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.

May 29, 2002

Abstract

We present a framework for development of real-time embedded systems based on the recently presented model of timed automata extended with real-time tasks. It has been shown previously that design problems such as reachability and schedulability are decidable for the model of timed automata with tasks. In this paper we describe how to automatically synthesise executable code with predictable timing behaviour, which is guaranteed to meet constraints (timing and other) imposed on the design model. To demonstrate the applicability of the framework, implemented in the TIMES tool, we present a case-study of a well known production cell, built in LEGO® and controlled by a Hitachi H8 based LEGO® Mindstorm control brick.

1 Introduction

A key facility of many commercial tools for development of embedded software is to automatically synthesize fully executable code (i.e. code generation) from design models. However, few of the existing tools are capable of producing software with *predictable* timing behaviors, i.e. code which is known a priori to satisfy given timing constraints. In fact, most of the commercial tools trust the software developer to resolve the timing issues that arise when the generated code should be executed on a specific target platform, such as e.g. analysing whether all tasks will meet their deadlines.

In contrast, research tools for real-time systems such as UPPAAL [LPY97] and KRONOS [BDM+98] are often dedicated to analysis and abstraction of formal high-level design description. This has proven useful for finding errors and checking correctness properties in many case studies, e.g. [BGK+96], [LPY98], [SMF97] and [DKRT97]. However, it provides little or no support for producing the actual program code to be executed in the final implementation.

In this work, we combine results and ideas from model-checking, scheduling, and synchronous programming to develop a framework for the development of real-time embedded systems. The goal is to support, on one hand formal specification, validation, analysis of design, and on the other hand code synthesis which guarantees that certain timing constraints are met when executed on the target system.

As a specification and design language, we use the recently suggested model of extended timed automata with real-time tasks [FPY02]. In this model, based on Alur and Dill's model of timed automata [AD94], a node of an automaton is associated with a task (or several tasks) that is assumed to be an executable program with given parameters, such as the worst case execution time, deadline, fixed priority etc. Intuitively, whenever an automaton reaches a control node 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 scheduling problem of timed automata with tasks is decidable both for non-preemptive [EWY99] and preemptive [FPY02] 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 model of timed system described as timed automata with tasks, it can be checked whether it is schedulable prior to its implementation.

In this paper, we propose a way of synthesising code with predictable timing behaviour from timed automata with tasks. Inspired by the design philosophy of synchronous languages e.g. Esterel [BG92], we assume that the target real-time operating system and hardware guarantees the *synchrony hypothesis*, i.e. the run-time of the operating system is neglectable compared with the execution times and deadlines of the application tasks. Based on this assumption, we synthesize from the control structure of a schedulable automaton code which is guaranteed to meet all the imposed constraints (timed and others) when executed.

The discrete transitions (i.e the control structure) of the automaton and its associated tasks are implemented using a small set of (common) system calls assuming that light-weight threads are provided by the operating system. With such few assumptions, the code synthesize should be applicable to a variety of different target platforms. So far, we have implemented a prototype that synthesizes C-code for the legOS operating system that runs on the LEGO® Mindstorm control brick, which is equipped with an 8-bit Hitachi micro-controller. This corresponds to the hardware used in the kind of embedded systems we are targeting.

The code synthesis and analysis have been implemented in the TIMES tool [AFM+02]. To evaluate our framework and the tool, we have applied it in a case-study of a well known production cell, built in LEGO® and controlled by LEGO® Mindstorm control bricks. The control program of the most central

component, the two-armed robot, is designed and analysed using the TIMES tool. In addition, we generate executable C-code for the legOS operating system. The generated code has been compiled, downloaded, and executed on LEGO® Mindstorm bricks.

