goal`, the automaton takes a transition to location SR.

Similar to MR, on the edge to ML, the associated task MvLeft commands the motor to rotate anti-clockwise the robot. When the robot is in the desired position, i.e. `ROTATION_1` = `Goal`, the automaton takes a transition to location SR.

On both edges to SR, an associated task StopRobot is released to stop the motion of the robot. The automaton takes an urgent transition to location Entry when the robot is stopped, at the same time it informs RobotControl over channel `stop`.

4.3 Feed Belt Controller Model

Similar to the robot controller, we use two timed automata extended with real-time tasks to model the two processes of the feed belt controller. The two automata BeltPosition and Alarm are given in Appendix A.

Automaton Alarm. The automation Alarm is shown in Fig. 8 in Appendix A. Before it enters the initial location Calib the task Calib is released that performs calibration of the background light by measuring the mean light value, and saves it in the shared variable `normalLight`. Finally the task sets the boolean variable `AlarmTaskDone` to `TT`. When the sensor is calibrated, i.e., `AlarmTaskDone==TT`, the automaton takes a transition to location WaitFrontEdge.

In location WaitFrontEdge, the automaton waits until the light value measured by sensor 1 (cf. Fig. 3) falls below 80% of the normal light. This indicates that the front end of a plate has passed through. The automaton then moves to location WaitBackEdge, where it waits for the whole plate to pass through the sensor, i.e. the measured light value rises to above 90% of the normal light. Once the whole plate has passed through the sensor, the automaton takes a transition to location AtStart and releases the AtStart which inserts a value into the array `Pos` .

In location AtStart, the clock `ArrivalTime` is used to measure the time elapsed since the plate was detected. When it reaches to the value equal to `SAFE_CALIB`, it is safe to accept a new plate and return to Calib. The time constant SAFE_CALIB is obtained by experiments.

Automaton BeltPosition. The automaton BeltPosition is shown in Fig. 9 in Appendix A. It is initially in location Idle. When there is a plate on the belt, indicated by `Pos[0]>-1`, the automaton moves to location Active and releases task UpdatePos.

In location Active, the task UpdatePos is released periodically to update the positions of the plates in array `Pos`. The period is represented by an invariant `x <= UpdateTime`, where `x` is again a clock. As soon as there is no plate on the belt, which is indicated by `Pos[0]==-1`, the automaton returns to the initial location Idle.

We could also have a process to handle the light sensor (i.e. sensor 2 in Fig. 3) at the end position of the feed belt to sense the arrival of a plate. In the current implementation we ignore this sensor and use `Pos[0]` to indicate whether a plate has reached the end position or not.

5. Analysis and Code Synthesis for the Production Cell

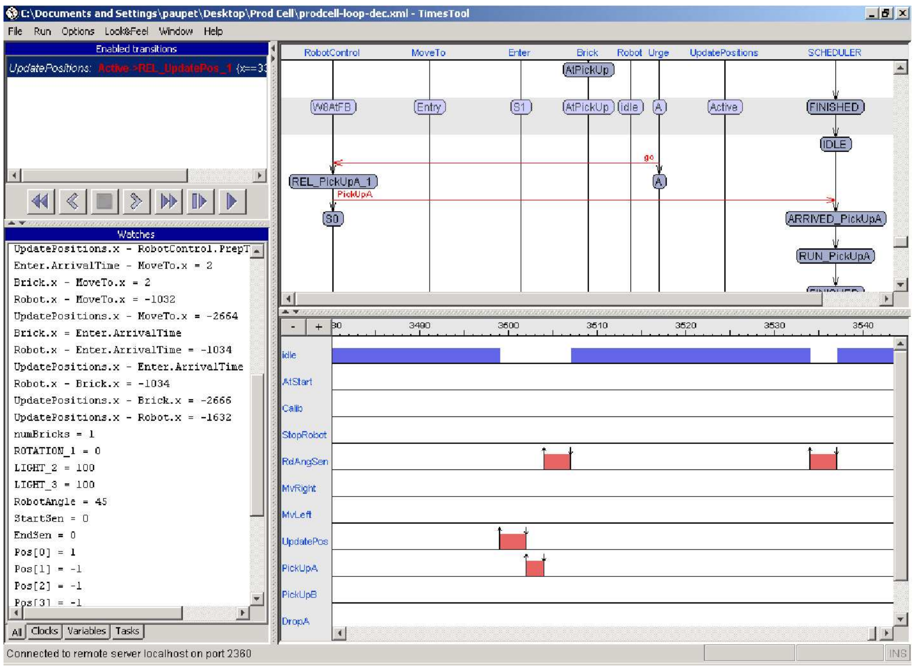
The analysis and code synthesis presented in Section 3 have been implemented in the TIMES[6] tool [Amnell *et al.* 2002]. The tool, shown in Fig. 7 performs analysis of a system model by transformation from the model of timed automata with tasks to the model of timed automata extended with subtraction operations in the clock assignments. During the transformation, a scheduler automaton is created and composed in parallel with the other automata. Its purpose is to ensure that the released tasks are scheduled according to the chosen policy, and to indicate if a task fail to meet its deadline. For a detailed description about the encoding see [Fersman *et al.* 2002].

The parameters and tasks used in the analysis of the production cell model are shown in Table II of Appendix A. Note that in the production cell model, tasks execute according to the fixed priorities listed in column P of Table II. The code performed by the tasks is given in column Description and Interface and is performed by the scheduler automaton at the time-point when a task finishes its execution.

Environment Model: In order to analyse the behaviour of the production cell model, the abstract behaviour of its environment is modelled with the two timed automata Brick and Robot shown in Fig. 10 and Fig. 11 of Appendix A. The automaton Brick models bricks arriving on the feed belt, and Robot models the position of the robot, and how it behaves when the robot rotates. The interaction between the environment and the control program is modelled using three shared variables: DIR used to control the rotation of the robot, POS used to model the robot position, and LIGHT_2 used to sense the value of the light sensor at the start of the feed belt.

