Verifying Temporal Constraints on Data in Multi-Rate Transactions using Timed Automata

Anders Wall, Kristian Sandström, Jukka Mäki-Turja, Christer Norström, and Wang Yi[1]
Mälardalen University
Mälardalen Real-Time research Centre (MRTC)
P.O Box 883, S-721 23 Västerås, Sweden
{awl,ksm,jma,cen}@mdh.se, yi@docs.uu.se

Abstract

Transactions involving multiple tasks, possibly with different period times, are common constructs used in the design of real-time systems. Data flowing through a transaction is usually subject to temporal constraints, such as maximum time from input to output or a maximum time difference between inputs. Such constraints are of great importance to guarantee the correct functioning of the designed system. But normally they cannot be checked using the traditional approach to schedulability analysis. In this paper we describe how to use timed automata and reachability analysis to verify such temporal constraints on data in transactions. By making a timed automaton model of the data dependencies in a transaction, we enable automatic verification of timing constraints such as end-to-end latency. The model can handle different computational models and any non-preemptive execution order of the tasks in the transaction. Our experiences from industrial case studies indicate that in a substantial number of applications, the transactions are of sizes that can be handled using this approach.

1. Introduction

Designing safety-critical real-time systems involves assessment of functionality, temporal requirements and dependability. The temporal requirements on such systems may come in many forms, examples are end-to-end deadlines, jitter constraints, and latency constraints. Such constraints can for example be found in multi-rate control systems, where sampling, control, and actuation may execute with different frequency. In such a system there are requirements on the delay from sampling to actuation in the feedback loop. More precisely, for a particular actuation one wants to know which sample the calculation is based on. The feedback loop delay is then defined as the time difference between actuation and sampling. To be able to fulfill such constraints, the designer has to have some means to express and preferably also verify them. However, computational models like fixed priority scheduling [1,2], and pre-run-time scheduling [3], used in the industry, cannot directly express such constraints. For instance, jitter constraints are handled by manual transformations into release times and deadlines of individual tasks. Translating these constraints to the attributes of the computational model is non-trivial and the schedulability analysis does not verify that the translation itself is correct.

In this paper, we will show how to verify that this mapping is correct. We do this by presenting an algorithm that transforms the data flow and available timing information of an application, or part of an application, into timed automata. In addition, we construct an automaton modeling the execution strategy that defines the execution orders of the involved tasks. By composing these two automatons and by using model checking, we can verify timing constraints such as latency. The benefit of this is two folded. First, the result from the scheduler can be checked, and second the high-level requirements from the specification can be verified. We also believe that this is the starting point for integrating real-time scheduling and timed automata to enable efficient design and verification techniques of both time-triggered and event-triggered systems in one framework.

Several researchers including Mok, Gerber, and Kim have provided specific computational models that directly allow specification of latency constraints [4, 5, 6]. Our approach makes few assumptions about the computational model, and can therefore be applied to different

[1] Also at the Department of Computer Systems, Uppsala University, Sweden

computational models. Furthermore, it also gives the possibility to model the functional behavior of tasks and to efficiently integrate handling of event-triggered tasks by defining an environment model, as reported in [7].
The paper is structured as follows: In section 2 we define transactions, data dependency, and execution strategies. Section 3 describes the construction of a timed automaton that models data dependencies and how to verify temporal constraints on data. Finally, in Section 4 we present our conclusions.

2. Transactions, Data Dependency Model and Execution strategies

A transaction is a set of tasks, collaborating in order to provide some desired function. For instance, consider a control transaction consisting of three tasks, *sample*, *control*, and *actuate*. The task *sample* reads an input value from the process, performs some filtering, and thereafter sends the value to the *control* task. The control task consumes the sample value, reads a reference value, and calculates a new control signal. Finally, the actuator task consumes the new control signal and imposes it on the controlled process. In our model, each task in a transaction has an input – calculate – output behavior. That is, when the task starts its execution it first consumes all its input data, performs the computations, and before completion, it outputs the results.

