City Research Online

City, University of London Institutional Repository

Thunderstriking Constraints with JUPITER

Christos Kloukinas
City University
Department of Computing
Northampton Sq., London EC1V 0HB, U.K.
C.Kloukinas(at)soi.city.ac.uk

Abstract

We present JUPITER, *a tool for analysing multi-constrained systems.* JUPITER *was built to explore three basic ideas. First, how to use controller synthesis so as to find the exact conditions under which a particular constraint will be satisfied. Second, how to successively refine the models used for the controller synthesis so as to obtain a series of more easily understandable and more robust controllers. Last but not least, how to structure & explain the synthesised controllers and provide hints to designers for further optimisations through the use of Machine Learning techniques.*

Thus, JUPITER *can help in the design and analysis of multi-constraint systems through the automatic synthesis of control logic for certain of the constraints and the aid it provides to designers for discovering further optimisations.*

The controllers it synthesises can be easily implemented on top of a standard real-time OS [7, 6].

Keywords: *controller synthesis, refinement methodology, machine learning.*

1. Introduction

The complexity of the systems we are called to build is increasing each day and so does the number of properties we desire to guarantee: functional, real-time, performance, response jitter, memory, energy consumption, QoS, dependability, *etc.* Unfortunately, we are still lacking the tools and methodologies which will allow us to ease the design and development of such systems. The obvious solution to this profusion of constraints is to attack each one in turn, refining our models successively. Indeed, it is evident that one cannot attack all the constraints at once, since the resulting search space is enormous. As we will see later on, treating each constraint in its turn also helps better understand them and build better systems.

In this paper we present JUPITER, a tool for aiding engineers of embedded systems to analyse and meet the con-straints of their systems. We mainly deal with *safety* properties, not *liveness* ones. Indeed, in the setting of real-time embedded systems for which JUPITER was developed, one is not usually asking that an event will be "eventually" answered but rather that it will be answered in a particular time frame. JUPITER was developed so as to explore three basic ideas:

- Controller Synthesis [11, 1] : Simply trying to verify some property is not enough; indeed, more often than not we know beforehand that the property will not hold in general, especially for non-functional properties (*e.g.*, hard real-time deadlines). Using controller synthesis, we instead try to find the conditions we must fulfil in order for the property to become true for the system, *e.g.*, in the case of hard real-time deadlines, find a scheduling of tasks.

- Refinement Methodology [7, 6] : The refinement methodology we propose is based on the usual wish to consider easier properties first, gradually enriching the system model so as to prove more difficult ones. In this way, we increase our chances of being able to attack realistically sized problems, at the expense of slightly over-constraining the system, a small price to pay usually.

  More importantly, our methodology aims at aiding engineers break down the analysis of a system, so as to understand better its behaviour and the conditions under which each particular requirement can be met.

- Analysis of Results using Machine Learning [5] : Probably the most innovative[1] part of JUPITER, it stems from our realisation that practitioners will need to understand the multiple controllers we synthesise using our refinement methodology. This is a direct consequence of their need to *validate* the final controller, as well as, to study how other, more compli-

---

[1]Or just bizarre according to some. . .

cated constraints could be introduced to the system, how the system could be further optimised, *etc.*

The rest of the paper is structured as follows: we first present the design of the controller synthesiser of JUPITER and the improvements we introduced from earlier versions of it [7, 6]. Then we present the refinement methodology we are using and, following that, we show how JUPITER helps in the analysis of the synthesised controllers. We then move to a concrete example, showing how one can use JUPITER to control and analyse a particular system. Finally, we finish with the conclusions and the work which will need to be done in the future.

2. Controller Synthesis

Generally speaking, controller synthesis [11] can be thought of as a natural next step to model-checking. Just as in model-checking, we need to build the state space and find the bad states (*i.e.*, where the desirable properties do not hold). Once the model-checking part has completed, we need to backtrack from the bad states, until we can find a set of states from whence we can render the bad states unreachable by disabling certain system actions. For example, when trying to assure deadlock-freedom, we need to find the set of states where, if we disable the execution of certain tasks, the system can no longer reach a deadlocked state.

Apart from the backtracking part, the basic difference with model checking lies in the fact that now the actions/transitions of the various automata modelling the system are divided into two sets: the set of *controllable* transitions and the set of *uncontrollable* transitions. Thus, the foremost decision to make when desiring to apply controller synthesis is to decide which are the controllable actions in the system. There are different possibilities: one can control only the election of the currently executing task (*e.g.*, run task X when at state Y) or also control the duration of its execution (*e.g.*, run task X when at state Y for a duration not longer than Z seconds). Other variables we can control is the exact time when we will elect some ready task for execution (*e.g.*, delaying it on purpose so as to minimise response jitter), the specific processor which should execute a task in a multi-processor system, the current energy consumption if so permitted by the underlying hardware (*e.g.*, by underclocking/undervolting the processor), *etc.*