Analysis: The two most important properties w.r.t. code-synthesis are schedulability and boundedness, described in section 2.3. We have used TIMES to check that the production cell model is *schedulable* with fixed priority scheduling in the sense that all released tasks are guaranteed to meet their deadlines in all possible runs

[6] More information about the TIMES tool can be found at the web site `www.timestool-.com`.


C:\Documents and Settings\paupet\Desktop\Prod Cell\prodcell-loop-dec.xml - TimesTool
File Run Options Look&Feel Window Help
Enabled transitions
UpdatePositions: Active->REL_UpdatePos_1 {x==3
Watches
All Clocks Variables Tasks
RobotControl MoveTo Enter Brick Robot Urge UpdatePositions SCHEDULER
AtPickUp
W8AtFB Entry S1 AtPickUp Idle A Active FINISHED
IDLE
go
REL_PickUpA_1 A
PickUpA
S0 ARRIVED_PickUpA
RUN_PickUpA
3490 3500 3510 3520 3530 3540
idle
AtStart
Calib
StopRobot
RdAngSen
MvRight
MvLeft
UpdatePos
PickUpA
PickUpB
DropA
INS

```
UpdatePositions.x - RobotControl.PrepT
Enter.ArrivalTime - MoveTo.x = 2
Brick.x - MoveTo.x = 2
Robot.x - MoveTo.x = -1032
UpdatePositions.x - MoveTo.x = -2664
Brick.x = Enter.ArrivalTime
Robot.x - Enter.ArrivalTime = -1034
UpdatePositions.x - Enter.ArrivalTime
Robot.x - Brick.x = -1034
UpdatePositions.x - Brick.x = -2666
UpdatePositions.x - Robot.x = -1632
numBricks = 1
ROTATION_1 = 0
LIGHT_2 = 100
LIGHT_3 = 100
RobotAngle = 45
StartSen = 0
EndSen = 0
Pos[0] = 1
Pos[1] = -1
Pos[2] = -1
Pos[3] = -1
```

Connected to remote server localhost on port 2360

Fig. 7: A screen shot of the TIMES tool.

of the system. In addition, we have checked that the ready queue of the system is guaranteed to be *bounded* to three, meaning that the number of simultaneously released tasks will never be more than three. We have also checked that for each task type, the bound is one, i.e. there is never more than one task instance of each type released simultaneously. A number of other correctness properties of the production cell behaviour have also been checked, e.g. that the robot will always be ready to pick up a brick before the brick reaches the end of the feed belt, and that whenever the robot tries to pick up a brick from the belt, there is a brick available. By reachability analysis, we found that the nodes W8P1 and W8P2 of automaton RobotControl are unreachable. This information has been used in the code synthesis to produce smaller code. In Appendix A.1 we list the verified properties in the input format currently accepted by the TIMES tool.

Analysis Results: The system has been verified on a machine equipped with two 1.8 GHz AMD processors and 2 GB of main memory, running Mandrake Linux. TIMES consumes 207 MB of memory and 11 minutes to perform exact analysis of the most time and space consuming property above (i.e. any property that generates the full state space, e.g. the schedulability analysis). If the same property is checked using the over approximation option of TIMES (based on the convex-hull approximation described in [Daws and Tripakis 1998]), the analysis requires 13 MB and 9 seconds on the same machine.

During the analysis we found and corrected several problems in the model, e.g. the deadlines of tasks DropB, DropA, PickUpB, UpdatePos, and RdAngSen were adjusted[7]; the priorities of the tasks released by automaton MoveTo were adjusted to preserve the intended execution order[8]; the constant `PREPARE_T` were found to be too short; the behaviour of automaton MoveTo was modified as the boundedness analysis showed that two instances of task ReadAngleSen could erroneously be ready for execution simultaneously.

During the debugging, we often found the simulator of TIMES very useful. In particular the Gantt chart view, shown in the lower right part of the screen shot in Fig. 7 proved useful for tracing the executions of the tasks. In the Gantt chart view it is illustrated how tasks are executed according to the chosen scheduling policy [Amnell *et al.* 2002].

5.1 Generated legOS*-code*

We have used TIMES to automatically generate C-code that has been compiled and executed on a LEGO® RCX-brick running the legOS operating system. The code generated from the analysed model of the production cell is listed in Appendix B. The source code consists of 550 lines of C-code which results in an executable file of 4892 bytes.

The code consists of two parts: one generic run-time system (listed in Appendix B.2) that is part of any generated code for the legOS target, and one design specific part (listed in Appendix B.1). The design specific code for the production cell model is essentially an encoding of the control automata in five look-up tables, as described in Section 3. The edge table has four fields: `active`, `from`, `to`, and `sync` where the boolean value `active` indicates if the edge is currently in the list ACTIVE. The other fields are indexes into the location and synchronisation tables. The design specific part of the code also include the task bodies as part of the program text.

The information in the look-up tables is used by the run-time kernel, which basically is an implementation of the event handling procedure shown in Table I. The main function is `check_trans()` that loops through the edge table and evaluates the guards of the edges marked as active. When a guard is found to be satisfied, and the edge does not have a synchronisation channel, the transition is taken, meaning that the assignments are performed (by a call to the function `assign()` with the edge identifier as parameter), the field `active` in the edge table is cleared for all edges leaving the source location and set for all edges leaving the target location. Finally, the task of the target location (if any) is released. When a transition has been taken the loop starts over, `check_trans()` will only return when no more transitions can be taken.

If the edge is labelled with a synchronisation channel, i.e. the `sync` field in the edge table contains a non-negative synchronisation identifier, a separate function `check_sync()` is called. This function determines, by using the table `chanusage` which stores back references from channel names to edges, if there is an enabled edge to synchronise with. If a synchronisation is possible the assignments of both edges are performed.

The code also includes macro and type definitions as shown in Appendix B.3.