The execution of tasks is considered to be non-preemptive. Apart from being non-preemptive, tasks may execute according to any strategy, e.g., time driven or event driven. Furthermore, transactions can be of multi-rate nature, i.e., tasks in the transaction may execute with different rates. In Figure 1, a transaction consisting of four tasks is displayed, where 1, 2, and 3 are input to the transaction and the arrows describe the data flow through the transaction.

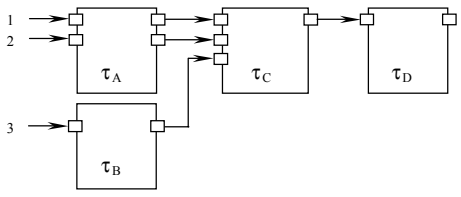

Figure 1. An example transaction.

We will represent a transaction as a data dependency graph. A data dependency graph is a set of nodes, $n_0,\ldots,n_p$, that represents the inputs and the tasks in the transaction, and a set of edges that represents the data flow in the transaction. The initial nodes of the dependency graph model the inputs to a transaction. If a task τ_q consumes several different data from task τ_p, only one edge between those nodes is needed. Moreover, if a task reads several inputs, there is need for only one initial node representing those inputs. This is not a restriction but a consequence of the input – calculate – output behavior of tasks, in which a task reads all its inputs at the beginning of its execution.
Figure 2 illustrates the data dependency graph of the transaction in Figure 1, where task τ_C depends on data produced by both τ_A and τ_B. Moreover, τ_D depends on τ_C. Note that τ_A consumes data from input 1 and 2, which is represented in the dependency graph as a single initial node n_0.

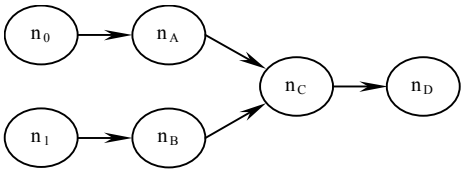

Figure 2. The data dependency graph for the example transaction.

Definition 1. *A data dependency graph for a transaction is a Directed Acyclic Graph (DAG) defined by a tuple* $\langle N, E, N_0, n_{end}\rangle$*, where*

- N *is a finite set of nodes representing tasks in the transaction.*
- $E \subseteq N \times N$ *denotes the edges between nodes.*
- $N_0 \subseteq N$ *is a finite set of initial nodes denoting the inputs.*
- $n_{end} \in N$ *represents the last task in the transaction.*

□

Note, that for each initial node $n_0 \in N_0$ there exist only a single edge e_0 to another node. If several tasks in a transaction read the same input, two or more initial nodes in the data dependency graph can be used to model that input. Each node $n_p \in N$ has an execution time specified as an interval $C(n_p)=[bcet, wcet]$, where *bcet* is the best-case execution time and *wcet* is the worst-case execution time for task τ_p.

Furthermore, as tasks in the transaction may execute in any arbitrary order, the dependencies do not imply a precedence relation between tasks. The execution order depends upon the execution strategy, e.g., event triggered tasks with fixed priorities or time triggered pre-run-time scheduled tasks. Formally, an execution order is defined as follows:

Definition 2. *An execution order* σ *is a sequence of pairs* $\langle t, s\rangle$ *where* $t \in N$ *denotes the start time and* $s \in N^+$ *is a sequence of one or more tasks, thus* $\sigma \in (N \times N^+)^*$.

□

The start time for the first task in the sequence *s* is equal to *t*, whereas the remaining tasks in *s* start as soon as the preceding tasks complete. Thus, the start time of a task, that is not the first task in *s*, is determined by the start time of the preceding task τ_p and the execution time interval ranging from *bcet* to *wcet*. An example of an execution order involving four tasks is $\{\langle 0, \tau_A \cdot \tau_B \cdot \tau_C \rangle, \langle 12, \tau_A \cdot \tau_D \rangle\}$.