The rest of this article is organised as follows: In the next section we present the model of timed automata extended with tasks and discuss how to analyse designs described in the model. In section 3 we describe a deterministic subset of the semantics of timed automata with tasks. Section 4 describes the implementation of the deterministic semantics on a small generic embedded operation system. In section 5 we describe the production cell case-study, and in section 6 we describe and demonstrate the analysis and code synthesis of the case study. Finally, we conclude the article in section 7

2 The Design Language

In [EWY99] and [FPY02], an extended version of timed automata with real time tasks is presented. The idea is to annotate each location of an automaton with a task (an executable program with computing time and deadline), that will be triggered by transitions leading to the location. 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. However a severe restriction in this model is that there is no interaction between an automaton and the annotated tasks.

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 *worst case 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

[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.

the value of $\mathcal{E}$ is returned when the task is finished). Thus, each task P is characterized as a triple denoted (C, D, A) with $C \leq D$, where C is the worst case 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 worst case execution time, relative deadline and set of assignments of P respectively.

Timed Automata. Assume a finite set of alphabets *Act* for actions and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b etc to range over *Act* 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. 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 $\mathcal{A}$.

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*
  - *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations 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.

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

To handle concurrency and synchronization, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [LPY95]) using the notion of synchronization function [HL89]. 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 synchronizable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronization function [HL89] , 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: $[P_1(c_0, d_0), P_2(c_1, d_1)...P_n(c_n, d_n)]$ 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 with 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 $t \in \mathcal{R}_{\geq 0}$, 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 $\mathcal{E}$ 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 *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:*

- $(l, \sigma, q) \xrightarrow{a}_{\mathsf{Sch}} (m, \sigma[r], \mathsf{Sch}(M(m) :: q))$ *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*.*

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 [EWY99] and preemptive scheduling policies [FPY02]. 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 [FPY02].

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 to check *safety properties*, i.e. that a certain undesired situation (represented by a

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

set of states) is guaranteed to never be reached, and dually *invariant properties*, i.e. that a property is guaranteed to hold invariantly. To determine reachability of a states of timed automata with tasks (with variables shared between tasks and control automata) the task queue must be considered. In general the task queue many be unbounded, in the sense that states can be reached in which the task queue holds an infinite number of tasks. Formally we define boundedness of the ready queue in the following way:

Definition 4 (Boundedness) *A timed automaton A with initial location (l_0, σ_0, q_0) is bounded by the queue length n iff for all reachable states (l, σ, q), $n \geq |q|$. If $n = \max\{|q'| : (l_0, \sigma_0, q_0)(\longrightarrow)^*(l', \sigma', q')\}$, n is the least upper bound on the queue length of A. If $n = \infty$, the queue length of A is unbounded.*

Note that checking the boundedness property for a given queue length and timed automata with tasks is an instance of reachability (where the queue length is taken into consideration) whereas the problem of finding the least upper bound on the queue length in general is a synthesis problem.

Similar to boundedness, it should be checked prior to the implementation (or code synthesis) that all tasks are guaranteed to always meet their deadlines:

Definition 5 (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 consisting of 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 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.

3 Synthesis

We again consider $S^{Design} = C \parallel E_1 \parallel E_2 \parallel \ldots \parallel E_m$ as a design model, where C is a model of the controller to be developed and $E_1, ..., E_m$ are models of the environment.

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 how 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 synchronize 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 $\mathsf{Pr} : E \mapsto \mathcal{Z}$ be a function assigning a unique priority to each edge, which defines an order that the guards of transitions are evaluated and the presence of actions are checked. If several transitions are enabled in a state, they should be taken in the priority order.
- Time non-determinism is resolved by implementing the so-called maximal-progress assumption [Yi91]. 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 [HN96] and the *run-to-completion* step of UML statecharts [LP99].

