
This is the unspecified version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: http://openaccess.city.ac.uk/2895/

Link to published version: http://dx.doi.org/10.1109/ASE.2004.10002

Copyright and reuse: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk
Abstract

The analysis of hard real-time systems, traditionally performed using RMA/PCP or simulation, is nowadays also studied as a scheduler synthesis problem, where one automatically constructs a scheduler which can guarantee avoidance of deadlock and deadline-miss system states. Even though this approach has the potential for a finer control of a hard real-time system, using fewer resources and easily adapting to further quality aspects (memory/energy consumption, jitter minimisation, etc.), synthesised schedulers are usually extremely large and difficult to understand. Their big size is a consequence of their inherent precision, since they attempt to describe exactly the frontier among the safe and unsafe system states. It nevertheless hinders their application in practice, since it is extremely difficult to validate them or to use them for better understanding the behaviour of the system.

In this paper we show how one can adapt data-mining techniques to decrease the size of a synthesised scheduler and force its inherent structure to appear, thus giving the system designer a wealth of additional information for understanding and optimising the scheduler and the underlying system. We present, in particular, how it can be used for obtaining hints for a good task distribution to different processing units, for optimising the scheduler itself (sometimes even removing it altogether in a safe manner) and obtaining both per-task and per-system views of the schedulability of the system.

Keywords: Software engineering, data-mining, hard real-time systems, schedulability analysis, scheduler synthesis, decision-tree induction.

1. Introduction

The ongoing advances of the model-checking community have made possible a third possibility to Rate Monotonic Analysis (RMA) [6]/Priority Ceiling Protocol (PCP) [9] and simulation for the analysis of hard real-time systems - that of scheduler synthesis [1]. The basic idea behind it is that one will model the system without performing any of the model “normalisations” needed by RMA/PCP, just as is done in the case of system simulation. However, instead of simulating the system, one uses automated tools for examining the entire state space of this model and synthesise a scheduler for it which can guarantee that all deadlock and deadline-miss states will be unreachable from now on. Of course the computational cost of performing scheduler synthesis is a lot higher than that of the RMA/PCP analysis. This is somewhat offset by the fact that scheduler synthesis, unlike RMA/PCP, can be applied to any kind of system and that, unlike simulation (itself a computationally expensive method), scheduler synthesis can prove (better yet, render) systems correct. Another incentive for using scheduler synthesis is that the models used with it are usually more detailed than the ones used for RMA/PCP, which means that systems declared to be unschedulable by RMA/PCP due to their model over-approximations may well be proved to be schedulable when using synthesis. So, by imposing the RMA/PCP policy upon the system model, one could well use scheduler synthesis as a more expensive but also more accurate method to perform RMA/PCP schedulability analysis. Finally, unlike RMA/PCP which imposes a static set of task priorities that are difficult to combine with scheduling policies for extra qualities, synthesis can produce schedulers which are easier to combine with quality scheduling policies of this sort. This is because synthesis imposes the minimum set of constraints upon the system for rendering it safe and, thus, leaves many choices, among which we can choose those which best optimise the behaviour of the system with respect to further qualities (memory/energy consumption, jitter minimisation, etc.).

In order to evaluate the practical usefulness of the scheduler synthesis method, we have investigated its application to real-time Java programmes [5, 4] obtaining promising results concerning its applicability. We investigated in particular a synthesis methodology [5] for obtaining sched-
ulizing constraints which can be directly linked to: (i) different safety properties (e.g., deadlock avoidance, respect of deadlines) and (ii) underlying assumptions concerning the system execution model (e.g., non-preemptive execution versus preemptive execution) or (iii) different assumptions about the scheduler itself (e.g., ability to observe the system clocks or lack of it). Our aim was to obtain schedulers which are easier to understand and analyse in practise, than what one would normally obtain when applying scheduler synthesis. At the same time, this methodology has the advantage of decreasing the problem of state space explosion, since it is based on examining successive refinements of the system model, thus paying the cost of extra accuracy only when this is absolutely necessary and beneficial. The schedulers thus synthesised are a set of predicates on the current system state, each declaring a system thread to be safe to execute or not.