From the data dependency graph the dependencies for each task in the transaction can be derived. The data dependencies for task τ_p are represented as the set $L(n_p)$ of independent paths from the set of initial nodes to node n_p. Formally a data dependency is defined as follows:

Definition 3. *A data dependency relation is*

- $L(n_0) = \varnothing$ *where* $n_0 \in N_0$
- $L(n_q) = \bigcup_{(p,q)\in E} \{\mu \cdot p \mid \mu \in L(n_p)\}$

□

Note that μ denotes a path from $n_0 \in N_0$ to node n_p. For instance, the dependency set $L(n_C)$ for node n_C in the data dependency graph depicted in Figure 2 is given as $L(n_C)=\{n_0 \cdot n_A, n_1 \cdot n_B\}$.

We will denote the set of all the data dependencies for the tasks in a transaction as *L*, which is a union of all data dependency sets.

3. Verifying temporal constraints using timed automaton

Timed automata has been recognized as a basic semantic model for specifying and verifying timing constraints for real-time systems. Here we give a brief introduction to the model of timed automata. For details, we refer to [9].

A timed automaton is a standard finite-state automaton extended with a finite set of real-valued clocks. On each transition there are constraints (guards) on clocks, synchronization action, and clocks to be reset. Whenever the guard is satisfied of the current values of the clocks, the transition can be taken, i.e., the synchronization action is performed and the clocks to be reset are set to 0. A state of a timed automaton can be considered as a tuple containing the current node of the finite automaton, and the current values of the clocks. Informally, the semantics of a timed automaton is given by two transition rules. First of all, it can stay in the current node letting time pass (delay), i.e. the clocks are updated and the current node remains unchanged. Secondly, it can take the transition instantaneous resulting in a state with a new node. In recent years, there have been a number of software tools developed e.g. KRONOS and UPPAAL [8,10] for automated analysis of logical properties of timed automata.

In this paper, we are aiming at using the existing tools to verify timing constraints on transactions by transforming the data dependency model to timed automata. We will refer to a timed automaton that describes data dependencies as a *data dependency automaton*. The temporal constraints that can be verified using the approach proposed in this paper are:

- End-to-End timing constraint, i.e., minimum and maximum time from readings of inputs until the end of the transaction.
- Variation in End-to-End timing, i.e. output jitter.
- Input synchronization, i.e., minimum and maximum time difference between input readings used by the transaction in order to produce a result.

3.1. From Data Dependency Graphs to Timed Automata: an Example

In this subsection, we use an example to show the main idea and intuition of the translation algorithm. The transaction for the example, illustrated in Figure 3, consists of the input *k* and the three tasks τ_A, τ_B, and τ_C. The task executes according to a non-preemptive time-triggered strategy. We want to verify that the data that τ_C uses to produce the result for the transaction does not origin from an input reading that is older than 10 time units.

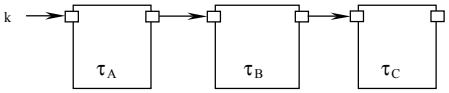

Figure 3. The example transaction with its three tasks.

The data dependency graph representing the transaction in Figure 3 will consist of four nodes and is displayed in Figure 4. The node n_{end} represents the last task τ_C.

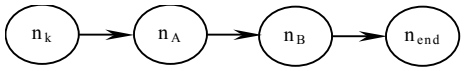

Figure 4. The data dependency graph.

According to definition 3, all tasks in the transaction depend on input data *k*. The data dependency relation sets for the nodes are $L(n_k) = \{\ \}$, $L(n_A) = \{n_k\}$, $L(n_B) = \{n_k \cdot n_A\}$, $L(n_{end}) = \{n_k \cdot n_A \cdot n_B\}$.

In the data dependency automaton we measure the age of data when an input data instance has been processed by the last task in the transaction (τ_{end}). Therefore we use time stamps to measure the time elapsed since a particular data entered the transaction. We denote a time stamp for

inputs represented by the initial node n_k in the dependency graph as X^k. As the transaction might be of a multi-rate nature, more than one instance of an input could exist simultaneously in the transaction, all with different age. Consequently, for all initial nodes in the dependency graph, there must be one or more associated time stamp instances. The actual number of time stamp instances for an initial node is correlated to the number of paths from that initial node to all other reachable nodes in the dependency graph, excluding n_{end}. For instance, the number of time stamp instances needed for X^k in this example is two, since there is one path from n_k to n_A and one additional path from n_k to n_B (see $L(n_A)$ and $L(n_B)$).