Now we have a refined (deterministic) version of the operational semantics for the extended timed automata:

Definition 6 *(Deterministic semantics) Let $A = \langle N, l_0, E, I, M \rangle$ be an extended timed automaton. Given a scheduling strategy* Sch*, the deterministic semantics is defined as a labelled transition system defined by the rules:*

- $(l, \sigma, q) \xrightarrow{a} (m, \sigma[r], \mathsf{Sch}(M(m) :: q))$ *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* $\mathsf{Enabled}(e, (l, \sigma, q))$
- $(l, \sigma, q) \xrightarrow{t} (l, \sigma + t, \mathsf{Run}(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C(\mathsf{Hd}(q)) > t$ *and for all* $d < t$ $\neg\mathsf{Enabled}(e, (l, \sigma + d, q))$
- $(l, \sigma, q) \xrightarrow{t} (l, (\sigma[A(\mathsf{Hd}(q))]) + t, \mathsf{Run}(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C(\mathsf{Hd}(q)) = t$ *and for all* $d < t$ $\neg\mathsf{Enabled}(e, (l, \sigma + d, q))$

Clearly the behaviour of the controller according to the refined semantics is deterministic. Safety properties in the design model are preserved since the behaviour, denoted $\mathcal{L}^d(C)$, defined by deterministic semantics is included in the

behaviour, denoted $\mathcal{L}(C)$, defined by the operational semantics, i.e. $\mathcal{L}^d(C) \subseteq \mathcal{L}(C)$. It is easy to see that whenever the controller may take a transition according to the refined semantics it can always take a transition according to the operational semantics.

When implementing the deterministic semantics the existence of stable states must be guaranteed. In other words, no execution of the system may contain an infinite sequence of action transitions, as by rule 1 in Definition 6 tasks are only executed when a delay transition is taken. In general the absence of such infinite sequences of action transition is not guaranteed by the deterministic semantics, therefor we impose a simple syntactic requirement. We require, for all automata, that at least one location in any loop has an associated task. With this requirement schedulability of the system will also guarantee the absence of infinite sequences of action transitions since the existence of such a sequence would imply that the system is non-schedulable.

4 Execution of the Deterministic Semantics

We now present how to perform code-generation for timed automata with tasks. The generated code implements the deterministic semantics on a generic small embedded operating system. As input the generator takes a schedulable design model with bounded task-queue length and a set of files containing the task code. We require schedulability of the design model according to the policy of the target OS. This allows us to reduce the complexity of the generated code as we e.g. do not need to monitor execution times of tasks since deadlines are guaranteed never to be violated. Boundedness of the queue length limit the memory needed by the code.

We assume that the OS provides the following general features:

- threads with unique priorities,
- preemptive or non-preemptive scheduling of threads,
- a scheduling policy for which analysis is possible (FPS, RMS, EDF, et.c.),
- interrupt handling routines or `wait_until()`.

Unique priorities of threads is needed since we let each task type execute in its own thread, and task types have unique priorities. To guarantee the behaviour the target OS must use a scheduling policy for which analysis is possible. To encode the control automata, especially event handling, interrupt service routine must be available. As an alternative polling could be used. Then a system call that suspend a thread until a specified time-point (`wait_until()`) is needed.

Our intended target for the generated code is a single embedded processor, but the intuition behind the execution of the code is more easily illustrated by assuming two processors as illustrated in Figure 1. The controller and the task code execute on two separate logical processors, the *control processor* and the *task processor* respectively. The interaction between them is via 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.

Figure 1: The logical view of the executing system, with a *control processor* handling events from the environment and communication with the executing *task processor* only through the task queue and shared variables.

We realise the execution model described above on a single processor by executing the code of the controller automata as a separate thread with the highest possible priority. The only thing this thread does is to encode the control-structure.

4.1 Encoding and Executing the Controller Automata

In Figure 2 we show the data structures that encode the control automata and in Table 1 the procedure that uses them. The set of *active transitions* in a state $\mathsf{s} = (l, \sigma, q)$ is all transitions $\{\, m \xrightarrow{g,a,r} m' \mid m = l \,\}$, i.e. all transitions leaving from the current location.