[7] Initially, all deadlines were set to 10ms. The deadlines of the tasks were changed to 13ms, 16ms, 17ms, 17ms, and 16ms respectively.

[8] The task in MoveTo should execute so that task RdAngSen precedes task MvRight or MvLeft.

6. Conclusions

In this article, we have shown how to synthesise executable code with predictable behaviour from high level design descriptions of embedded real-time systems. As design language, we proposed to use the model of timed automata extended with real-time tasks. For this model, it has been shown previously that design problems such as schedulability and reachability checking are decidable and can be efficiently checked using model-checking techniques.

The technique for code-synthesis developed in this paper is based on a deterministic subset of the operational semantics of extended timed automata, which is guaranteed to preserve timing, resource, and logical constraints imposed on the model. Based on the presented deterministic semantics, we have shown that executable code can be generated for a generic target platform assuming that it guarantees the synchronous hypothesis and that the associated tasks consume their given computing times. As prerequisite for code generation, we require that the model has been analysed to be schedulable (i.e. all tasks must be guaranteed to meet their deadlines). Other analysis results can be used to further optimise the generated code by e.g. limiting the amount of allocated memory, or to exclude code segments which are guaranteed to be unreachable (i.e. dead code).

A prototype C-code generator for the legOS operating system has been implemented in the TIMES tool, and applied in a case study in which the control software of a production cell is designed. The production cell is built in LEGO® and controlled by a Hitachi H8 based LEGO® Mindstorms control brick. The design, analysis, and code generation was successfully completed in the TIMES tool, and the generated C-code was compiled and downloaded to the control brick, and executed to control the production cell.

As future work we plan to adjust the code generation to target other operating systems than legOS, such as RT-Linux. Furthermore, we consider to extend the code synthesis to also generate code for the scheduling kernel and the necessary run-time system functions. This would make it possible to generate a tailored made run-time system optimised to support only functions being used in the designed system, and further optimised based on results from analysing the design model.

Acknowledgement

The TIMES tool has been implemented in large parts by Leonid Mokrushin.The authors would like to thank him for his excellent work. We would also like to thank the anonymous reviewers of the Nordic Journal of Computing for their valuable comments on our work.

References

AMNELL, TOBIAS, FERSMAN, ELENA, MOKRUSHIN, LEONID, PETTERSSON, PAUL, AND YI, WANG. 2002. TIMES - A Tool for Modelling and Implementation of Embedded Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Volume 2280 of *Lecture Notes in Computer Science*. Springer–Verlag, 460ff.

BENGTSSON, JOHAN, GRIFFIOEN, W.O. DAVID, KRISTOFFERSEN, KÅRE J., LARSEN, KIM G., LARSSON, FREDRIK, PETTERSSON, PAUL, AND YI, WANG.

1996. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In *Proc. of the 8th Int. Conf. on Computer Aided Verification*, Number 1102 in Lecture Notes in Computer Science. Springer–Verlag, 244–256.

BERRY, G. AND GONTHIER, G. 1992. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. *Science of Computer Programming 19*, 87–152.

BOZGA, MARIUS, DAWS, CONRADO, MALER, ODED, OLIVERO, ALFREDO, TRIPAKIS, STAVROS, AND YOVINE, SERGIO. 1998. Kronos: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, Number 1427 in Lecture Notes in Computer Science. Springer–Verlag, 546–550.

D'ARGENIO, P.R., KATOEN, J.-P., RUYS, T.C., AND TRETMANS, J. 1997. The bounded retransmission protocol must be on time! In *Proc. of the* 3rd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Number 1217 in Lecture Notes in Computer Science. Springer–Verlag, 416–431.

DAWS, CONRADO AND TRIPAKIS, STAVROS. 1998. Model Checking of Real-Time Reachability Properties Using Abstractions. In *Proc. of the* 4th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Number 1384 in Lecture Notes in Computer Science. Springer–Verlag, 313–329.

ERICSSON, CHRISTER, WALL, ANDERS, AND YI, WANG. 1999. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of the* 6th *International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press.

FERSMAN, ELENA, MOKRUSHIN, LEONID, PETTERSSON, PAUL, AND YI, WANG. 2003. Schedulability Analysis Using Two Clocks. To appear in *Proc. of the* 9th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

FERSMAN, ELENA, PETTERSSON, PAUL, AND YI, WANG. 2002. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In *Tools and Algorithms for the Construction and Analysis of Systems*, Volume 2280 of *Lecture Notes in Computer Science*. Springer–Verlag, 67ff.

HAREL, DAVID AND NAAMAD, AMNON. 1996. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology 5*, 4 (October), 293–333.

HÜTTEL, H. AND LARSEN, K. G. 1989. The Use of Static Constructs in a Modal Process Logic. In *Logic at Botik'89.*, Number 363 in Lecture Notes in Computer Science.

LARSEN, KIM G. PETTERSSON, PAUL, AND YI, WANG. 1995. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the* 16th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 76–87.

LARSEN, KIM G. PETTERSSON, PAUL, AND YI, WANG. 1997. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer 1*, 1–2 (Oct.), 134–152.

LEGOS-TEAM. 2002. The LegOS Homepage. http://legos.sourceforge.net.

LEWERENTZ, CLAUS AND LINDNER, THOMAS, EDITORS. 1995. *Formal Development of Reactive Systems: Case Study Production Cell*, Number 891 in Lecture Notes in Computer Science. Springer–Verlag.

LILIUS, JOHAN AND PALTOR, IVAN PORRES. 1999. Formalising UML state machines for model checking. In *In Proceedings of UML'99*, Volume 1723. Springer–Verlag, Berlin, 430–445.

LINDAHL, MAGNUS, PETTERSSON, PAUL, AND YI, WANG. 1998. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the* 4th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Number 1384 in Lecture Notes in Computer Science. Springer–Verlag, 281–297.