We use clocks, which can only be reset, to implement time stamp instances in timed automata. Therefore, when a task that reads an input executes, the corresponding time stamp instance is reset. When consumers of data produced by that task execute, the time stamp instance is distributed. Since several time stamp instances may be needed for an input, nodes in the data dependency graph may use any of these instances. To ensure that the time stamps are consistent, a state in the data dependency automaton is used to keep track of the time stamp instances currently used by each node, i.e., one state in the data dependency automaton models the assignments of time stamp instances for all nodes. This gives for all nodes the age of all data that the tasks, represented by the nodes, have read at this point. Table 1 presents the assignment of time stamp instances for state *S0* to *S3* in the data dependency automaton displayed in Figure 5.

node	n_A	n_B
S0	X_1^k	X_1^k
S1	X_2^k	X_1^k
S2	X_2^k	X_2^k
S3	X_1^k	X_2^k

Table 1. Time stamp instance assignments for state *S0* to *S3* in the dependency automaton.

Since several nodes simultaneously can use the same time stamp instance, a time stamp instance cannot be reset without considering possible multiple uses. Assume that a task τ_p executes, and as a consequence a time stamp for an input should be reset. If the time stamp instance used by node n_p is used by at least one other node, then node n_p will have to use a time stamp instance that is not assigned to any node. Consequently, a transition has to be made to a state in which node n_p is assigned the new time stamp instance. As an example of such an instance replacement consider the transition from *S0* to *S1* in Figure 5 where the time stamp instance for node n_A changes from X_1^k to X_2^k. If, on the other hand, no other node in the data dependency graph uses X_i^k, it can be reused. The transition *S1* to *S1* in Figure 5 is an example of reusing a time stamp instance. In this case X_2^k is reused in order to reflect the most recent reading of input *k*.

In addition to the states that are needed to represent the use of time stamp instances, there must exist states that represent that the transaction is completed, i.e., corresponding to the last task, τ_{end}. We will refer to such a state as an *end-state*. Upon a transition from a state *S* to an end state a time stamp is reset, thereby making it possible to verify the end-to-end age constraint on the data. Note that this transition does not affect the assignment of time stamp instances, thus there is always a transition back to state *S* in the automaton.

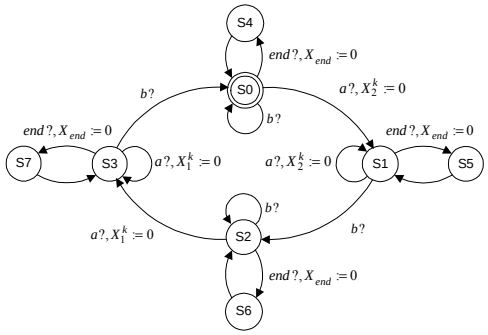

Figure 5. The data dependency automaton for the example.

In order to verify temporal constraints for a transaction, an automaton describing the execution strategy is needed. The execution strategy of tasks can be time triggered or event triggered. For time triggered systems the translation of the execution order to a timed automaton is straightforward. If the system is event triggered, the system environment that generates the events must be modeled as well [7]. For the purpose of illustration, a simple execution scenario for the tasks in the example transaction is depicted in Figure 6. All tasks have a *bcet* equal to 1 time unit and a *wcet* equal to 2 time units. The start time for the two instances of τ_A and the second instance of τ_B is fixed, whereas the start times for the rest of the task instances are relative to the preceding task. The complete execution sequence is repeated as soon as the second instance of τ_C has completed.

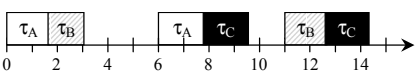

Figure 6. A possible execution scenario in the three-task transaction.

The execution scenario depicted in Figure 6 results in the automaton illustrated in Figure 7.

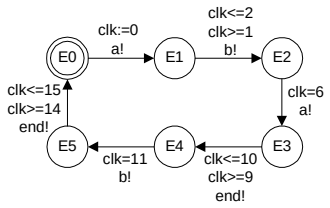

Figure 7. The execution order automaton.