In Figure 2 ACTIVE is a list of references to transitions sorted in priority order. The transitions are stored in priority order in a table with five fields: *guard*, *assign*, $from$, *to*, *sync*. The *guard* and *assign* fields are references to code for evaluating the guard and performing the assignments of the transition respectively. The fields $from$ and *to* are references into a list of locations. For each location the outgoing transitions and any associated task is stored in a separate table. The last field in the transition table (*sync*) is either empty or a reference to a list of transitions (in priority order) that has the complementary synchronisation label.

The datastructure ACTIVE is the list of active transitions, which is managed by the procedure shown in Table 1. The procedure is executed by the controller, either periodically or, if the target OS support interrupts, when an external event has occurred. During execution no new external events are processed and timers are not updated. I.e. the whole procedure is executed in a critical region where no interrupts are allowed. Note that tasks can not update any shared variables while the procedure is executing since they execute with lower priority.

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

Table 1: Algorithm, in pseudo-code, that execute the encoded controller.

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

Algorithm:
`START:`
for each $l \xrightarrow{g,a,r} l'$ in ACTIVE **do**
if σ satisfies g **then**
if exists $m \xrightarrow{g',\overline{a},r'} m'$ in ACTIVE such that σ satisfies g' **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 ACTIVE
add $\{n \longrightarrow n' | n = l' \vee n = m'\}$ to ACTIVE
ACTIVE := $sort(\mathsf{ACTIVE}, \mathsf{Pr})$
$q := \mathsf{Sch}(M(l') :: M(m') :: q)$
goto `START`
fi
else
$\sigma := \sigma[r]$
remove $\{n \longrightarrow n' | n = l\}$ from ACTIVE
add $\{n \longrightarrow n' | n = l'\}$ to ACTIVE
ACTIVE := $sort(\mathsf{ACTIVE}, \mathsf{Pr})$
$q := \mathsf{Sch}(M(l') :: q)$
goto `START`
fi
od

By $\overline{a}$ we denote the complementary synchronisation label of a, i.e. *sync*! for an *sync*? and vice versa. Let *sort* be a function that takes two parameters: a list of transitions and a function that assigns unique priorities to transitions, and outputs a sorted list. Initially (before execution starts) the sorted list of active transitions consists of all transitions leaving from initial locations.

The procedure walks through the list and evaluates corresponding guards. If a guard is satisfied there are two distinct cases:

- no synchronisation – then perform assignments of the transition, update the list of active transitions, and release the tasks associated with the target location.
- synchronisation (i.e. the transition is labeled with a channel) – then search (in priority order) for a transition with the complementary channel whose guard is true. If such a transition is found among the active transitions perform the assignments of both transitions (the sending transition first), update the set of active transitions and release tasks associated with the target locations.

When a transition is taken the procedure returns to the beginning and walks through the list one more time to find if another transitions could be taken. The procedure implements a run-to-completion step, only when a stable state is reached (i.e. no transitions are enabled) will the procedure return and the controller thread is be suspended which allow the task threads to execute.

4.2 Handling Tasks and Variables

For scheduling of task threads and management of the ready queue the generated code rely on the target OS. Recall, that we assume boundedness (see Definition 4) of the queue length which guarantees that the queue will never overflow. This means that the memory allocated for the task queue can be fixed at compile time and no exception handling for queue overflow is needed.

Data variables in the design model are mapped to global integer variables in the generated code. This allows them to be used and updated by the task code.

To encode clocks let *sc* be a global system clock. For each clock x in the timed automata, let x^r be an integer variable holding the system time of the last reset of the clock. Then the value of the clock is $(sc - x^r)$ and a reset is performed as $x^r := sc$.

Stopping the clocks while executing the controller is achieved by taking a copy of the system time before starting the procedure and use the copy to calculate the clock values during execution.

4.3 Deterministic Behaviour of Implementation

We claim that the described implementation follows the deterministic semantics and point out how the refinements introduced to resolve non-determinism are implemented in the generated code.