STAUNER, THOMAS, MLLER, OLAF, AND FUCHS, MAX. 1997. Using HyTech to Verify an Automotive Control System. In *Proc. Hybrid and Real-Time Systems, Grenoble, March 26-28, 1997*. Technische Universität München.

YI, WANG. 1991. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Science, Chalmers University of Technology.

Appendix A. The Analysis Model

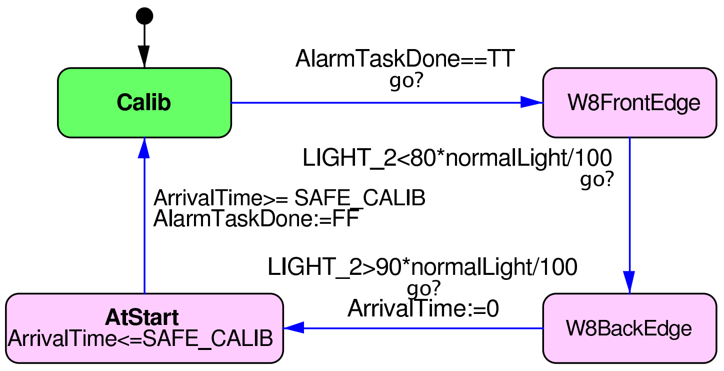


Fig. 8: The automaton Alarm.

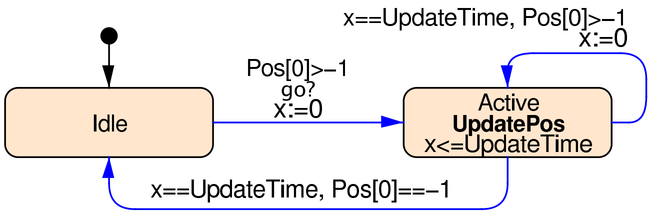


Fig. 9: The automaton BeltPosition.

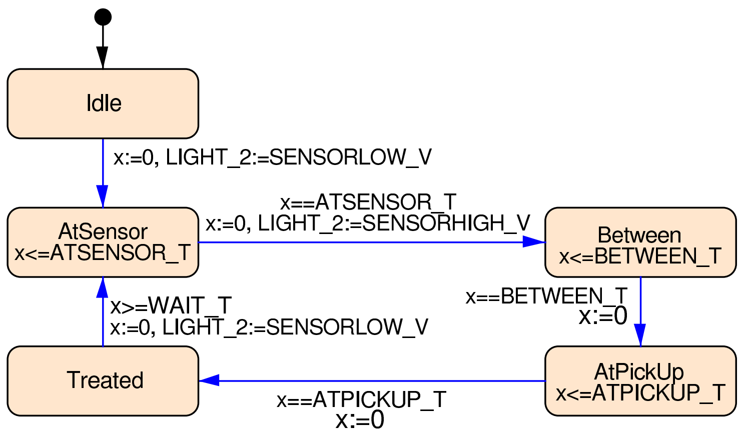


Fig. 10: The automaton Brick.

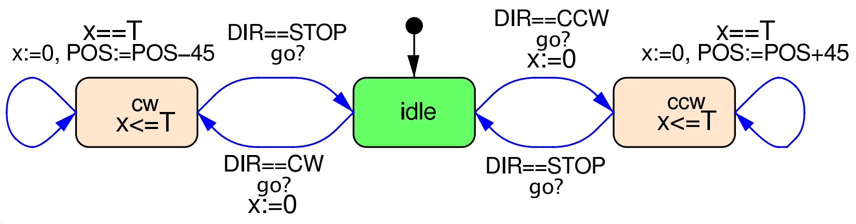


Fig. 11: The automaton Robot.

TABLE II: Task parameters used for analysis and synthesis.

Name	**C**	**D**	**P**	**Description and Interface**
Alarm				

AtStart	4	10	12	Append value to Pos.
				`Pos[numBricks]:=BELT_LENGTH,`
				`numBricks:=numBricks+1`

Calib	4	10	11	Read a calibrated value for background light.
				`AlarmTaskDone:=TT`
				`normalLight:=100`

BeltPositions				
UpdatePos	5	17	5	Update values in Pos. $\forall i$ `Pos[i]:=(Pos[i]>=0?Pos[i]-1:-1)`
RobotControl				
DropA	4	16	2	De-activate magnet on arm A. `TaskDone:=TT`
DropB	4	13	1	De-activate magnet on arm B. `TaskDone:=TT`

PickUpA	4	10	4	Activate magnet A. Remove value from Pos.
				$\forall i$ `Pos[i]:=Pos[i+1],`
				`Pos[BRICKS-1]:=-1,`
				`numBricks:=numBricks-1,`
				`TaskDone:=TT,`

PickUpB	4	17	3	Activate magnet on arm B. `TaskDone:=TT`
MoveTo				
MvRight	4	10	8	Start motor to rotate robot rightwards. `RobotAngle:=ROTATION_1`
MvLeft	4	10	7	Start motor to rotate robot leftwards. `ROB_DIR:=CW`
RdAngSen	4	16	9	Read rotation sensor and convert to degrees. `RobotAngle:=ROTATION_1`
StopRobot	4	10	10	Stop rotation motor. `ROB_DIR:=STOP`