The execution order automaton in Figure 7 give rise to the following state transitions in the data dependency automaton in Figure 5:

$$S0 \xrightarrow{a?, X_2:=0} S1 \xrightarrow{b?} S2 \xrightarrow{a?, X_2:=0} S3$$

$$\xrightarrow{end?, X_C:=0} S7 \longrightarrow S3 \xrightarrow{b?} S0$$

$$\xrightarrow{end?, X_C:=0} S4 \longrightarrow S0$$

The transaction is completed when the data dependency automaton reaches the end-states S7 and S4. Consequently, the age constraint is satisfied if, when in those states, a related reading of input *k* occurred no longer than ten time units earlier. By related reading we mean a reading of data that has propagated through the entire transaction. Thus, in order to verify the age constraint the following invariant must hold.

$$\forall \Box$$
$$((S4 \rightarrow (X_{end} \geq X_1^k - 10)) \wedge (S5 \rightarrow (X_{end} \geq X_1^k - 10)) \wedge$$
$$(S6 \rightarrow (X_{end} \geq X_2^k - 10)) \wedge (S7 \rightarrow (X_{end} \geq X_2^k - 10)))$$

That is, in all possible execution scenario, restricted by the execution automaton, the time difference between an instance of the time stamp corresponding to the reading of input *k* (X_1^k or X_2^k), and the time stamp corresponding to the completion of τ_C, is never greater than 10 time units.

1.2. From Data Dependency Graphs to Timed Automata: the Translation Algorithm

In this section we present how to construct a timedautomaton that represents the data dependencies for a given transaction. The translation considers the data dependencies in the transaction, represented by the *L* set. Transactions are assumed to have a N-to-1 topology. That is, there can be multiple tasks that may read inputs, but only one task that produces outputs. A transaction with multiple tasks producing outputs can be represented as several different transactions, each with a single output-task. Furthermore, the execution of tasks is considered to be non-preemptive.

As discussed in Section 3.1, several instances of a time stamp may be needed in order to keep track of the age of data flowing through a multi-rate transaction. We denote a particular instance *i* of time stamp X^k as X_i^k . The number of time stamp instances num_k needed to measure the age of input data represented by an initial node n_k is finite and equal to the number of nodes depending on n_k, i.e., the size of $\{n_k \cdot \sigma | n_k \cdot \sigma \in L\}$where $n_k \in N_0$. As the tasks in the transaction can execute in arbitrary order, they can in particular execute in a manner that gives each task a unique age of the input data it depends on. If the number of time stamps needed for an initial node n_k is num_k, the instances will be enumerated from 1 to num_k.

The age of an input *k*, that a task depends on, is represented as a pair consisting of the time stamp for *k*, and the path from the initial node in the data dependency graph that represents *k* to the node that represents the task. This path uniquely identifies a dependency and distinguishes different dependencies of the same data. For each task τ_p in a transaction, the set $A(n_p)$ contains all such pairs for the data that τ_p depends on.

Definition 4. *The set $A(n_p)$ contains pairs $\langle X^k, s\rangle$ where X^k is the time stamp of input, represented by the initial node n_k, and s is a path from n_k to the node n_p, representing task τ_p.*

$$A(n_p) = \bigcup_{n_k \cdot \mu \in L(n_p)} \left\langle X^k, n_k \cdot \mu \right\rangle$$

□

Note that for each pair $\langle X^k, s\rangle$ in $A(n_p)$, the path *s* is static whereas the particular instance *i* of time stamp X^k may vary between states in the data dependency automaton. We will use *A* to denote the set containing the sets $A(n_p)$ for all tasks in the transaction excluding the last task τ_{end}.

A state in the data dependency automaton represents a unique time stamp instance assignment for the set *A*. Moreover, the number of states in the data dependency automaton is finite. Thus, the problem of verifying temporal constraints using reachability analysis is decidable.