Nevertheless, these schedulers still suffer from a number of problems, which hinder their practical use. First of all, they tend to be big, since we had refrained from any sort of optimisation on the predicates. In fact, synthesised schedulers are by nature considerably bigger than the set of static priorities the RMA/PCP analysis produces, since this approach attempts to construct an exact description of the frontier among the safe and unsafe system states. This precision allows for an easier control of the system, hopefully using fewer resources, and for an easier consideration of further quality aspects of the system during scheduling. However, their big size is particularly problematic in the case of embedded systems, where we have stringent memory constraints. In addition to memory, the big size of our schedulers has also repercussions upon its execution time, which we would like to render as small as possible; indeed, in our models we make the assumption that the scheduler executes “instantly”, i.e., that its execution time is negligible when compared to that of the system sub-tasks. Another problem with the schedulers we synthesise is that they are completely unstructured. That is, for each system task we synthesise a long predicate, as a disjunction of the conjunctions which characterise each unsafe state for this task, i.e., the states where the scheduler should not allow this task to execute so as to keep the whole system in the set of states where the safety properties can be guaranteed. This form makes it extremely difficult for a human to understand the scheduling constraint that should be imposed upon a system task, thus rendering the validation of the scheduler problematic. At the same time, it does not allow the designer to obtain any kind of reasonable feedback concerning the behaviour of the system, which could help in seeding further system optimisations.

When referring to minimisation of some predicate, one may naturally think of binary decision diagrams (BDDs). Nevertheless, BDDs’ primary role is to provide a canonical representation which allows for fast computations with predicates. It is well known that what plays a crucial role in obtaining a small BDD is the choice of the variable ordering along the diagram. BDDs do not offer any substantial help in this respect and the fact that they impose a single variable ordering across all the decision tree branches sometimes makes it difficult to obtain a small decision tree.

In our context, where we do not need to perform any further computations with our scheduling constraints and where we would also like their final representation to be easily readable by humans, we can obtain a much better representation by using machine-learning techniques for data-mining. Indeed, there exist data-mining methods such as ID3 [7] which can produce a small representation of our scheduling constraints as a decision tree, using information-theory heuristics for choosing the order of variables at each sub-tree. This last feature allows data-mining methods to produce a decision tree which is more useful to a human, highlighting the system variables which are more important at different stages of the system’s execution, at least as far as the schedulability of the system is concerned. As usual with applying data-mining techniques, we will have to adapt the basic data-mining algorithm to our particular domain, in order to reap the most benefits out of it.

In the following we present the basic idea behind scheduler synthesis and the scheduler we synthesise for a simple case study. We then show how one can adapt ID3 to better structure synthesised scheduler constraints and analyse/optimise a real-time system.

2. Synthesising Scheduler Constraints

The basic idea behind scheduler synthesis for safety properties (e.g., deadlock freedom, guaranteeing deadlines) is to explore the complete state space graph of the system and find the states where the system scheduler must not allow some application tasks to execute, in order to be able to guarantee that the system will avoid undesired states later on (e.g., states where the system is deadlocked, or misses some deadline). If at some state the scheduler is left with no safe choice, we add it to the set of bad states, so that we can avoid it as well (otherwise, the scheduler itself may cause the system to deadlock). The scheduler takes its decisions based on a set of observation variables, which form its model of the current state of the system. In our setting, these variables are the programme counters of the application tasks and the global clock of the system. Note, that in our models the scheduler is not allowed to exert control to the system at all states [5]. It can only control the application at states where the application attempts to lock (or unlock) some resource, or when an application task becomes ready to execute after some alarm has expired (e.g., a new
task period has been signalled), or the task has timed-out while waiting for some notification). This effectively means that the programme counters of the tasks need not refer to real programme instructions; they can instead be abstract locations of these programmes signifying the start (respectively end) of some block of non-controllable actions. For example, a mutually exclusive block of code can be described with three abstract positions: one modelling the demand of the mutually exclusive access to the block (i.e., the lock or the action of entering a monitor), another modelling the execution inside the mutually exclusive block and a final one, modelling the end of the block (i.e., the unlock action, or the monitor exit one). In our model, tasks can be in either one of three basic states: blocked, ready & executing. Tasks which are ready can either be safe to execute or not. Note here that the safe task predicate is not defined upon all system states.

2.1. Case Study: A Simple R-T DB

Figure 1 gives an example of an abstract control-flow graph of a simple real-time database system consisting of three tasks. In this system, the Writer task is producing a series of values (e.g., by reading some sensor) and the Refresher task is ensuring that the values the periodic User task is using are fresh enough, i.e., have been produced sometime during the last 13ms time frame. Synchronisation among the system tasks is achieved through the use of monitors, using primitives monitorEnter and monitorExit, which basically lock/unlock the mutex associated with the object they are called with. Communication among tasks is done through condition variables, using primitives wait, timed_wait, notify and notifyAll (not used in our example). Finally, the wait_for_period primitive blocks a task until its next period, which is given as an argument. In the ex-
♯Unsafe(Writer) := ♯(User = U2) ∨ ♯(User = U5) ∨ ♯(User = U0) ∨ ♯(User = U6) ∨ ♯(User = U5) ∨ ♯(User = U2_Relock) 