- **Priority** Recall that all transitions are assigned globally unique priorities. The controller code maintains a sorted list (in priority order) of the *active transitions*, i.e. the transitions from the currently active locations. The controller thread evaluates the guards of the transitions in the list 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 chosen. The order of the list is maintained when replacing the outgoing transitions of the source location with those of the target location by resorting the list before proceeding.
- **Maximal Progress** The code has maximal-progress behaviour since, when a transition has been taken the list of active transitions is checked again from the beginning. Only when no active transition has a satisfied guard will the controller suspend and let other threads with lower priority execute. That is, tasks may execute, which corresponds to a delay transition.

4.4 Prototype for legOS

We have implemented a prototype code-generator for timed automata extended with tasks (cf. Definition 1) following the principles laid out above. As target for the prototype we have used legOS [Lt02], a small open source operating system that runs on the LEGO® Mindstorm RCX control brick. The OS is implemented in C which makes the choice of using C as the language for tasks and generated code simple.

The hardware consists of a Hitachi H8 micro-controller 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 task types released in the control automata to at most 19, since each need a unique priority in the schedulability analysis and the controller thread uses the highest priority level.

Another limitation in the current version of legOS is in interrupt handling. The OS uses polling for interaction with the environment, which imply that quickly changing sensor values could be missed if they occur between sampling points.

The polling of environment is performed by wake-up functions that are executed by the thread scheduler. Wake-up functions are used in legOS in all cases that a thread must wait for some condition to become true (such as the release of a needed semaphore). The thread will register a boolean function that the scheduler will execute when it looks for waiting thread that could be executed. The scheduler will pick the first waiting thread whose wake-up function return true. The scheduler is executed when the current thread is suspended or periodically every 20 ms[4].

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

Wake-up functions are the general method underlying all suspensions of threads in legOS: semaphores, timeouts, communication messages, key presses, etc.

One such wake-up function contain the code that execute the control structure. It is used by the control thread that just waits for the wake-up function to return true, but it never does. Instead the procedure is executed every 20 ms by the OS with interrupts disabled and while the system time is stopped.

Since legOS is not a real-time OS there is no guarantees on response times for system calls, e.g. creation of threads. But it is still desired to minimise their response time and variance. For thread creation we achieve this by starting on thread for each task type at boot time. These threads are sleeping until the controller signals that they should run their bodies once. The signaling uses a global integer array, the *release list*, with one element for each task type. A non-zero value, which is monitored by a wake-up function, indicates that the task should be executed. When the task has executed its body once the element is decreased by one, i.e. the value count the number of times the loop should be executed. Since priorities are fixed the release list is a representation of the ready queue.

Sensor readings in legOS are available to the generated program as sensor variables which are updated independently by the hardware. Sensor variables are either read by the task code and could influence the control automata through shared variables or are used directly in guards on the transitions. 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.

5 Production Cell Case-Study

In this section, we describe and illustrate how we apply the code synthesis framework described in the previous sections to a case-study of a LEGO® production cell.

5.1 The Production Cell

The Production Cell model used in this case-study is a model of an actual industrial unit in a metal plate processing plant in Karlsruhe. The model has been developed by FZI in Karlsruhe [LL95] 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 is composed of four subsystems, the *feed belt*, the *robot*, the *press* and the *deposit belt*, corresponding to the physical components as shown in Figure 3.

Each of the subsystems fulfills a specific operation during the plate processing:

The *feed belt* transfers a plate from the entry position to the end position where the robot arm named *arm A* can pick the plate up. 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

Figure 3: The LEGO® Production Cell.

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 on 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*.

Examining the system requirements we find that the processing of a metal plate comprises two kinds of actions:

- A local processing inside one subsystem, e.g. the plate is moved on the feed belt, or the plate is forged on the press.
- A transfer from one subsystem to the other, e.g. the plate is conveyed from the feed belt to the robot.

The first kind of action is performed locally within one subsystem while the second requires cooperation between two subsystems.

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. We shall here focus on the program controlling the *robot* and discuss its formal model in terms of timed automata extended with tasks.