Just like model-checking, controller synthesis can be performed in a symbolic manner (*e.g.*, using BDD's [3] for representing the states in the state space), in an on-the-fly manner [9], in a forwards/backwards manner, *etc.* Given that we are interested in a multitude of constraints, we need to search for the maximal controller of our system's model, so as not to over-constrain ourselves. In this way, we can

then still have our options open for controlling other aspects of the system. Therefore, it does not suffice to find just one cyclic behaviour which stays in a set of safe states. Take for example the real-time controllers derived through the classic rate-monotonic analysis [8]. These impose a fixed priority on the tasks and thus make it extremely difficult to examine further constraints for performance, jitter, *etc.* [2]

Currently, JUPITER has a hard-coded definition of what constitutes a controllable action: the election of a system task/automaton for execution. The controller synthesis is then performed as described above; the full state space is constructed first, using an explicit state representation. Then the synthesis phase starts, backtracking from the bad states until it finds the set of frontier states, which separate the set of safe states from the set of unsafe states [7, 6]. This set of frontier states, along with the constraints imposed on each state in it, is our (maximal) controller. The models that JUPITER works with have a structure inherited from the fact that it was initially meant to analyse real-time Java programs [6]. So it assumes that there is an automaton for each of the concurrently executing system tasks, one automaton for modelling the environment (firing alarms & keeping track of time by advancing the discrete-time clocks) and another automaton which models the system scheduler. The actions of the scheduler are controllable, while the actions of the other automata are uncontrollable. The scheduler becomes enabled when one of the application automata try to enter a monitor on a shared resource or when an alarm is fired. It then calculates the set of *ready* tasks and those among them which it considers as *safe* to execute, so as to pick a safe task for execution non-deterministically (it assumes a uni-processor system). Starting initially with the set of safe tasks equal to the set of ready ones, JUPITER uses the synthesised constraints to successively strengthen its (current state based) definition of safe tasks until all bad states become unreachable.

We can think of our synthesised controller as a table-driven/rule-based *scheduler*; the table index/if-part of the rule is the current state and the table content/then-part of the rule is the set of tasks/automata which must not be elected to run at the particular state so as to keep the system safe. We can also see the resulting controller as a *description/explanation* of all the undesirable behaviours of the system. Being maximal, the controller will delay disabling some action (task execution) as much as possible, thus coming as close as possible to the root of an undesirable system behaviour. This means that we can use the controller as a succinct description of the numerous (often infinite) counterexample traces produced by a normal model-checker, which highlights the *cause* of an undesirable behaviour.

---

[2]Rate-monotonic analysis does, however, have the advantage of being an extremely simple and fast analysis method.

Equally important to the characterisation of controllable/uncontrollable actions is the characterisation of observable/unobservable state variables from the controller's/scheduler's point of view. We partially support this in JUPITER by assuming that the only observable system variables are the ones describing the current states of the automata (*i.e.*, what in a program would be the current program counter) plus the current value of the system clocks. These latter are *discrete-time* stopwatches [4], chosen for their ability to model preemptive scheduling of tasks. Thus, the controller is unable to observe the rest of the system variables; in fact, as we shall see later, we also examine the case where the controller cannot observe the values of the system clocks either, so as to examine whether we can construct a *time-independent controller*.

Compressing Periodic Clocks. A simple optimisation for timed models is the compression of all the *periodic* clocks, c_i. When all of the periodic clocks start at time instance 0, then we can replace them with a single periodic clock c taking values in the interval $[0, P)$, where P is the *hyper-period* of the system. The hyper-period of the system is defined as the least common multiplier of the periods, P_i, of the periodic tasks, $P = \text{lcm}_i(P_i)$. This is because at time instance P the periodic clocks we are representing using the single clock c will have reached for the first time the same configuration they had at time instance 0.

Compressing Periodic Clocks with Phases. Initially [7, 6] we refrained from dealing with periodic clocks having non-zero phases, asking for all application tasks to start at time instance 0. We discovered however that dealing with this case is not as difficult as we initially believed. So now JUPITER also compresses the periodic clocks whose phase/start time, S_i, is non-zero. Let us now assume that the periodic clocks c_i take values in the intervals $[0, S_i + P_i)$ with S_i being the *phase/start time* of the clock. This effectively means that our single periodic clock will be taking values in the interval: $[0, S + P)$, where $S = \max_i(S_i)$. Indeed, time instance S is the first time all the periodic clocks we are compressing will have entered their periodic behaviour, and this configuration will be repeated every P time units. That is, whatever was their phase at time S, it will be exactly the same P time units later on.

This compression means that instead of having to use $\sum_i \lceil \log_2(S_i + P_i) \rceil$ bits to represent all the periodic clocks of a system state, we need only use $\lceil \log_2(P + S) \rceil$ bits, a reduction which can help a lot as the number of states increases (especially in the case where we allow task preemption).