<table>
<thead>
<tr>
<th>IsUnsafe(Writer):=</th>
<th>(User=U0) ((Refresher=R3) (Writer=W0))</th>
<th>(User=U2) ((Refresher=R3) (Writer=W0))</th>
<th>(User=U2_Relock) ((Refresher=R3) (Writer=W0))</th>
<th>(User=U5) ((Refresher=R3) (Writer=W0))</th>
<th>(User=U6) ((Refresher=R3) (Writer=W0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsUnsafe(Refresher):=</td>
<td>(User=U0) ((Refresher=R2_Relock) (Writer=W1))</td>
<td>(User=U2) ((Refresher=R2_Relock) (Writer=W2))</td>
<td>(User=U2_Relock) ((Refresher=R2_Relock) (Writer=W2))</td>
<td>(User=U5) ((Refresher=R2_Relock) (Writer=W1))</td>
<td>(User=U6) ((Refresher=R2_Relock) (Writer=W1))</td>
</tr>
</tbody>
</table>

Table 1. Scheduler against deadlocks

Example of Figure 1, the User is the only periodic task, with a period of 20ms. Figure 2 shows how the wait primitive functions internally - it is in fact two states, the first of which atomically exits the monitor and blocks the thread waiting for a notification, while the second (i.e., R2_Relock or U2_Relock) is the state where the thread has been notified and thus wakes up and attempts to re-enter the monitor. In the following, we will use the location of the wait primitive (i.e., R2, U2) to refer to the atomic state where the task blocks, using the same location with the suffix _Relock (i.e., R2_Relock, U2_Relock) to refer to the state where the task attempts to re-enter the monitor once notified.

Table 1 shows the synthesised constraints for avoiding deadlocks in this system. Even though there are only a few constraints in this case, the problem is still obvious. One cannot easily identify which variables play the major role in establishing that a certain task will deadlock if allowed to execute, nor is it easy to extract any further information about the inter-dependencies of the tasks. Consider, however, the reformulation of the above constraints in Table 2. Immediately it becomes obvious that, as far as the Writer is concerned, the problem arises when it finds itself at the W0 location (where it wants to lock object Fresh and will subsequently ask to lock object Fresh), in states where the Refresher is at the R3 location (where it has already acquired Fresh and will next try to lock Value). Similarly, the Refresher is unsafe to execute at location R2_Relock (where it tries to acquire the object Fresh), when the Writer is at locations W1 or W2 (where it has already locked the object Value). Indeed, this situation will lead to a deadlock, since the Refresher will subsequently ask for object Value while the Writer will ask for object Fresh. It is also clearly evident that the User task has no effect on either the Writer or the Refresher tasks being safe or not. Structuring the synthesised constraints in such a manner can greatly help when their number increases. Indeed, for the case where we are trying to guarantee the deadlines of the tasks under a non-preemptive execution model (i.e., that User never misses its period), we synthesise 61 constraints in total, while the decision-tree structured constraints are only 19.

Table 2. Structured deadlock-free scheduler

3. Inducing Schedulers with ID3

Let us start with a brief introduction of Ross Quinlan’s ID3 data-mining algorithm [7]. The basic idea for structuring the scheduler constraints as a decision tree, is that we can consider the controllable states in the state-space graph of the system as a set of classified examples, where the class of a state is exactly the set of currently safe tasks. Given this and a representation of each state as the vector of the values of the observable system variables, we can use the top-down induction of decision trees (TDIDT) algorithm ID3 so as to extract the underlying structure of our scheduler constraints. ID3 performs a greedy heuristic search on the set of possible variable observations, having as a goal the minimisation of the height of the final decision tree. The skeleton of the ID3 tree-induction algorithm, as adapted to our domain, is shown in Figure 3 (adaptations are type-set using an italic bold face). Faced with a set of training examples (i.e., controllable states), we select the attribute which maximises the gain ratio, defined as the ratio of the information gain over the information value of an attribute. The main idea behind this heuristic is to try to maximise the mutual entropy of the attribute and class random variables, correcting for the fact that many-valued attributes will have a higher mutual entropy than few-valued ones. The definitions of these information-theory metrics are shown in equations (15) where $C$ is the ensemble of classes, A the ensemble of the values $v$ of an attribute, and $H(X)$ the entropy of