Appendix A.1 Query File

```
/*
Robot can become ready to pick up a brick from the feedbelt.
*/
E<>( RobotControl.W8AtFB and Pos[0]==0)

/*
Scheduler can execute task to pick up a brick from the feed belt.
*/
E<>( SCHEDULER.RUN_PickUpA )

/*
Robot arm B can reach the position of the press.
*/
E<>( RobotControl.AtPressB )

/*
Robot arm A can reach the position of the press.
*/
E<>( RobotControl.AtPressA )

/*
Scheduler can execute task (released by robot controller) to drop brick
from arm A to the press.
*/
E<>( SCHEDULER.RUN_DropA )

/*
Robot can never reach a state where it is waiting for press to complete,
with arm B at the position of the press. Thus, location W8P1 is dead-code
in the robot controller model.
*/
A[ ]not( RobotControl.W8P1 )

/*
Robot can never reach a state where it is waiting for press to become
ready (with arm A at the position of the press). Thus, location W8P2 is
dead-code in the robot controller model.
*/
A[ ]not( RobotControl.W8P2 )

/*
Scheduler can execute task (released by robot controller) to picked up
brick from press with arm B.
*/
E<>( SCHEDULER.RUN_PickUpB )

/*
Scheduler can execute task (released by robot controller) which drop brick
from arm B on the deposit belt.
*/
E<>( SCHEDULER.RUN_DropB )

/*
The robot controller is never waiting unnecessarily at the pick-up
```

```
position of the feed belt. If it is there, it is because a brick has
been detected on the feed belt (but perhaps not yet arrived to the
pick-up position).
*/
A[ ]( RobotControl.W8AtFB imply Pos[0]>=0 )

/*
When the controller is waiting for a plate there should be no plate
on arm A.
*/
A[ ] not( RobotControl.AtW8 and RobotControl.PlateOnA==TT)

/*
Whenever task PickUpA is executing, a brick is at the pick-up position
of the belt.
*/
A[ ]( SCHEDULER.RUN_PickUpA imply Brick.AtPickUp )

/*
The pos of robot is always in the interval [FB_POS,DB_POS].
*/
A[ ]( ROTATION_1>=FB_POS and ROTATION_1<=DB_POS )

/*
The variables sharedPos[0] and numBricks are correctly updated.
*/
A[ ] not( Pos[0]==-1 and numBricks>=1 )

/*
Boundedness Analysis: The nr of simultaneously released task is bounded
to 3.
*/
A[ ]( (SCHEDULER.n0 + SCHEDULER.n1 + SCHEDULER.n2 +
       SCHEDULER.n3 + SCHEDULER.n4 + SCHEDULER.n5 +
       SCHEDULER.n6 + SCHEDULER.n7 + SCHEDULER.n8 +
       SCHEDULER.n9 + SCHEDULER.n10 ) <= 3 )

/*
Schedulability Analysis: all tasks are always guaranteed to meet their
deadlines.
*/
A[ ]not( SCHEDULER.error )
```