Proposition 1. *The number of states in the data dependency automaton is finite and given by:*

$$2 * \prod_{n_p \in N} \prod_{\langle X^k, s\rangle \in A(n_p)} num_k \quad \textit{where } n_p \neq n_{end}.$$

Proof: Since every time stamp X^k in $A(n_p)$ can be one of num_k instances, there are

$\prod_{\langle X^k, s\rangle \in A(n_p)} num_k$ ways of constructing $A(n_p)$.

The total number of possible states for the data dependency graph excluding n_{end} is then given as all possible ways of combining the time stamp instances for

all nodes. Consequently, the total number of combinations is given by:

$$\prod_{n_p \in N} \prod_{\langle X^k, s \rangle \in A(n_p)} num_k$$

Since from every state there must be a transition to a unique end state, the total number of states in the data dependency automaton is $2 * \prod_{n_p \in N} \prod_{\langle X^k, s \rangle \in A(n_p)} num_k$

□

We will now present the rules for constructing the time automaton representing the data dependency graph for a given transaction that complies with the assumptions given earlier in this section. The automaton is constructed starting from an initial state *S* and the rules *G1* to *G5* decides what action to take and how the states changes when a task τ_p executes. Two basic rules R_1 and R_2 constitute the basis for *G1* to *G4*, whereas *G5* corresponds to completion of the transaction. If node n_p is an immediate successor to the initial node n_k and if node n_p uses the time stamp instance X_j^k to represent the age of data read by task τ_p, then R_1 is satisfied if X_j^k is not used by any other node. Moreover, R_1 is also satisfied if node n_p does not depend upon an initial node. The second rule R_2 is satisfied if every immediate predecessor to node n_p in the data dependency graph uses the same time stamp instance as n_p itself. R_2 is also satisfied if n_p has no dependencies to other nodes.

$$R_1: \langle X_j^k, n_k \rangle \in A(n_p) \rightarrow \langle X_j^k, n_k \cdot \mu \rangle \notin A(n_q) \vee n_q = n_p$$

$$R_2: \langle X_j^k, n_k \cdot \mu \cdot n_q \rangle \in A(n_p) \rightarrow \langle X_j^k, n_k \cdot \mu \rangle \in A(n_q)$$

In Figure 8, the transitions corresponding to rules *G1* to *G5* are displayed. For each node in the data dependency graph, one out of four possible transitions, *G1* to *G4*, should be present in every state of the resulting timed automaton. The rules *G2* to *G4* result in a change of the time stamp instance assignment in the set *A* and therefore a transition must be made to another state in the timed automaton that represents the assignment of time stamp instances. *G1* on the other hand, does not change time stamp instance assignments, and consequently, a transition to a new state in the timed automaton is superfluous. Finally, *G5* corresponds to completion of the transaction by reaching an *end-state*. The rules should be applied in all states for all nodes in the dependency graph.

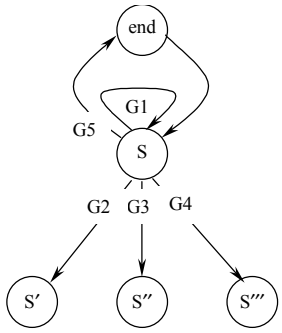

Figure 8. The rules for constructing the data dependency automaton.

The rules are described by a condition consisting of a composition of the basic rules R1 and R2, an action describing the transition taken in the timed automata and which time stamp instance, if any, that should be reset. Furthermore, the initial state *S* is formed as:

$$S = A \text{ where } \langle X_j^k, s \rangle \in A(n_p) \Rightarrow j = 1$$

That is, initially tasks that depend on the same input data uses the same time stamp instance for that data.

Transition G1. The time stamp instance for the input (if any) that the task reads is not used by any other task, and there is no updated time stamp for the data that the task consumes (if any). The time stamp for the input is updated.

Condition: $R_1 \wedge R_2$

Action:

$$S \xrightarrow{p?, X_j^k := 0} S \text{ iff } \langle X_j^k, n_k \rangle \in A(n_p)$$

$$S \xrightarrow{p?} S \text{ iff } \langle X_j^k, n_k \rangle \notin A(n_p)$$

Transition G2. The time stamp instance for the input that the task reads is used by at least one other task. If the task consumes data from other tasks, there is no updated time stamp, i.e., the tasks use the same time stamp. As a consequence, the state *S* changes to S'.