1 There are 45 for the Writer, 14 for the Refresher and 2 for the User. The respective trees have 11, 7 & 1 leaves, for a total of 19 constraints.
Adapted-ID3(\text{examples}, \text{attributes}, \text{attribute\_observation\_costs})

Let

\[
C-C := \text{COMPOSED} \quad \text{class of examples} \quad // \ M-C := \text{Majority class of examples}
\]

In

\begin{enumerate}
\item If all examples in class C Then
\item return a leaf labelled as C
\item Else If no more attributes Then
\item return a leaf labelled as \(C-C\) // ...labelled as \(M-C\)
\item Else
\item Select an attribute, A, for the root, \textit{among the least cost attributes}
\item For each possible value \(v_i\) of A
\item examples, := subset of examples having value \(v_i\) for A
\item Add a branch for the test \(A=v_i\)
\item If examples, is empty Then
\item create a leaf labelled as \(\text{SAFE}\) // ...labelled as \(M-C\)
\item Else
\item new subtree := Adapted-ID3(\text{examples,\,attributes - \{A\},\,attribute\_observation\_costs})
\end{enumerate}

(Adaptations are shown using an \textit{italic bold} font; original code has been commented out.)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image3}
\caption{Adapted ID3 algorithm}
\end{figure}

an ensemble \(X\) \textit{(i.e., the average Shannon information content of an outcome)}, see Quinlan \[7\] for more details.

\[
\text{Gain\_Ratio}(A, E) = \frac{\text{Info\_Gain}(A, E)}{\text{Info\_Value}(A, E)} \quad (1)
\]

\[
\text{Info\_Value}(A, E) = H(A) \quad (2)
\]

\[
\text{Info\_Gain}(A, E) = \text{Info}(E) - \text{Exp\_Info}(A, E) \quad (3)
\]

\[
\text{Info}(E) = H(C) \quad (4)
\]

\[
\text{Exp\_Info}(A, E) = -H(A) + H(C, A) \quad (5)
\]

3.1. Applying ID3 on Scheduler Constraints

Having introduced the ID3 basics, let us now describe how it can be adapted for structuring and mining our synthesised scheduler constraints. As aforementioned, the training examples we have are the states at which the scheduler is called to exert control. Our attributes are the PC vector of the tasks and the value of the global system clock, since these are the system variables we allow our scheduler to observe. The classes of the examples/states belong to the set describing which tasks are safe. Given our specific data domain, we can adapt the basic ID3 algorithm so that it uses our knowledge of the domain and thus produces better results.

3.1.1. Adapting ID3 First, we can use the fact that our classes are not independent, as is generally assumed, but can be \textit{composed}. The composed class of two examples having different classes is the most conservative of the two \textit{(i.e., the task-unsafe one)}. So, when the ID3 algorithm is faced with a set of examples of different classes which it cannot classify for lack of further observation attributes (line 8 of Figure 3), then we can take the \textit{composition} of these classes as the label of the tree leaf, instead of the most common class in the current example set. ID3 would have chosen the most common class because it considers the other examples as outliers, since it aims to increase the overall classification accuracy of the final decision-tree. In our case, these examples/states are simply a set of unsafe/safe states which the scheduler cannot distinguish in its system abstraction, not noise due to some badly performed experiment/measure. Since we aim for \textit{safety}, instead of maximising the overall classification accuracy, we must always err on the \textit{conservative side}, thus label the leaf with the composed class \textit{(i.e., unsafe)}.

Then, at line 15 of Figure 3 we do not have to label a leaf lacking examples with the most common class of its parent node, but can simply label it as \textit{SAFE}. Here again, ID3 assumed that the examples were missing because our experiments/measurements failed to cover the state space. Lacking any domain knowledge, it makes sense to assign the leaf the majority class of its parent node. However, in our domain, missing examples represent the case where we do not have any control state whose attributes have these values, \textit{i.e.,} it is an unreachable state of the system \textit{(or one where the task is not ready and thus its safety is not defined)}. Since we do not want to over-constraint the system, we explicitly assign to this leaf the \textit{SAFE} class, knowing that our decision will not have any impact on the schedulability of the system.

Attributes with an \textit{Info\_Value} \textit{(see equation 2)} equal to zero are not considered by our tree induction algorithm, since a zero value means that there is only one value for this attribute in the examples, so this attribute is adequately characterised by the attributes which have been observed higher in the decision-tree branch we are currently examining.