To keep the models simple, we introduce macros for the original periodic clocks, c_i, which are now defined as:

$$c_i = \begin{cases} c & \text{when } c < S_i \\ (c - S_i) \bmod P_i & \text{when } c \geq S_i \end{cases} \qquad (1)$$

The system clock c itself "advances" using the following tick function, thus ensuring that once it has taken the value S it will forever remain in the interval $[S, S + P)$:

$$\text{tick(c)} = \begin{cases} c + 1 & \text{when } c < S \\ [(c + 1 - S) \bmod P] + S & \text{when } c \geq S \end{cases} \qquad (2)$$

State Reduction. The most important optimisation however over its previous versions [7, 6] is how JUPITER decreases the number of states which must be saved in memory. Currently, JUPITER computes for each controllable state (*i.e.*, a state where there's at least one controllable action enabled) its *controllable future*. That is, for each possible action α it explores all the uncontrollable states which can be reached from the current controllable state, until it finds these controllable states which are reachable through only uncontrollable states. It then discards all the intermediate uncontrollable states and keeps just the reachable controllable states, thus constructing a *hyper-graph*. So, when taking an action α from a controllable state s_i^C to an uncontrollable state s_j^U we will eventually reach a set of controllable states $\{k = 1 \dots n : s_{i_k}^C\}$, where the system scheduler becomes enabled anew or some bad state is reached (we consider these as controllable states by definition). By the very nature of the models, where application tasks either compute for a finite duration or ask for resources by attempting to enter a monitor, a controllable (or bad state) is bound to be reached eventually from any uncontrollable state. So, instead of storing all the states, we discard state s_j^U and all its uncontrollable successors, keeping just the fact that $s_i^C \xrightarrow{\alpha} \{k = 1 \dots n : s_{i_k}^C\}$. By defining all bad states to be controllable, this optimisation is safe, since the states which we remove do not have any repercussion on our ability to control/observe the system state, nor do they modify its branching structure. This is because in our models, the controller/scheduler does not permit any other task to execute when it is taking an action; this is similar to an OS kernel inhibiting interrupts while scheduling. Thus, each state in the state space will have only one kind of actions, either controllable or uncontrollable, making our reduction safe to apply.

Unlike the branching bisimulation equivalence [10] we were using in our previous work [7, 6], which needed to construct the full state space before reducing it, this optimisation (dismissal of all intermediate uncontrollable states) can be applied in an on-the-fly manner and thus lead to a substantial reduction of the memory needs for exploring the state space. Indeed, now we only have to store the current "window" of uncontrollable state-space, which we discard as soon as we have covered it completely, instead of keeping it until the end. This is of particular importance for the case where we are examining the timed model of the system, since then the uncontrollable states can be a quite

lengthy series of time steps/ticks, which can quickly cause a state space explosion if we do not discard them. It also helps in the synthesis phase, since now we only consider the controllable states when backtracking.

Using these optimisations, JUPITER can be used to synthesise controllers for any kind of real-time uni-processor systems, where actions have a duration described using minimum and maximum execution durations, *e.g.*, $s_i \xrightarrow{\alpha; w_\alpha \in [B_\alpha, W_\alpha]} s_j$, where α is an action, w_α a stopwatch measuring the *pure execution* duration of action α and B_α (W_α) the best (resp. worst) case execution duration of action α. References [7, 6] describe our modelling of multi-task, preemptive real-time systems in more detail.

3. Refinement Methodology

As aforementioned, the refinement methodology we are using with JUPITER attempts to construct a series of controllers, each one able to guarantee one more of the many system requirements. This methodology is not used solely for being able to attack more realistically sized systems; it is also a software engineering attempt to break down the analysis of a system in order to better understand its behaviour and the conditions under which the various systems requirements are met.

3.1. Deadlock-Freedom

The order in which we consider these requirements in our methodology depends on both their difficulty and certain, more practical, engineering concerns. The first requirement we try to guarantee by synthesising a controller for it is deadlock-freedom. We do this using an untimed model of the system, *i.e.*, one where actions have an unspecified duration. We only try to guarantee other requirements, such as meeting deadlines, once we have managed to synthesise a controller for deadlock-freedom and we have used it to constrain our system.

The first reason for following this approach is because untimed models are in general much easier to analyse than timed ones. In addition, the synthesised controller for deadlock-freedom helps us in the analysis of the timed model of the system by constraining it into the non-deadlocked states. At the same time, it is also a sensible engineering approach to try to remove all deadlocks for all possible time durations of the various system actions. This is because, the durations of the system actions is in many cases nothing more than simple *guesstimates*. They can thus *hide* potential deadlocks which will resurface later on. Even, however, in the case where action durations are safely bounded by their declared best and worst values, they are not constant over the system's life-time. Indeed, it may

well be the case that hardware or software components of the system will be replaced in the future, thus changing the timing relations. This may possibly unmask *dormant* deadlocks which were so far avoided only by chance, because of the particular initial action durations. Trying to debug such a problem can be daunting; for critical systems such a situation is simply unacceptable due to the great danger/cost it implies.

3.2. Guaranteeing Deadlines

Once our system has been rendered deadlock-free, our methodology examines how it is possible to guarantee deadlines in a timed model of this *constrained* system, under the hypothesis that task actions *cannot be preempted*. The non-preemptive execution model hypothesis helps us to examine a reduced state space first for the timed model, since it removes all the cases where the execution of a task action is suspended so as to handle an action of some other task. Disallowing preemption of actions effectively transforms our stopwatches back to *discrete timed automata* [2].