5.2 The Robot Controller Model

The robot controller is composed of two automata, RobotControl and MoveTo. The RobotControl automaton controls the overall motions of the robot, and

Figure 4: The Structure of the Robot Controller.

communicates with both the feed belt controller and the local controller MoveTo. The MoveTo automaton controls the robot movements according to the rotation sensor (i.e. sensor 3 in Figure 3) information. Figure 4 sketches the structure of the robot controller.

In Figure 4, the robot controller communicates with the feed belt controller by means of the shared integer variable `Pos[0]`. `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. It communicates with MoveTo by means of the synchronized channels `start` (to start the robot movement) and `stop` (to stop the robot movement), and the shared integer variable `Goal` that sets the goal position where the robot should reach.

Figure 5 shows the formal model of the RobotControl process in terms of timed automata extended with tasks.

In Figure 5, 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 that there is 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 these two time values 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.

Figure 5: The RobotControl Process.

The automaton is initially at the 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 that:

- The robot is moving from one position to the other, e.g. the locations *Moving2FB* and *Moving2PrB*. At the former location the robot is moving towards the position where *arm A* points to the feed belt, and at the latter the robot is moving towards the position where *arm B* points to the press.
- The robot is at a position 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.

At each of the locations *PickUpA*, *PickUpB*, *DropA* and *DropB*, there is a corresponding executable task associated with. For example, at the location *PickUpA* the task PickUpA is executed for *arm A* to pick up a plate from the feed belt, while the location *DropA* is associated with the task DropA by which *arm A* drops a plate onto the press.

The model of the MoveTo process is given in Figure 7 in A where all the locations except the location *Entry* are associated with tasks to carry out the corresponding actions.

5.3 The Feed Belt Controller Model

In A we present also the model of the feed belt controller, which is composed of two automata, Alarm in Figure 8 and BeltPosition in Figure 9. The Alarm process handles the light sensor (i.e. sensor 1 in Figure 3) at the entry position of the feed belt. The light sensor senses the arrival of a plate on the feed belt. The BeltPosition process updates the positions of the plates on the feed belt periodically. In both processes, tasks associated with locations are given in bold font.

In Figure 8, the process is initially in the location *Calib*, corresponding to that it is calibrating the sensor background light so that the sensor reading is stable. The calibration is done by an associated task, Calib, which measures the light sensor for a while and calibrates for the steady state value of the sensor, and updates the shared variable `normalLight` with the calibrated value. Finally the task sets the boolean variable `AlarmTaskDone` to `TT`. Once the sensor is calibrated, i.e., `AlarmTaskDone==TT`, the process moves to the location *WaitFrontEdge*, where it waits until the light value measured by sensor 1 in Figure 3 falls below 80% of the normal light. This indicates that the front of a plate has passed the sensor and the process proceeds to the location *WaitBackEdge* where it waits for the whole plate to pass. This is sensed when the measured light value again rises to above 90% of normal light. Once there is a plate sensed, the process moves to the location *AtStart*, where a task, AtStart, will be executed to insert a value into the array `Pos[]` that handles the positions of the plates. The clock `ArrivalTime` is used to measure the time interval that a plate has been sensed. When it reaches to a value, `SAFE_CALIB`, it is safe to return to the location Calibrate. The time constant `SAFE_CALIB` is obtained by experiments.

The BeltPosition process, which is modelled as in Figure 9, is initially in location *Idle*. When there is a plate on the belt, indicated by `Pos[0]>-1`, the process moves to the location *Active*. This location is associated with a task UpdatePos which periodically is executed to update the positions of the plates in the array `Pos[]`. The period is represented by an invariant `x <= UpdateTime`, where x is again a clock. As soon as there is no more plates on the belt, which is indicated by `Pos[0]==-1`, the process moves back to the location *Idle*.

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

6 Production Cell Analysis and Code-Synthesis

The analysis and code synthesis presented in Section 4 have been implemented in the TIMES[5] tool [AFM+02]. The tool, shown in Fig. 6 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

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

Figure 6: A screenshot of the TIMES tool.