3.1.2. Continuous Variables In order to produce trees which are easier to understand, we have also changed the
treatment of continuous attributes, that is, the system clock.
So, instead of treating each clock value as a distinct one,
we transform them into those intervals which correspond
to an unsafe task. This value to interval transformation
is performed in the following manner. For each task we com-
pute the list of examples/states for which the task is un-
safe, sort the examples according to the values of the clock
and then construct intervals of the sorted values. For ex-
ample, if for task T the clock values at the set of exam-
ple/states for which this task is unsafe are \{0, 3, 4, 6, 7, 8\},
then we construct the intervals \([0, 0], [3, 4], [6, 8]\). Then, we
combine the intervals constructed for each task, in order
to obtain those intervals which can best characterise all
the unsafe tasks. So, if we had another task T’ whose un-
safe interval was \([4, 5]\), the combined intervals would be
\([0, 0], [3, 3], [4, 4], [5, 5], [6, 8]\), where we have broken up the
\([3, 4]\) and the \([4, 5]\) intervals, since at 3 only the first task is
unsafe, at 5 only the second task is unsafe, while at 4 both
of them are unsafe. This greatly reduces both the depth and
the breadth of the induced trees, since we avoid performing
consecutive tests/splits on specific midpoint values (which
is one way of handling continuous attributes, see \([8]\))
or splitting into as many sub-trees as possible clock values.

3.1.3. Variable Observation Costs
Finally, we have introduced observation costs for each variable
but we do not treat them as weights, which is the usual way of using them,
i.e., only as guidelines which can be ignored sometimes. In-
stead, we always choose the next attribute to observe among
the set of attributes having the smallest cost, see line 10 of
Figure 3. This allows us to postpone the observation of the
clock attribute, since this has a much larger observation cost
than the programme counters of the different tasks. Indeed,
in \([5]\) we explicitly aimed for obtaining time-independent schedulers,
effectively because of the high cost of observing the system and task execution clocks and the great decrease
of the scheduler size thus obtained. In addition, variable obser-
vation costs allow us to induce trees where we have imposed
the choice of the first attribute. In this way, we can choose a particular programme counter as the top attribute
and obtain a tree which gives us a view of the system’s be-
avouring according to that particular task.

3.2. Types of Decision Trees

Having set the framework, we now examine the two ba-
sic ways we can use our adapted ID3 algorithm. First,
we can induce separate decision trees for each task, ef-
effectively computing the predicates IS_UNSAFE(T_i),
for all \(i \leq N\), where \(N\) is the number of tasks. In this
case, we would use binary classes \{Safe, Unsafe\}
for all tasks. The second alternative is to induce a de-
cision tree for all tasks at the same time, that is, com-
pue in parallel the safe/unsafe bit-vector with one decision
tree. In this case, we will have \(2^N\) classes \([3]\). For ex-
ample, for the system of Figure 1 the classes would be \{All_-
Safe, Writer_Unsafe, Refresher_Unsafe, User_Un-
safe, Writer_Refresher_Unsafe, Writer_User_Unsafe, Refresher_User_Unsafe, Writer_-
Refresher_User_Unsafe\}. The “parallel” decision-tree
form of the scheduler obtained for the example of Figure 1
is shown in Table 3. According to this decision-tree, the sys-
tem task which seems to be most problematic for the sys-
tem’s schedulability is the Writer. It is also clear that
the User never causes the system to move towards an un-
safe (here deadlocked) state, since its programme counter
does not even appear in the tree.

In both cases, as we have already seen, we set the clock
attributes to have the highest cost, so as to observe them
last. In this way, the attribute order of trees will start with the
PCs of the tasks and, thus, we will obtain an order of
importance for the tasks of the system, with respect to our
ability to schedule the system safely. The task whose be-
haviour has the highest effect on system schedulability will
be placed first, the next one second, etc., and this will be
done separately for each branch of the decision tree.

3.3. Preprocessing of the Training Examples

Preprocessing of the training examples is another way
we can introduce our knowledge of the particular domain,
for obtaining better decision trees. In the case of the binary
class trees, the preprocessing consists of removing from the
element set all these examples which correspond to states
where the respective task is not ready, since in these states
the safety of the task is undefined (remember that a task
needs to be ready to be characterised as safe or unsafe).
This allows ID3 to use them in its convenience, as either
safe or unsafe examples, depending on which characterisa-
tion will help it construct a smaller tree. For example, the
tree for Writer in Table 2 was obtained by considering that
the missing examples W0, R3, \{U1, U3, U4\} are all un-
safe, even though in these states Writer is blocked (since
User has the lock on object Value which is needed by the
Writer). To see why, imagine having chosen the at-
tributes/values Writer/W0 and Refresher/R3. At that
point, since we have removed all examples/states where
Writer is blocked, all remaining states belong to the
Writer_Unsafe class, and the value of attribute User
does not offer any further increase in the classification pre-
cision of the decision-tree. Thus, ID3 will choose to directly
construct a leaf labelled as Writer_Unsafe, executing
line 6 of Figure 3.