Once we can safely control the system under the hypothesis that tasks are never preempted when performing some computation, then we can use the constraints obtained during this step to *reduce even further* the state space that we have to construct and analyse, when we do allow tasks to be preempted. We should note here that there are certain systems where the execution model is by nature non-preemptive (*e.g.*, network messages cannot be "preempted" by other messages - at least not without a big cost). In these systems, it makes no sense to examine the preemptive execution model.

We should also note here that we cannot safely control all systems when we do not allow tasks to be preempted. This means that for these systems we will not obtain any control constraints and, therefore, will be obliged to examine the larger, unconstrained state space of the timed model, which corresponds to a (deadlock-free) preemptive execution model. Even in such a case, where we find that the system cannot be safely controlled under a non-preemptive execution model, we can nevertheless obtain some useful information about the system. Indeed, such a failure means that the system is *overloaded* and/or that there are *exceptionally small deadlines* (compared to the duration of the computations). In such a case, the coontroller synthesis method will provide us with the tasks which miss their deadlines under a non-preemptive execution model, as well as, with traces showing what other tasks caused them to miss their deadlines, which could help us in devising a *partially preemptive* model, where some tasks can be preempted, while others cannot. Another possibility to remedy our inability to synthesise a controller for a non-preemptive execution model is to break up computations, thus manually

introducing *explicit preemption points* in the application.

Meeting Deadlines under Preemption. Having already constrained the system for staying deadlock-free and meeting the deadlines under a non-preemptive execution model, we are in a better position to explore the conditions under which the system can meet deadlines under a preemptive execution model. If non-preemption proved indeed to be controllable, then so will preemption; in the worst case the synthesised controller will have to re-impose non-preemption.

The benefits of being able to support a preemptive execution model are quite obvious; we can better utilise resources and increase our chances for meeting further constraints (*e.g.*, performance, jitter, *etc.*).

3.3. Quality of Service

Quality of Service (QoS) effectively introduces levels of service quality paired to resource needs. When the resources are available, the system is able to offer certain levels of QoS and guarantee that it will meet them.

Each particular QoS level effectively introduces a new system. Action durations change (*e.g.*, the processor now uses more energy thus speeding up its execution) and so do deadlines (*e.g.*, we may now wish for shorter/longer response times). This means that the real-time analysis of the system must be done separately for each different QoS level. If deadlines do not change, then we can analyse all QoS levels together by enlarging the action duration intervals to cover both the fastest and the slowest QoS. However, this will over-constrain the system, imposing constraints on certain QoS levels which are only needed at others.

3.4. Other Constraints

As shown in our previous work [7, 6], once we have synthesised a controller for real-time deadlines, we can investigate whether we can synthesise a time-independent controller, that is, a controller which does not observe any system clocks. This will help make the implementation of the controller faster and simpler, since we will no longer have to use clocks in the final system (at least for the controller). It will usually also decrease the size of the controller because the controller synthesiser will now be forced to synthesise more coarse constraints. The analysis technique described in the following section will show more ways one can optimise a synthesised controller.

The constraints we have so far described have stayed within the well studied properties of deadlock-freedom and real-time requirements. Other constraints, such as controlling energy consumption (either to minimise it or to stabilise it for security reasons) or memory consumption, have not yet been studied as much and we need to gain more experience with them.

Nevertheless, it seems logical to try to analyse the memory usage patterns first and control the system tasks in order to minimise the total memory needed (*e.g.*, by giving priority to tasks which will free memory over those which will allocate even more). Having constrained the system using all these rules, then we can try to analyse energy consumption by first taking into account only the energy consumption of the processor. Once this step is performed and a controller has been synthesised for it, one can refine the model even further, introducing the energy consumption of the memory units as well.

The resulting controller/scheduler stack (where each stack layer is responsible for a specific property) need not be deterministic. As we showed in [7] we can add further layers to this stack to render our scheduler deterministic by explicitly giving precedence to certain tasks over others, when all of them are safe.

4. Analysis of Controllers

As aforementioned, through controller synthesis and our refinement methodology, JUPITER will construct a stack of controllers. Each one of them will be guaranteeing a particular constraint: deadlock-freedom, deadlines under non-preemption, deadlines under preemption, *etc.* Even though this separation of control logic helps in better understanding both the control conditions and the system itself, synthesised controllers are usually difficult to study, just like any other computer generated piece of code (*e.g.*, YACC parsers). This is a grave problem because, most often than not, engineers need to go further: either *validate* the final controller or study it to figure out how they could meet even more constraints (*e.g.*, performance, distribution, *etc.*). In both cases, *understanding* the controller is of paramount importance.

In their present form, the synthesised controllers lack two basic properties which render them difficult to understand: *structure* and *importance* of the control conditions. As aforementioned, the synthesised controllers' form is a set of rules on the current state vector (the observable part of it) with associated actions to perform/avoid. A much better way to present these controllers is by turning them into decision trees [5]. This structures the set of rules together, showing which state variables are more important for the purposes of controlling the system and which are not.