Condition: $\neg R_1 \wedge R_2$

Action: $S \xrightarrow{p?, X_j^k := 0} S' \text{ where } \langle X_j^k, n_k \cdot \mu \rangle \notin A(n_q) \text{ for all } n_q \in N$

Transition G3. The time stamp instance for the input (if any) that the task reads is not used by any other task, but there are one or more updated time stamps for the data that the task consumes. The state *S* changes to *S″* as one or more time stamp instances has to be changed considering the data dependency.

Condition: $R_1 \wedge \neg R_2$

Action:

$S \xrightarrow{p?, X_j^k := 0} S''$ *iff* $\langle X_j^k, n_k \rangle \in A(n_p)$

$S \xrightarrow{p?} S''$ *iff* $\langle X_j^k, n_k \rangle \notin A(n_p)$

Transition G4. The time stamp instance for the input that the task reads is used by at least one other task, and there are one or more updated time stamps for the data that the task consumes. The new state S''' reflects the fact that we need both a unique time stamp instance for the input and that one or more time stamp instances have to be changed considering the data dependency.

Condition: $\neg R_1 \wedge \neg R_2$

Action: $S \xrightarrow{p?, X_j^k := 0} S'''$ *where* $\langle X_j^k, n_k \cdot \mu \rangle \notin A(n_q)$ for all $n_q \in N$

Transition G5. The last task in the transaction executes and completes. The time stamp X_{end} is reset on the completion of τ_{end}.

Condition: τ_{end} completes.

Action: $S \xrightarrow{end?, X_{end} := 0} end \longrightarrow S$ *where end* = n_{end}

1.3. From Data Dependency Graphs to Timed Automata: The example revisited

As an example on how to apply the rules *G1-G5* in order to construct a data dependency automaton, reconsider the transaction of the example in Section 3.1. The data dependency graph for that transaction is equal to the graph in Figure 9.

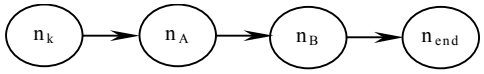

Figure 9. The data dependency graph.

The L sets for the nodes are $L(n_k) = \{\ \}$, $L(n_A) = \{n_k\}$, $L(n_B) = \{n_k \cdot n_A\}$, $L(n_{end}) = \{n_k \cdot n_A \cdot n_B\}$. Using Definition 4 gives the *A* sets $A(n_A) = \{\langle X^k, n_k \rangle\}$ and $A(n_B) = \{\langle X^k, n_k \cdot n_A \rangle\}$.

For the initial node *S* the sets $A(n_p)$ is assigned time stamp instances according to $S = A$ where $\langle X_j^k, s \rangle \in A(n_p) \Rightarrow j = 1$, which gives $A(n_A) = \{\langle X_1^k, n_k \rangle\}$ and $A(n_B) = \{\langle X_1^k, n_k \cdot n_A \rangle\}$.

Now we will explore the rules for constructing a data dependency automaton by deducing the transitions from the initial state *S*.

Transition G1 ($R_1 \wedge R_2$)

Starting with node n_A, rule R_1 is not satisfied since $\langle X_1^k, n_k \cdot n_A \rangle \in A(n_B)$ and thereby no transition is taken. For node n_B, both rule R_1 and R_2 is satisfied and therefore the transition $S \xrightarrow{B?} S$ is taken. Since $\langle X_1^k, n_k \rangle$ is not in $A(n_B)$ no time stamp instance is reset.

Transition G2 ($\neg R_1 \wedge R_2$)

For node n_A, $\neg R_1$ is satisfied as well as rule R_2, and thereby the transition $S \xrightarrow{A?, X_2^k := 0} S'$ is taken, where $\langle X_2^k, n_k \cdot \mu \rangle \notin A(n_q)$ for all $n_q \in N$. For state S' $A(n_A) = \{\langle X_2^k, n_k \rangle\}$ whereas $A(n_B) = \{\langle X_1^k, n_k \cdot n_A \rangle\}$ remains unchanged. For node n_B, $\neg R_1$ is not satisfied and therefore no transition is taken.