\[^2\text{In reality, there will be }2^{N+1}\text{ classes, since we also have the IDLE task in timed models of the system and the scheduler needs to decide at each control point whether idling is safe or not.}\]
3.3.1. Preprocessing Examples for “Parallel” Trees

Unfortunately, we cannot do the same kind of preprocessing for the tree with the $2^N$ classes, which attempts to compute in parallel the $N$ different safety predicates. This is because a state may well contain some tasks which are not ready (so their safety is undefined) and some tasks which are unsafe. Removing these states from the training examples could make us take a wrong decision where the scheduler considers certain unsafe tasks as safe to execute, with possibly catastrophic consequences. Therefore, in this case the preprocessing of the examples is performed in the following manner. For each example we use the $N$ binary-class trees to classify it as either safe or unsafe for each task and then use the composition of these classifications as its global class. This allows us to classify certain states where a task is blocked as a state where this task is safe, while other blocked states are classified as unsafe for that task. This different treatment of non-ready tasks helps in obtaining a smaller $2^N$-classes “parallel” tree, than we would have obtained if we considered all non-ready tasks to always be either safe or unsafe. Therefore, the binary-class trees help us in finding a convenient characterisation for non-ready tasks, whose class (i.e., safety) is undefined.

4. System Analysis & Optimisation

The $N$ binary-class decision trees allow us to look at the system from the view-point of the particular task, i.e., find out, for each of its possible locations, what is the task that has the highest effect on its execution being safe or not. This knowledge can help the designer find strong inter-dependencies among tasks and thus construct sets of strongly dependent tasks. These task sets can then be used for selecting distribution of tasks to processing units, e.g., placing dependent tasks together on the same processing unit in order to avoid an increased inter-processor communication for their synchronisation. For example, by examining the trees in Table 2 we can see that Writer and Refresher are strongly dependent upon each other and thus it might be better to place them on the same processing unit, while User could be placed in another one. They can also be used for highlighting potential design problems. Imagine, for example, the case where a highly critical task is strongly dependent on a low criticality task; it is evident that we would like to re-design the system so as to decrease such a dependency or avoid it altogether.

Another observation we can make by examining in Table 3 the (partial) decision-tree of the scheduler against deadlock-misses in a non-preemptive model, is that the most important task is the Refresher, not the User as we might have expected. Indeed, this is logical since it is the Refresher which in the end will allow the User to advance from waiting for a fresh sensor value. We also see that when the User, the Refresher and the Writer all compete for the same resource (i.e., Value) at locations $R2_{lock}, W0$ and $U0$ or $U2_{lock}$, then the scheduler gives User the priority, so that it does not miss its deadline.

4.1. Counterexample Analysis

By data-mining the synthesised scheduler, we effectively get a succinct view of all the counterexamples for the property we want to guarantee, highlighting just the first states where the system execution went wrong. Table 3 summarises all the system execution counterexample traces reaching a deadlock state in our case study, that is the counterexamples we would obtain from a model-checker if checking our system for deadlock-freedom. Table 3 tells us that the root of the erroneous behaviour in these counterexample traces is the states where the system finds itself in one of the configurations $(*, W0, *, R3, *), (*, W1, *, R2_{lock}, *), (*, W2, *, R2_{lock}, *)$. In this manner, it clearly specifies the tasks which initiate an erroneous behaviour, showing us where we should focus our attention, as well as, from what moment/state on the erroneous behaviour starts. Therefore, unlike current practise, we do not have to simulate and study the execution of the system at all the states of a counterexample trace leading to the problem. This is a particularly interesting feature when the erroneous behaviour needs a great number of steps before it becomes possi-

<table>
<thead>
<tr>
<th>Attribute: Writer [Gain-Ratio : 0.16776516]</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt; W0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>--&gt; W1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>--&gt; W2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 3. Scheduler against deadlocks, structured as a “parallel” decision tree
ble \footnote{One can imagine “watchpointing” a counterexample simulation, instead of manually stepping over each state.} It also allows us to “group” all the counterexample traces one can obtain, into a small set of the root problematic states. By keeping just the variables which are important in the decision tree, data-mining also allows us to identify and subsequently ignore the tasks which are not really part of the problem when examining a counterexample (as is the case for the User in our example). Thus, the combination of scheduler synthesis and data-mining can also be used to more easily understand the results of a model-checker. Instead of model-checking against a property and examining all the different counterexample traces trying to figure out when & why things started going wrong, one can synthesise a scheduler which guarantees and use this synthesised scheduler as a starting point for understanding the reasons for which does not hold in the system.