One can think of this process as an attempt to *summarise/compress* the information present in the controllers. This is different to using a BDD representation; there we're effectively obtaining a *canonical* representation, not a minimum one. Indeed, the problem of obtaining a *small* BDD lies in choosing a proper variable ordering. The fact that
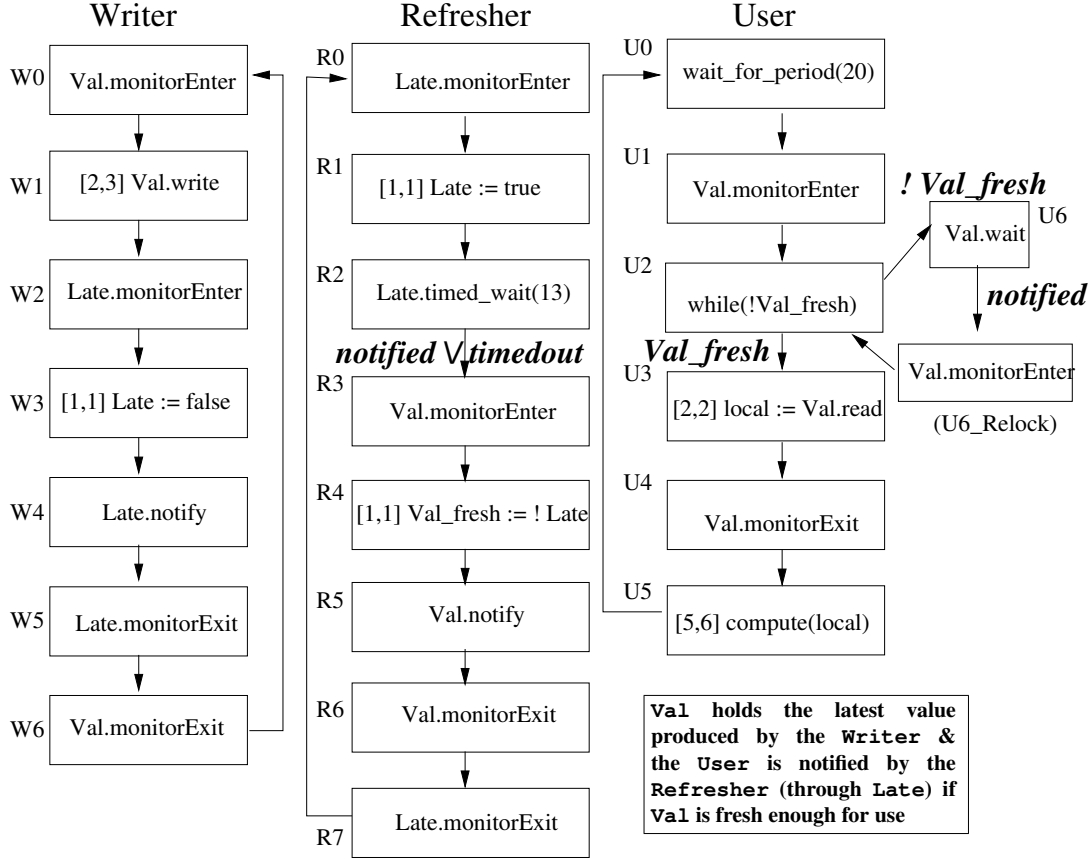
Writer

W0	Val.monitorEnter
W1	[2,3] Val.write
W2	Late.monitorEnter
W3	[1,1] Late := false
W4	Late.notify
W5	Late.monitorExit
W6	Val.monitorExit

Refresher

R0	Late.monitorEnter
R1	[1,1] Late := true
R2	Late.timed_wait(13)

notified ∨ *timedout*

R3	Val.monitorEnter
R4	[1,1] Val_fresh := ! Late
R5	Val.notify
R6	Val.monitorExit
R7	Late.monitorExit

User

U0	wait_for_period(20)
U1	Val.monitorEnter
U2	while(!Val_fresh)

! Val_fresh

| U6 | Val.wait |

notified

Val.monitorEnter
(U6_Relock)

Val_fresh

U3	[2,2] local := Val.read
U4	Val.monitorExit
U5	[5,6] compute(local)

Val holds the latest value produced by the **Writer** & the **User** is notified by the **Refresher** (through **Late**) if **Val** is fresh enough for use

Figure 1. A simple R-T system

BDD's impose the same ordering throughout the whole structure makes it difficult to obtain the smallest description possible and in any case BDD's give us very little, if any, information concerning the *importance* of system variables. We can instead use Machine Learning methods for inducing a decision tree out of a set of pre-classified examples, so as to extract a succinct description of the example set, as proposed in [5]. Indeed, our setting matches the decision tree induction methods perfectly. Our example set is the (controllable) state space itself. Through the controllers each state is effectively classified as safe or unsafe for some action. Thus, we can induce a decision tree form of the controllers to characterise the states as unsafe/safe.

Unlike the usual application of Machine Learning methods, we have a great advantage: full coverage of the state space. Thus the trees we are inducing are by definition 100% accurate, a near impossible situation in the Machine Learning domain, where example sets used for induction/training cover only a part of the state space, contain measurement errors, *etc.* This means that we can change certain aspects of the Machine Learning algorithms which were introduced because of the underlying uncertainty in the quality of the training examples [5]. In this way, we can take full advantage of the fact that we cover the full state space and that all our data are accurate.