Since neither *G3* ($R_1 \wedge \neg R_2$) nor *G4* ($\neg R_1 \wedge \neg R_2$) is satisfied for any of the nodes, there are no transitions for these cases.

Transition G5

Finally, according to *G5*, the transition $S \xrightarrow{end?, X_{end} := 0} end \longrightarrow S$ is added to the automaton.

The part of the data dependency automaton constructed so far is displayed in Figure 10. Repeating the procedure above for state *S'* will eventually complete the automaton, resulting in the automaton of Figure 5.

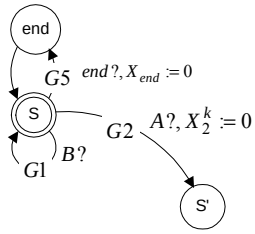

Figure 10. The partial data dependency automaton.

4. Conclusions

A temporal constraint can for instance be the time from input to output or the time difference between several inputs to a transaction. Such constraints of a transaction are not always possible to express in the task models at hand. Thus, the designer has to map such a constraint manually onto the temporal attributes in the existing task model, e.g. period times, deadlines, offsets, etc. The schedulability analysis only verifies whether the mapped system description can be realized or not. It does not verify that the requirement mapping itself is correct. In this paper we have described how to use timed automata to verify temporal constraints on data in transactions. By constructing a timed automaton model of the data

dependencies in a transaction, we enable verification of, for instance, end-to-end constraints using model-checking.

As the model is general, it can handle arbitrary computational models and execution orders of the task in the transaction. The main contribution of the paper is the rules for automatically generating such a model in timed automata. Although the size of the constructed automaton grows, in the worst case, exponentially with the number of tasks in a transaction, we believe that the method is suitable and applicable to real-world applications. The method can be applied to one transaction in isolation, i.e., modeling and verification can be performed on one transaction at a time. Consequently, only the transactions of interest in the complete system have to be verified. Our experiences from industrial case studies indicate that, in a substantial number of applications, the majority of the transactions are of size feasible for this method.

As future work we will extend the method to also include preemptive execution strategies and we will implement a tool that, from a given system description, generates data dependency automata and timed automata modeling the tasks' execution. The model of the tasks' execution will be derived from an existing scheduler, and for model-checking we will use the existing model-checkers, e.g. UPPAAL [10]. Furthermore, we plan to investigate the possibility of making timed automata models of the functional behavior of tasks at some appropriate level of abstraction. Such models enable verification of, not only temporal correctness, but also functional properties such as safety and functional correctness.

5. References

[1] Liu C. L. and Layland J. W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of ACM 20(1), 1973.

[2] Audsley N. C., Burns A., Davis R. I., Tindell K., and Wellings A. J. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. Real-Time Systems 8(2-3): 173-198, 1995.

[3] Xu Jand Parnas D. L. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. IEEE Transaction on Software Engineering, Vol. 16 No. 3, March 1990.

[4] Mok A. K., Tsou D., and De Rooij R. C. M., The MSP.RTL Real-Time Scheduler Synthesis Tool, In proceedings of 17th IEEE Real-Time Systems Symposium, pp. 118-128, 1996

[5] Gerber R., Hong S., and Saksena M. Guaranteeing Real-Time Requirements with Resourse-Based Calibration of periodic Processes. IEEE Transactions on Software Engineering, 21(7), 1995.

[6] Kim N. A Scheduling Technique for Real-Time Systems with End-to-End Timing Constraints, In proceedings of RTCSA, 1996.

[7] Norström C., Wall A., and Yi W., Timed Automata as Task Models for Event-Driven Systems, In proceeding of RTCSA, 1999.

[8] Daws C. and Yovine S., Two examples of verification of multirate timed automata with KRONOS, In proceedings of 16th IEEE Real-Time Systems Symposium, PP 66-77, 1995

[9] Alur R. and Dill D. A theory of timed automata, Theoretical Computer Science vol. 126 pp. 183-235, 1994

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