4.2. Optimising Scheduler Implementation

Obtaining \( N \) decision trees can also be useful for the implementation of the synthesised scheduler. Normally, we implement the scheduler as a large function (containing the “parallel” decision tree), which the application tasks call at their control points to compute the next task to execute. Now, we can implement it as small, embedded code at the control points of each task, effectively copying there the sub-tree for the particular task position, as extracted from that task’s “personal” decision tree. An evident optimisation we can do with this distributed scheduler is at the control points where we know that a task is safe to execute; there we can remove the scheduling code altogether, effectively giving it priority over other tasks. In our case study, the only point in the Writer where we would need to ask our synthesised scheduler the permission to proceed so as to avoid deadlocks is the \( W_0 \) location and the only ones where Refresher should ask for permission are the \( R_3 \) and \( R_2_{\text{Relock}} \) ones. This means that we can avoid the scheduling decisions for deadlock avoidance at the locations \( W_2, W_5, W_6, R_0, R_6, R_7, U_0, U_2_{\text{Relock}} \) and \( U_4 \). That is, we can remove 9 out of the 12 control points of our system, since there we do not need to execute any particular synchronisation protocol – the simple action of locking/unlocking the mutex which is performed at these locations suffices to keep the system deadlock-free.

5. Pruning Decision Trees

Once we have synthesised a scheduler for safely controlling the system, we might be interested in exploring the case where we do not allow this scheduler to observe certain system variables, for example the clocks. This way, we can obtain a smaller, more robust, time-independent scheduler, which is also easier to implement and faster to execute (no need to use timers, etc.), see \footnote{Indeed, it suffices to examine the composition of the classes labelling the leaves which are reachable from the sub-trees corresponding to the clock attribute in the “parallel” decision tree. If the composed class assigns at least one application task to the safe class, then we can safely prune the tree at that point and replace all the sub-tree starting from the observation on the clock by a leaf labelled with the composed class, without needing to validate the new, pruned scheduler. As an example, let us consider a case where when testing on the global clock, we obtain 3 leaves with classes Writer,Refresher,Unsafe, Refresher_-Unsafe, and Writer,Unsafe. The composed class being Writer,Refresher,Unsafe, we can substitute the test on the global clock and the subsequent 3 leaves, with a single leaf labelled as Writer,Refresher,Unsafe, i.e., allow only the User task to execute (even though the Writer and the Refresher tasks could have being safely executed for certain values of the global clock).

In this manner, we can see if we need to do the synthesis again. Apparently, this is only needed if at some point the composed class turns out to be All,Unsafe. Note that when the composed class of some attribute’s sub-tree is All,Unsafe it doesn’t mean that pruning will render the system non-schedulable. This is because, it is possible that we can synthesise further scheduling constraints for the system, or that we have already rendered the newly problematic states unreachable, as a side-effect of our having pruned this attribute away at some other node of the tree.

We can continue applying this decision-tree pruning until we obtain a decision-tree/scheduler which is as small as we want. Since the decision tree has placed the least important variables last, we have a clear indication of which variables should be candidates for pruning - always start pruning from the leaves, moving towards the root of the tree.

5.1. Pitfall: Pruning & Blocked Tasks

We should note here that our argument holds only if while synthesising the scheduler we had allowed the system to idle even at states where there existed safe tasks to
execute. In the opposite case, the pruned decision tree of the scheduler may well forbid certain tasks to execute once they reach a certain location, because we are essentially forcing them to idle and this may not be safe. In this case, one must make sure that the tree ensures tasks’ liveness. That is, at each leaf the tasks which are classified as unsafe do not contain all the ready tasks which have been used for deciding upon the class of the current state, thus ensuring that some task will execute and change the current state. Let us consider an example. If we synthesise a scheduler to avoid deadline-misses for our case study under a non-preemptive execution model and always execute one of the safe tasks (i.e., idling is allowed only when we have no safe choice), the decision tree of the scheduler constraints will contain the sub-tree shown in Table 4(a). The pruned decision sub-tree shown in Table 4(b) is evidently correct, because once User changes location we will leave this sub-tree and thus allow Writer & Refresher to execute again. If, however, we prune even further (e.g., remove the observation of User’s location, see Table 4(c)) then this sub-tree will force Refresher and Writer to stop executing forever, once they reach their respective locations $R7$ and $W0$, which will eventually cause User itself to miss its deadline. The difference between the two cases is that in $[b]$ the set of tasks classified as unsafe $\{\text{Writer, Refresher}\}$ is a true subset of the set of ready tasks used for the classification $\{\text{Refresher, Writer, User}\}$, while in $[c]$ this is no longer the case. If we had allowed the system to idle even when there are other safe choices, then the scheduler (and therefore its corresponding decision tree) would have also included the situations under which it is not safe to idle and, in that case, pruning at User as was done in Table 4(c) would have resulted in an $\text{AllUnsafe}$ class.