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 [FPY02].

The parameters and tasks used in the analysis of the production cell model are shown in Table 2 of A. Note that in the production cell model, tasks execute according to the fixed priorities listed in column P of Table 2. 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 Figure 11 and Figure 12 of 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, defined 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 sequences 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 of each type released simultaneously. A number of other correctness properties have 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 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 2GB 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 [DT98]), the analysis requires only 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[6]; the priorities of the tasks released by automaton MoveTo were adjusted to preserve the intended execution order[7]; 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 Gant chart view, shown in the lower right part of the screen shot shown in Figure 6, proved useful for tracing the executions of the tasks. In the Gant chart view it is illustrated how tasks are executed according to the chosen scheduling policy [AFM+02].

6.1 Synthesised legOS-code

We have used TIMES to automatically generate C code that can be compiled and executed on a LEGO® RCX-brick with legOS. The code generated from the analysed model of the production cell is listed in B. It is 550 lines long and produces an executable file of 4892 bytes that can be downloaded to the hardware.

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

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

The code consists of two parts: one generic run-time system (listed in B.2) which is part of any generated code for the legOS target, and one system specific part (listed in B.1).

The system specific code for the production cell model is basically an encoding of the control automata in five look-up tables where the guards, assignments, synchronisations, transitions and locations are stored, respectively. The implementation is centered around the transitions which are stored in an array of `trans_t` elements, which are 4-byte structs with the fields `active`, `from`, `to` and `sync`. The active field is either 0 or 1, where 1 indicates that the transition has a current location as source, i.e. it is included in the list ACTIVE. The array is sorted in priority order of the transitions. The system specific code for the production cell also consists of the task bodies which have been inserted into the code in B.

The information in the look-up tables is used by the run-time system, which basically is an implementation of the event handling procedure described in Table 1. The main function is `check_trans()` which loops through the transition array and evaluates the guards of the transitions marked as active. When a guard is satisfied, and the transition does not have a synchronisation channel, the transition is taken. Which means that its assignments are performed (by a call to the function `assign()` with the transition identifier as parameter), the field `active` in the transition array is cleared for all transitions leaving the source location and set for all transitions 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 a transition may synchronise, i.e. its `sync` field is greater than -1, `check_trans()` will call `check_sync()`. This function uses the table `chanusage` which contains the indexes of the transitions that may synchronise on a given channel. `check_sync()` will evaluate the guards of these transitions and return the index of the first that is true (or zero if none). If there is a transition that may synchronise the updates that `check_trans()` would do for a single transition are performed for both transitions, the sending one before the receiving.

In B.3 some definitions of macros and types used in the code is provided.

7 Conclusions

In this paper, we have presented a framework for model-based development of real-time and embedded systems, based on the recently suggested model of timed automata extended with tasks. It has been shown previously that design problems, such as schedulability and reachability analysis, are decidable for the model of timed automata with tasks. In this paper, we have presented how to synthesize executable code with predictable timing behavior, which is guaranteed to meet the constraints imposed on the design model.

We have presented a deterministic semantics of timed automata with tasks that is guaranteed to define a subset of the behavior of a given design model. Based on the deterministic semantics we generate executable code for a small

embedded real-time operating system. As prerequisites for the code-synthesis, the model must have been analysed to be bounded, schedulable, and guaranteed to reach stable states in which the tasks are able to execute. To ensure stability, we give a simple syntactic requirements such that code generated from any schedulable system is also guaranteed to not have any instable executions. Other analysis results are used to generate more efficient executable code, for example, the result of the boundedness analysis is used to limit the memory allocated, and reachability analysis results may be used to eliminate dead-code, etc.