Just like BDD's, the decision tree induction algorithm we use will remove unneeded variables from the decision tree, showing us just the ones which really matter for the controllability of the system. Unlike BDD's, however, the decision tree induction algorithm will also offer an *order of significance* of these variables, helping us understand which system variables are more important. For example, when using this approach to structure the synthesised controller against deadlocks, this ordering will give us such information as which task is most important for another task to avoid deadlock, highlight task interdependencies which can aid in finding good task partitioning to multi-processors, *etc.* [5].

5. Example

In this section we will demonstrate the use of JUPITER in practise through an example, see Figure 1. Our example consists of a very simple system composed of three tasks:

an aperiodic task (`Writer`) which continually reads some sensor and writes its value into a shared variable (`Val`), a periodic task (`User`) which uses this value to perform some task and a third aperiodic task (`Refresher`) which attempts to ensure that values used by the `User` task have been produced in the last 13 time units. The three tasks use monitors and condition variables to synchronise and communicate, as in Java programs. Certain actions are prefixed with an interval, *e.g.*, "`[2, 3] Val.write`" which describes the minimum and maximum execution time this action may take. Initially, all actions are assumed to be safe, so the scheduler simply chooses non-deterministically among the application tasks which are *ready* to execute.

```
Writer-Is-Unsafe :=
  (and (= Writer |W0|)
          (= Refresher |R3|))

Refresher-Is-Unsafe :=
  (and (= Refresher |R2_Relock|)
          (or (= Writer |W1|)
              (= Writer |W2|)))
```

Figure 2. Controller against deadlocks

If we analyse this system, we will discover that there is the possibility of a deadlock between the `Writer` and `Refresher` tasks. To solve this deadlock, JUPITER synthesises a controller which effectively removes 12 controllable transitions from the state-space. The decision tree form of this controller is the one shown in Figure 2. Indeed, it is easy to see that if we allow the `Writer` to execute at position `W0` when the `Refresher` is at `R3` the system will deadlock, since `Refresher` has already entered the monitor of `Late` and will subsequently need to enter `Val`'s monitor, while `Writer` after executing the `W0 → W1` action will have entered `Val`'s monitor and will then attempt to enter `Late`'s monitor. The constraint on `Refresher` is also easy to understand. It says that whenever `Refresher` is at position `R2_Relock`, which is the internal position between `R2` and `R3` where the task has been notified (or timedout) and tries to reenter the monitor, we should not allow it to execute (and reenter the monitor of `Late`) if the `Writer` is already at position `W0` or `W1`. Note how we have constructed the decision trees in a way which allows one to look at the system's controllability *from each task's point-of-view*. The tree of the `Writer` *imposes* the `Writer`'s current position as the most important attribute; see [5] for a lengthier discussion on this. Given that, the tree induction algorithm then finds that the position of the `Refresher` task is the most important variable, for controlling `Writer`. For the tree of the `Refresher` we have the opposite order, showing that `Writer`'s position is the most important information for `Refresher`'s safety. This close interdependence between `Writer` and `Refresher`

which is highlighted by the decision-tree form of the controller can be a good reason to deploy both these tasks on the same processor and the `User` in another one, if we decide to distribute the system's tasks.

Using this synthesised controller for avoiding deadlocks, by adding its constraints to the function which characterises ready tasks as safe, we can then analyse the system with respect to its deadlines. Examining first the case where the tasks cannot be preempted, we synthesise a controller which forbids 64 transitions in the system, see Figures 4 and 5. If we examine them we will see that the most constrained task now is `Writer`, since it looses precedence from both the `User` and the `Refresher` at many control points. Note also how the variable ordering differs in the sub-trees. The first part of `Writer`'s constraints (for position `W0`) ensures that the `User` will have precedence in the use of their mutually shared resource (`Val`) and the processor so as not to miss its deadline. The second part (for position `W2`) ensures that `Writer` will allow `Refresher` to advance enough so that it can capture `Writer`'s forthcoming signal (and subsequently signal `User` itself so that `User` can use the new value of the sensor). Looking at `Refresher`'s constraints we see that it also needs to examine `User`'s position so that it does not delay it, either by taking resource `Late` when the `User` also wants it (position `U6_Relock`) or by using the processor when the `User` needs it too (at positions `U1` and `U5` - note also the value of `User`'s clock there). The constraints imposed on `User` effectively cause it to release the processor when the `Writer` is about to notify `Refresher` (and there is enough time left until the next deadline). In summary, the synthesised scheduler imposes an order $User > Refresher$ and $User > Writer$ when the `User` is about to miss its deadline and an order $Refresher > Writer$ when the `User` is safe and the `Refresher` needs to receive `Writer`'s signal .