Appendix B. Generated Code

Appendix B.1 Production Cell Code

```
/********************************************
 * This code was AUTOMATICALLY generated by:
 * TimesTool, Version 0.99, May 2002
 * for: tobiasa
 * on: Fri May 17 17:31:26 CEST 2002
 * with Target: LegOS
 * with Synchronisation: Synchronous
 *
 * -> EDIT WITH CARE! <-
 ******************************************/

#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>
#include <sys/tm.h>
#include <conio.h>
#include <dsound.h>

#include "legos_definitions.h"

// Defines & declarations for tasks:
#define BRICKS 8
#define BELT_LENGTH 8 //measured in time

int normalLight;
int numBricks = 0;

// Task identifiers (tid)
#define tid_offset 200
#define tid_AtStart tid_offset+0
#define tid_Calib tid_offset+1
#define tid_DropA tid_offset+2
#define tid_DropB tid_offset+3
#define tid_MvLeft tid_offset+4
#define tid_MvRight tid_offset+5
#define tid_PickUpA tid_offset+6
#define tid_PickUpB tid_offset+7
#define tid_RdAngSen tid_offset+8
#define tid_StopRobot tid_offset+9
#define tid_UpdPos tid_offset+10
#define tid_NOP tid_offset+11
#define NB_TASK 11

// Transition ids
#define T1 0
#define T2 1
#define T3 2
#define T4 3
#define T5 4
#define T6 5
#define T7 6
#define T8 7
#define T9 8
#define T10 9
#define T11 10
#define T12 11
#define T13 12
#define T14 13
#define T15 14
#define T16 15
#define T17 16
#define T18 17
#define T19 18
#define T20 19
#define T21 20
#define T22 21
#define T23 22
#define T24 23
#define T25 24
#define T26 25
#define T27 26
#define T28 27
#define T29 28
#define T30 29
#define T31 30
#define T32 31
#define T33 32
#define T34 33
#define T35 34
#define T36 35
#define T37 36
#define NB_TRANS 37

// Define location offsets
#define RobCtrl_AtW8 0
#define RobCtrl_AtPressB 3
#define RobCtrl_Mv2PrB 7
#define RobCtrl_S0 9
#define RobCtrl_W8AtFB 11
#define RobCtrl_Mv2FB 13
#define RobCtrl_S1 15
#define RobCtrl_MvToPrA 17
#define RobCtrl_AtPressA 19
#define RobCtrl_S2 22
#define RobCtrl_AtPress 24
#define RobCtrl_Mv2DB 27
#define RobCtrl_Mv2W8 29
#define RobCtrl_S3 31
#define RobCtrl_AtDB 33
#define RobCtrl_MvTwPr 35
#define RobCtrl_Branch 37
#define MoveTo_MR 40
#define MoveTo_Read 42
#define MoveTo_Entry 46
#define MoveTo_SR 48
#define MoveTo_ML 50
#define Enter_S0 52
#define Enter_W8FrontEdge 54
#define Enter_S1 56
#define Enter_W8BackEdge 58
#define UpdPos_Idle 60
#define UpdPos_Active 62

#define NB_LOC 28

char release_list[NB_TASK]=
   {0,1,0,0,0,0,0,0,0,0,0};

// Constant values
#define FB_POS 0
#define WAIT_POS 45
#define PRESS_B_POS 0
#define PRESS_A_POS 90
#define DB_POS 90
#define PREP_T 200
#define PRESS_T 250
#define TT 1
#define FF 0
#define RobCtrl_TM2PRESS 1000
#define RobCtrl_GOPRESS_T -750
#define Enter_SAFE_TIME 4000
#define Enter_DL_CALIB 0
#define Enter_SAFE_CALIB 4000
#define UpdPos_UpdTime 333

// Clock variables
time_t clock_RobCtrl_PressTime;
time_t clock_RobCtrl_PrepTime;
time_t clock_MoveTo_x;
time_t clock_Enter_ArrivalTime;
time_t clock_UpdPos_x;

// Integer variables
int RobotAngle=45;
int Pos[5]={-1,-1,-1,-1,-1};
int AlarmTaskDone=0;
int TaskDone=1;
int Goal=0;
int RobCtrl_PlateInPress=0;
int RobCtrl_PlateOnA=0;
int RobCtrl_PlateOnB=0;

void Prodcell_init() {

   // Deactivate rotation sensor
   ds_passive(&SENSOR_1);

   // Sleep, needed to get correct rotation
   // sensor readings
   msleep(100);

   // Activate rotation sensor
   ds_active(&SENSOR_1);
   ds_rotation_off(&SENSOR_1);
   ds_rotation_set(&SENSOR_1,0);
   ds_rotation_on(&SENSOR_1);
}

// Define channel names
#define startS 0
```

```
#define stopS 11
#define startR 14
#define stopR 16
#define recv_channels startR

// Is a channel id a sending channel?
#define IS_SEND( c ) (c < recv_channels )

// Channel usage
unsigned char chanusage[23] = {
  T2,T5,T6,T8,T12,T13,T17,T18,T22,T23,NB_TRANS
  ,T27,T30,NB_TRANS
  ,T24,NB_TRANS
  ,T1,T4,T9,T14,T15,T21,NB_TRANS
};

// Evaluate guard on transition g
int eval_guard(char g) {
  switch(g) {
  case T2: return (TaskDone==TT);
  case T3: return (Pos[0]==0);
  case T5: return (Pos[0]>-1);
  case T6: return (RobCtrl_PlateInPress==TT
                   && Pos[0]==-1);
  case T7: return (TaskDone==TT);
  case T8: return (RobCtrl_PlateInPress==FF
                   && Pos[0]==-1);
  case T10: return (TaskDone==TT);
  case T11: return (RobCtrl_PlateOnA==FF);
  case T12: return (RobCtrl_PlateOnB==TT);
  case T13: return (RobCtrl_PlateOnB==FF);
  case T14: return (RobCtrl_PlateOnB==FF);
  case T16: return (TaskDone==TT);
  case T18: return (Pos[0]>-1 &&
                    RobCtrl_PlateOnA==FF);
  case T19: return (Pos[0]==-1 &&
                    clock(RobCtrl_PressTime)
                    > PRESS_T &&
                    RobCtrl_PlateInPress==TT);
  case T20: return (clock(RobCtrl_PrepTime)>
                    PREP_T &&
                    RobCtrl_PlateOnA==TT);
  case T22: return (Pos[0]==-1);
  case T23: return (Pos[0]>-1 &&
                    RobCtrl_PlateOnA==FF);
  case T25: return (clock(MoveTo_x)==30 &&
                    RobotAngle<Goal);
  case T26: return (clock(MoveTo_x)==30 &&
                    RobotAngle>Goal);
  case T28: return (ROTATION_1==Goal);
  case T29: return (ROTATION_1==Goal);
  case T30: return (clock(MoveTo_x)==30 &&
                    RobotAngle==Goal);
  case T31: return (AlarmTaskDone==TT);
  case T32: return (clock(Enter_ArrivalTime)>=
                    Enter_SAFE_CALIB);
  case T33: return
(LIGHT_2<80*normalLight/100);
  case T34: return
(LIGHT_2>90*normalLight/100);
  case T35: return (Pos[0]>-1);
  case T36:
    return (clock(UpdPos_x)==UpdPos_UpdTime
            && Pos[0]>-1);
  case T37:
    return (clock(UpdPos_x)==UpdPos_UpdTime
            && Pos[0]==-1);
  case T1:
  case T4:
  case T9:
  case T15:
  case T17:
  case T21:
  case T24:
  case T27:
    return true;
  } /* case */
  return false;
}

// Perform assignments on transition a
void assign(char a) {
  switch(a) {
  case T2:
    RobCtrl_PlateOnA=TT;
    Goal=PRESS_B_POS; break;
  case T3:
    TaskDone=FF; break;
  case T5:
    Goal=FB_POS; break;
  case T6:
    Goal=PRESS_B_POS; break;
  case T7:
    RobCtrl_PlateInPress=FF;
    RobCtrl_PlateOnB=TT;