### 5.2. Tree Pruning and Schedulability

As we have seen in the previous discussion, the tree induction process will optimise the representation of the constraints by making decisions only on the variables which can indeed change the class of a task and selecting them in an order which reflects their importance for the system’s schedulability. For example, this allowed us to minimise the 15 constraints used for deadlock-freedom in our case study to just 3, and the 61 constraints used for respecting deadlines under a non-preemptive execution model, to just 19\[4\] Note, however, that the set of the important variables for the class of tasks is computed with respect to the currently available training set.

### 5.2.1. When to Prune?

When we move from a constrained system, e.g., one where the system uses a non-preemptive execution model, to a less constrained one, 

---

4 Constraints and respective trees not shown due to lack of space.
scheduler. Once we have performed the synthesis for the most refined model of our system and the additional quality aspects we wish to meet, we can use the decision-trees for implementing the scheduler as well, without fear of over-constraining the system.

During analysis, we could use more “aggressive” pruning algorithms, like ID3’s successor C4.5 \[5\]. C4.5 uses the gain-ratio attribute selection criterion as we do but prunes the induced tree in a way which, even though not safe for our domain, can nevertheless help to better highlight the major problematic situations.

6. Conclusions

In \[5\] we presented a methodology for performing scheduler synthesis for hard real-time systems, which allows a designer to obtain fine-grain scheduling constraints corresponding to different safety properties (e.g., deadlock or deadline-miss avoidance), different underlying assumptions concerning the system execution model (e.g., non-preemptive execution versus preemptive execution) or different assumptions about the scheduler (e.g., ability to observe the system clocks or lack of it). In this paper, we have presented a method, based on a classic Machine Learning algorithm for data-mining (ID3 \[7\]), which allows to extract the underlying structure of the scheduler constraints synthesised with our methodology. Since these synthesised constraints are usually too numerous and unstructured, their reformulation is imperative for being able to understand and validate them.

We have shown how one can adapt ID3 to the task of structuring synthesised schedulers for safety properties and how one can use it to look at the controllability of the system from the viewpoint of each task comprising it, as well as, from a global, system-wide view. This allows to more easily identify the tasks which have the highest impact upon the controllability of the system, to which the designer should focus his attention, as well as, identify undesired dependencies between critical and non-critical tasks. Another information we can more easily derive from the decision-tree form of the synthesised schedulers is the classes of strongly interdependent tasks, which would be good candidates for being mapped to the same processing unit, if the system is distributed. In addition to this kind of information, we showed how synthesised schedulers (especially in their succinct decision-tree form) can be used for explaining the (usually too numerous) counterexamples of a model-checker, directly pinpointing the (usually much fewer) states where the problematic behaviours become possible.

We have also shown how to decrease the number of constraints needed to implement a safe scheduler by carefully pruning the decision-tree, so as to obtain a scheduler which is smaller and faster to execute.

Finally, we showed how a safe scheduler can be distributed through the different tasks and be directly embedded in their code, possibly eliminating altogether the scheduling protocol where this is not needed, as a further optimisation.

The only other work we are aware of where data-mining (indeed ID3) was used in a model-checking context is the work of Edmund Clarke et al. \[3\]. There, the authors used ID3 for discovering the model variables which could better dissociate a number of concrete states belonging to the same abstract state, in order to check whether a counterexample obtained in an abstracted system is also present in the more concrete one or whether it is a spurious one, caused by a coarse abstraction. However, this is the first time that data mining has been applied in a scheduler synthesis context, to the best of our knowledge.

Acknowledgements: The implementation of the adapted tree induction method (ID3) presented herein was based on code written by Raymond Joseph Mooney \[6\].

References


5 See J48 in \[http://www.cs.waikato.ac.nz/˜ml/weka/\]

6 Original code: \[www.cs.utexas.edu/users/ml/ml-progs.html\]