Strengthening the characterisation of safe tasks with the new constraints, we then examine the case where task preemption is allowed. Interestingly enough, the synthesised controller for the non-preemptive execution model manages to keep the system safe even in the case where tasks can be preempted. More interesting still, is the final controller we synthesise to test whether we can meet deadlines without observing the system's clocks, see Figure 3. In order to synthesise this controller we apply to the system the previous ones, making sure to remove any references they were making to the clocks, both periodic or stopwatches. So, if one of the previous controllers had a constraint of the form $(X \wedge Clock_i > 3)$ we need to transform it to (X). Looking at the resulting controller, we can see that it effectively gives priority to the `User` over the `Refresher` so that the former does not miss its deadline, and also gives priority to the `Refresher` over the `Writer` so that the `Refresher` can reach its `timed_wait()` primitive used to measure

Table 1. Performance Statistics for the system of Figure 1

Model	Space Construction	States (c/u)	Max u per c	Trans. (c/u)	Synthesis
deadlocks	.5 sec / 3.0 MB	551 / 3612	18	571 / 4173	.04 sec / 378 KB
non-preemption	10.6 sec / 53.6 MB	2747 / 27962	30	2837 / 28895	1.51 sec / 5.7 MB
preemption	4.5 sec / 19.6 MB	1435 / 13312	30	1470 / 13826	—
time independence	1.4 sec / 4.7 MB	481 / 3867	41	495 / 3990	.02 sec / 185 KB

Table 2. Performance Statistics for a larger system (2 periodic & 2 aperiodic tasks)

Model	Space Construction	States (c/u)	Max u per c	Trans. (c/u)	Synthesis
deadlocks	26.5 sec / 295.3 MB	6002 / 76098	28	6292 / 86665	—
non-preemption	92.5 sec / 341.9 MB	7699 / 130923	48	7760 / 142398	7.39 sec / 5.1 MB
preemption	65.7 sec / 241.7 MB	6197 / 95757	48	6239 / 104333	—
time independence	62.7 sec / 241.3 MB	5947 / 91304	48	5989 / 99554	—

the age of the latest value produced. This is not too difficult to explain - the User is the only task with a deadline, so it needs a higher priority. At the same time, the User cannot advance unless the Refresher notifies it, which shows why we need to give the latter priority over the Writer.

```
Writer-Unsafe-When-Not-Observing-Clocks :=
  (and (= Writer |W1|)
       (or (= Refresher |R0|)
           (= Refresher |R7|)))

Refresher-Unsafe-When-Not-Observing-Clocks :=
  (and (= Refresher |R1|)
       (= User |U5|)
       (= Writer |W2|))
```

Figure 3. Time-independent controller for deadlines

Table 1 gives certain performance data for this example. We can see that the number of controllable states (c) is around 15% of that of uncontrollable states (u) and that the ratio of controllable to uncontrollable transitions is similar. This shows not only the great savings in space we gain by our state reduction method but also the savings in time for the synthesis itself, since now the synthesis need not examine the uncontrollable states at all. Indeed, we can see that the maximum number of uncontrollable states in the window of a controllable state is rather high, going from 18 to 41. If we increase the duration of computations or the accuracy of our clocks (*e.g.*, counting milliseconds instead of seconds, *etc.*) then this number will increase even more and we will be able to perform a finer grain analysis at a small cost, since we only keep one such window each time. Table 2 shows the same set of data for a larger system, comprising of 2 periodic and 2 aperiodic tasks. This system has no deadlocks and the controller synthesised for the non-

preemptive execution model can also keep the system safe in a preemptive execution model, and does so even if we do not allow it to observe the system clocks. We can see that there the ratio of controllable to uncontrollable states is even better; the former are not more than 7% of the latter. This is also shown in the increase of the maximum number of uncontrollable states belonging to the window of a controllable state.

6. Conclusions & Future Work

JUPITER is based on previous work [7, 6] on scheduling approached as a controller-synthesis problem [11, 1]. It has allowed us to further explore three basic ideas: application of controller synthesis in practise, successive model refinement and, last but not least, use of Machine Learning techniques so as to structure and explain synthesised controllers [5]. We believe that the initial results we have obtained show the potential of our approach. They have also shown us other routes to explore, for example how we could use the decision tree form of the controllers to choose other system variables to observe or further understand and optimise a system (see [5] for further details on this).

JUPITER is still in the early development stage. We are currently examining various extensions, such as the ability to perform the synthesis in an *on-the-fly* manner during the state-space exploration [9]. This will allow us to not examine the whole state-space (unless it is safe), since we will be constraining the system while constructing the state-space itself. Another extension we want to introduce is *directed synthesis*. The idea behind it is to examine first those unexplored states which are more probable to lead to a bad state, thus allowing us to synthesise constraints earlier on in the exploration of the system. We are also examining such ameliorations as supporting functions (*i.e.*, introducing execution stacks in the state vector), dynamic creation

```
User-Misses-Deadlines :=
  (and (= User |U5|)
          (= Writer |W2|)
             (= Refresher |R2|)
                (or (and (= User_Periodic_Clock 14)
                            (= Refresher_Stopwatch 4))
                   (and (<= 17 User_Periodic_Clock) (<= User_Periodic_Clock 18)
                       (or (= Refresher_Stopwatch 2)
                           (= Refresher_Stopwatch 4)))))

Refresher-Misses-Deadlines :=
  (or (and (= Refresher |R0|)
            (or (and (= User |U1|)
                       (or (and (= Writer |W0|)
                                 (= User_Periodic_Clock 11))
                          (and (= Writer |W2|)
                                 (= User_Periodic_Clock 10))
                          (and (= Writer |W6|)
                                 (= User_Periodic_Clock 11))))
               (and (= User |U5|)
                       (or (and (= Writer |W0|)
                                 (= User_Periodic_Clock 13))
                          (and (= Writer |W2|)
                                 (= User_Periodic_Clock 13))
                          (and (= Writer |W6|)
                                 (= User_Periodic_Clock 13))))
               (and (= User |U6_Relock|)
                       (or (and (= Writer |W0|)
                                 (= User_Periodic_Clock 11))
                          (and (= Writer |W6|)
                                 (= User_Periodic_Clock 11))))))
      (and (= Refresher |R3|) (or (= User |U1|)
                                   (= User |U5|)
                                   (= User |U6_Relock|)))))
```