    reset(RobCtrl_PrepTime); break;
  case T8:
    Goal=PRESS_A_POS-45; break;
  case T10:
    RobCtrl_PlateInPress=TT;
    RobCtrl_PlateOnA=FF;
    reset(RobCtrl_PressTime); break;
  case T12:
    Goal=DB_POS; break;
  case T13:
    Goal=WAIT_POS; break;
  case T14:
    Goal=WAIT_POS; break;
  case T15:
    TaskDone=FF; break;
  case T16:
    RobCtrl_PlateOnB=FF; break;
  case T17:
    Goal=WAIT_POS; break;
  case T18:
    Goal=FB_POS; break;
  case T19:
    TaskDone=FF; break;
  case T20:
    TaskDone=FF; break;
  case T22:
    Goal=PRESS_A_POS; break;
  case T23:
    Goal=FB_POS; break;
  case T24:
    reset(MoveTo_x); break;
  case T32:
    AlarmTaskDone=FF; break;
  case T34:
    reset(Enter_ArrivalTime); break;
  case T35:
    reset(UpdPos_x); break;
  case T36:
    reset(UpdPos_x); break;
  }
}

trans_t trans[NB_TRANS] = {
  {0,RobCtrl_Mv2PrB,RobCtrl_AtPressB,stopS},
  {0,RobCtrl_S0,RobCtrl_Mv2PrB,startR},
  {0,RobCtrl_W8AtFB,RobCtrl_S0,-1},
  {0,RobCtrl_Mv2FB,RobCtrl_W8AtFB,stopS},
  {1,RobCtrl_AtW8,RobCtrl_Mv2FB,startR},
  {1,RobCtrl_AtW8,RobCtrl_Mv2PrB,startR},
  {0,RobCtrl_S1,RobCtrl_AtPressB,-1},
  {0,RobCtrl_AtPressB,RobCtrl_MvTwPr,startR},
  {0,RobCtrl_MvToPrA,RobCtrl_AtPressA,stopS},
  {0,RobCtrl_S2,RobCtrl_AtPressA,-1},
  {0,RobCtrl_AtPressA,RobCtrl_AtPress,-1},
  {0,RobCtrl_AtPress,RobCtrl_Mv2DB,startR},
  {0,RobCtrl_AtPress,RobCtrl_Mv2W8,startR},
  {0,RobCtrl_Mv2W8,RobCtrl_AtW8,stopS},
  {0,RobCtrl_Mv2DB,RobCtrl_S3,stopS},
  {0,RobCtrl_S3,RobCtrl_AtDB,-1},
  {0,RobCtrl_AtDB,RobCtrl_Mv2W8,startR},
  {0,RobCtrl_AtPressB,RobCtrl_Mv2FB,startR},
  {0,RobCtrl_AtPressB,RobCtrl_S1,-1},
  {0,RobCtrl_AtPressA,RobCtrl_S2,-1},
  {0,RobCtrl_MvTwPr,RobCtrl_Branch,stopS},
  {0,RobCtrl_Branch,RobCtrl_MvToPrA,startR},
  {0,RobCtrl_Branch,RobCtrl_Mv2FB,startR},
  {1,MoveTo_Entry,MoveTo_Read,startS},
  {0,MoveTo_Read,MoveTo_MR,-1},
  {0,MoveTo_Read,MoveTo_ML,-1},
  {0,MoveTo_SR,MoveTo_Entry,stopR},
  {0,MoveTo_MR,MoveTo_SR,-1},
  {0,MoveTo_ML,MoveTo_SR,-1},
  {0,MoveTo_Read,MoveTo_Entry,stopR},
  {1,Enter_S0,Enter_W8FrontEdge,-1},
  {0,Enter_S1,Enter_S0,-1},
  {0,Enter_W8FrontEdge,Enter_W8BackEdge,-1},
  {0,Enter_W8BackEdge,Enter_S1,-1},
  {1,UpdPos_Idle,UpdPos_Active,-1},
  {0,UpdPos_Active,UpdPos_Active,-1},
  {0,UpdPos_Active,UpdPos_Idle,-1}
};
```

```
unsigned char loc[NB_TRANS+NB_LOC] = {
  T5,T6,tid_NOP/*AtW8*/,
  T8,T18,T19,tid_NOP/*AtPressB*/,
  T1,tid_NOP/*Mv2PrB*/,
  T2,tid_PickUpA/*S0*/,
  T3,tid_NOP/*W8AtFB*/,
  T4,tid_NOP/*Mv2FB*/,
  T7,tid_PickUpB/*S1*/,
  T9,tid_NOP/*MvToPrA*/,
  T11,T20,tid_NOP/*AtPressA*/,
  T10,tid_DropA/*S2*/,
  T12,T13,tid_NOP/*AtPress*/,
  T15,tid_NOP/*Mv2DB*/,
  T14,tid_NOP/*Mv2W8*/,
  T16,tid_DropB/*S3*/,
  T17,tid_NOP/*AtDB*/,
  T21,tid_NOP/*MvTwPr*/,
  T22,T23,tid_NOP/*Branch*/,
  T28,tid_MvRight/*MR*/,
  T25,T26,T30,tid_RdAngSen/*Read*/,
  T24,tid_NOP/*Entry*/,
  T27,tid_StopRobot/*SR*/,
  T29,tid_MvLeft/*ML*/,
  T31,tid_Calib/*S0*/,
  T33,tid_NOP/*W8FrontEdge*/,
  T32,tid_AtStart/*S1*/,
  T34,tid_NOP/*W8BackEdge*/,
  T35,tid_NOP/*Idle*/,
  T36,T37,tid_UpdPos/*Active*/
};

#include "legos_kernel.h"

int AtStart() {
  TASK_BEGIN(AtStart)

  // Plate at the start
  // Insert "brick" at first non used position
  Pos[numBricks]=BELT_LENGTH;
  numBricks++;

  TASK_END
}

int Calib() {
  TASK_BEGIN(Calib)

  // Take the average light value over 500 ms
#define SAMPLES 10
  int i;
  int readsum = 0;
  extern int normalLight;
  for( i=0; i < SAMPLES; i++) {
    readsum += LIGHT_2;
    msleep(50);
  }

  // Update shared variables
  normalLight=readsum/SAMPLES;
  AlarmTaskDone=TT;

  TASK_END
}

int DropA() {
  TASK_BEGIN(DropA)

  // Deactivate the magnet on arm A
  motor_a_speed(off);

  // Update shared variables
  TaskDone=TT;

  TASK_END
}

int DropB() {
  TASK_BEGIN(DropB)

  // Deactivate the magnet on arm B
    motor_b_speed(off);

  // Update shared variables
  TaskDone=TT;

  TASK_END
}
```

```
int MvLeft() {
  TASK_BEGIN(MvLeft)

  // Start robot motor moving left
  motor_c_dir(fwd);
  motor_c_speed(100);

  TASK_END
}

int MvRight() {
  TASK_BEGIN(MvRight)

  // Start robot motor moving right
  motor_c_dir(rev);
  motor_c_speed(100);

  TASK_END
}

int PickUpA() {
  TASK_BEGIN(PickUpA)

  int i, temp;

  // Activate the magnet on arm A
  motor_a_speed(MAX_SPEED);

  // Shift the values in the pos array.
  // Update the first (shared) element last.
  temp = Pos[1];

  for( i=1; i<numBricks; i++) {
    Pos[i] = Pos[i+1];
  }
  numBricks--;

  Pos[BRICKS-1] = -1;

  // Update shared variables
  Pos[0]=temp;
  TaskDone=true;

  TASK_END
}

int PickUpB() {
  TASK_BEGIN(PickUpB)

  // Activate the magnet on arm B
  motor_b_speed(MAX_SPEED);

  // Update shared variables
  TaskDone=true;

  TASK_END
}

int RdAngSen() {
  TASK_BEGIN(RdAngSen)

  // The rotation sensor is update every
  // 1/16 th of a turn. Cog-wheel mounted
  // between the rotation axis and the sensor
  // give a precision of 360 degees.

  // Update shared variables
  RobotAngle = -ROTATION_1;

  TASK_END
}

int StopRobot() {
  TASK_BEGIN(StopRobot)

  // Stop motor moving robot.
  motor_c_dir(brake);

  TASK_END
}

int UpdPos() {
  TASK_BEGIN(UpdPos)

  // Update the brick positions in the position array.
  int i;
  for (i=1; i<BRICKS; i++) {
    Pos[i] = ((Pos[i] >= 0) ? (Pos[i]-1) : -1) ;
  }

  // Update shared variables
  Pos[0] = ((Pos[0] >= 0) ? (Pos[0]-1) : -1) ;

  TASK_END
}

int main(int argc, char **argv) {

  Prodcell_init();

  execi( &AtStart, 0, NULL, 12, SMALL_STACK);
  execi( &Calib, 0, NULL, 11, SMALL_STACK);
  execi( &DropA, 0, NULL, 2, SMALL_STACK);
  execi( &DropB, 0, NULL, 1, SMALL_STACK);
  execi( &MvLeft, 0, NULL, 7, SMALL_STACK);
  execi( &MvRight, 0, NULL, 8, SMALL_STACK);
  execi( &PickUpA, 0, NULL, 4, SMALL_STACK);
  execi( &PickUpB, 0, NULL, 3, SMALL_STACK);
  execi( &RdAngSen, 0, NULL, 9, SMALL_STACK);
  execi( &StopRobot, 0, NULL, 10, SMALL_STACK);
  execi( &UpdPos, 0, NULL, 5, SMALL_STACK);

  // Reset clocks
  reset(RobCtrl_PressTime);
  reset(RobCtrl_PrepTime);
  reset(MoveTo_x);
  reset(Enter_ArrivalTime);
  reset(UpdPos_x);

  execi( &controller, 0, NULL,
         PRIO_HIGHEST, SMALL_STACK);
  return 0;
}
```