To show an application of the presented framework, we have described a case-study of a well known production cell, built in LEGO® and controlled by LEGO® Mindstorm control bricks. The control program of the most central component, the two-armed robot, was designed and analyzed using the tool TIMES, in which we have implemented the analysis and code synthesis described in the paper. The tool has been used to generate executable C-code for the legOS operating system. The generated code has been compiled, downloaded, and executed on LEGO® Mindstorm bricks.

As future work we plan to adjust the code synthesis to produce executable code for other operating systems than legOS, such as RT-Linux, to make the presented framework available at other platforms. We also consider to extend the code-synthesis to also generate code for the scheduling kernel and the necessary run-time system functions. This would allow us to generate a tailored run-time system supporting only the functions used in the designed system, and to further optimize the run-time system based on the results from analyzing the design model.

Acknowledgment

The TIMES tool has been implemented in large parts by Leonid Mokrushin.The authors would like to thank him for his excellent work.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - a tool for modelling and implementation of embedded systems. To appear in Proc. of TACAS'02., 2002.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. 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, pages 546–550. Springer–Verlag, 1998.

[BG92] G. Berry and G. Gonthier. The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BGK+96] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[DKRT97] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the 3*rd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 416–431. Springer–Verlag, April 1997.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[EWY99] Christer Ericsson, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Proceedings of the 6*th *International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press, 1999.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. To appear in Proc. of TACAS'02., 2002.

[HL89] H. Hüttel and K. G. Larsen. The Use of Static Constructs in a Modal Process Logic. In *Logic at Botik'89.*, number 363 in Lecture Notes in Computer Science, 1989.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, number 891 in Lecture Notes in Computer Science. Springer–Verlag, 1995.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising uml state machines for model checking. In *In Proceedings of UML'99*, volume 1723, pages 430–445, Berlin, 1999. SPRINGER.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[Lt02] LegOS-team. The legos hompage. http://legos.sourceforge.net, January 2002. http://legos.sourceforge.net/.

[SMF97] Thomas Stauner, Olaf Mller, and Max Fuchs. Using HyTech to Verify an Automotive Control System. In *Proc. Hybrid and Real-Time Systems, Grenoble, March 26-28, 1997*. Technische Universität München, Lecture Notes in Computer Science, Springer, 1997.

[Yi91] Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Science, Chalmers University of Technology, 1991.

A The Analysis Model

Figure 7: The automaton MoveTo.

Figure 8: The automaton Alarm.

Figure 9: The automaton BeltPosition.

Figure 10: The automaton Brick.

Figure 11: The automaton Brick with decorations for verifications.

Figure 12: The automaton Robot.

Table 2: 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	Updates values in Pos[] $\forall i$ `Pos[i]:=(Pos[i]>=0?Pos[i]-1:-1)`
RobotControl				
DropA	4	16	2	Switch of magnet on arm A `TaskDone:=TT`
DropB	4	13	1	Switch of magnet on arm B `TaskDone:=TT`

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

PickUpA
decorated
	4	10	4	Switch on magnet A. Remove value from Pos[]
				$\forall i$ `Pos[i]:=Pos[i+1],`
				`Pos[BRICKS-1]:=-1,`
				`numBricks:=numBricks-1,`
				`dec_PUA:=1,`*`decoration`*
				`TaskDone:=TT,`

PickUpB	4	17	3	Switch on magnet on arm B `TaskDone:=TT`
MoveTo				
MvRight	4	10	8	Start motor to rotate robot right `RobotAngle:=ROTATION_1`
MvLeft	4	10	7	Start motor to rotate robot left `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`

A.1 Query File

```
// Generated by Paul, Elena and Leonid.

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

/*
Scheuler 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 )

/*
Whenever a brick arrives to the pickup position, it will be picked up by
robot arm A within ATPICKUP_T ms, i.e. before it is out of reach. (Needs decorations in
model)
*/
A[ ]( ( Brick.AtPickUp and Brick.x>=Brick.ATPICKUP_T ) imply dec_PUA==1 )

/*
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 )
```

B Generated Code

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;
}
```

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;
}
```

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);\
    }\
}
```