Figure 4. Controller for deadlines, non-preemptive execution, part I

of objects and system tasks, or user-definable controllable actions and controller-observable variables, which will ease the use of JUPITER in practise and allow it to be used in the analysis of more systems. Finally, we wish to eventually extend JUPITER with even more models and refinement steps, so as to be able to synthesise controllers for the memory usage of a system, its energy consumption, *etc.* This is a result of our wish to use JUPITER for modelling and analysing embedded systems, where engineers need to meet a multitude of different non-functional requirements.

References

[1] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1):55–84, July 2002.

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

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, Aug. 1986.

[4] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *Proc. of CONCUR 2000*, LNCS 1877, pages 138–152, 2000.

[5] C. Kloukinas. Data-mining synthesised schedulers for hard real-time systems. In *ASE-2004*, pages 14–23. IEEE Computer Society Press, Sept. 2004.

[6] C. Kloukinas, C. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time Java applications. In *EMSOFT'03*, LNCS 2855, pages 274–289, Oct. 2003.

[7] C. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *ECRTS'03*, pages 287–294. IEEE Computer Society Press, July 2003.

```
Writer-Misses-Deadlines :=
  (or (and (= Writer |W0|)
              (or (and (= User |U1|)
                          (or (and (= Refresher |R0|)
                                        (<= 8 User_Periodic_Clock) (<= User_Periodic_Clock 11))
                              (and (= Refresher |R2|)
                                        (<= 8 User_Periodic_Clock) (<= User_Periodic_Clock 11))
                              (and (= Refresher |R2_Relock|)
                                        (<= 7 User_Periodic_Clock) (<= User_Periodic_Clock 11))
                              (and (= Refresher |R7|)
                                        (<= 8 User_Periodic_Clock) (<= User_Periodic_Clock 11))))
                  (and (= User |U5|)
                          (or (and (= Refresher |R0|)
                                        (<= 11 User_Periodic_Clock) (<= User_Periodic_Clock 13))
                              (and (= Refresher |R2|)
                                        (<= 11 User_Periodic_Clock) (<= User_Periodic_Clock 13))
                              (and (= Refresher |R2_Relock|)
                                        (<= 11 User_Periodic_Clock) (<= User_Periodic_Clock 13))
                              (and (= Refresher |R7|)
                                        (<= 11 User_Periodic_Clock) (<= User_Periodic_Clock 13))))
                  (and (= User |U6|)
                          (= Refresher |R2_Relock|)
                            (<= 7 User_Periodic_Clock) (<= User_Periodic_Clock 10))
                  (and (= User |U6_Relock|)
                          (or (and (= Refresher |R0|)
                                        (<= 8 User_Periodic_Clock) (<= User_Periodic_Clock 11))
                              (and (= Refresher |R2|)
                                        (<= 9 User_Periodic_Clock) (<= User_Periodic_Clock 11))
                              (and (= Refresher |R2_Relock|) )
                              (and (= Refresher |R7|)
                                        (<= 8 User_Periodic_Clock) (<= User_Periodic_Clock 11))))))
      (and (= Writer |W2|)
              (or (and (= Refresher |R0|)
                          (or (and (= User |U1|)
                                        (<= 5 User_Periodic_Clock) (<= User_Periodic_Clock 8))
                              (and (= User |U5|)
                                        (= User_Periodic_Clock 13))
                              (and (= User |U6|)
                                        (<= 5 User_Periodic_Clock) (<= User_Periodic_Clock 8))))
                  (and (= Refresher |R2|)
                          (or (and (= User |U1|)
                                        (<= 5 User_Periodic_Clock) (<= User_Periodic_Clock 8))
                              (and (= User |U5|)
                                        (= User_Periodic_Clock 13))
                              (and (= User |U6|)
                                        (<= 5 User_Periodic_Clock) (<= User_Periodic_Clock 8))))
                  (and (= Refresher |R2_Relock|)
                          (= User |U5|)
                            (= User_Periodic_Clock 13)))))
```

Figure 5. Controller for deadlines, non-preemptive execution, part II

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1):46–61, Jan. 1973.

[9] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *FM'99*, volume 1708 of *LNCS*, Toulouse, France, Sept. 1999. Springer-Verlag.

[10] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *JACM*, 43(3), 1996.

[11] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, May 1987.