Appendix B.2 Run-time system

```
// legos_kernel.h: The kernel functions that
// executes the controller.


// Deterimines if it is possible to
// synchronise on a channel.
//
// parameter: sync, channel id, i.e. index
// into the chanusage array.
//
// return: 0, if synchronisation is not
// possible, transition id of the
// complementary transition if it is.
int check_synch(unsigned char sync) {
  while( chanusage[sync] < NB_TRANS ) {
    if( trans[chanusage[sync]].active &&
          eval_guard( chanusage[sync] ) )
      return chanusage[sync];
    sync++;
  }
  return false;
}

// Update the list of active transitions,
// and release task of target location.
//
// parameter: trn, the transition taken.
void clear_and_set (unsigned char trn) {
  int jj;
  // Clear outgoing transition from source
  jj=trans[trn].from;
  do {
    trans[loc[jj]].active=0;
  } while(loc[++jj]<tid_offset);
  // Set outgoing transition from target
  jj=trans[trn].to;
  do {
    trans[loc[jj]].active=1;
  } while(loc[++jj]<tid_offset);
  // Release task of target location
  release_list[loc[jj]-tid_offset]++;
}

// Check if an active transition is enabled,
// and if so take it. Will continue until a
// stable state (no more enabled
// transitions) is reached.
//
// parameter: data, unused (should be null).
// return: false, when in stable state
wakeup_t check_trans( wakeup_t data ) {
  int trn, compl_trans;
  for(trn=0; trn<NB_TRANS; trn++) {
    if( trans[trn].active ) {
      if( eval_guard(trn) ) {
        if( trans[trn].sync > -1 ) {
          if( (compl_trans =
                check_synch( trans[trn].
                                sync ))){
            if( IS_SEND(trans[trn].sync) ) {
              assign( trn );
              assign( compl_trans );
            } else {
              assign( compl_trans );
              assign( trn );
            }
            clear_and_set( trn );
            clear_and_set( compl_trans );
            trn=-1;
          }
        } else {
          assign( trn );
          clear_and_set( trn );
          trn=-1;
  } } } }
  return false;
}

// Wake-up function for threads (tasks).
// parameter: data, unused (should be null)
wakeup_t task_release(wakeup_t data) {
    return release_list[data];
}

// Thread body for automata controller
// return: 0 (allways)
int controller() {
  wait_event( &check_trans, 0);
  return 0;
}
```

Appendix B.3 Header file

```
// legos_definitions.h: Define types and
// macros used by generated code and kernel.

typedef struct trans_s{
  char active;
  char from;
  char to;
  signed char sync;
} trans_t;

#ifndef true
#define true (1==1)
#endif

#ifndef false
#define false (0==1)
#endif

#define SMALL_STACK 128

// Macros thar read and reset clocks.
#define clock(c) (sys_time - clock_##c)
#define reset(c) clock_##c = sys_time

#define TASK_BEGIN( taskname ) while( true ) {\
    wait_event( &task_release, tid_##taskname -
tid_offset );\
    release_list[ tid_##taskname - tid_offset]−−; {

#define TASK_END }}\
    return 0;\

#define CONTROLLER( AUT ) int
AUT##_controller() {\
    while( true ) { \
        wait_event(& AUT##_events, 0);\
    }\
}
```