Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 83–110

Stressing Symbolic Scheduling Techniques within Aircraft Maintenance Optimization

Viviana Bruno viviana.bruno@polito.it
Luz Garcia luz.garcia@polito.it
Sergio Nocco sergio.nocco@polito.it
Stefano Quer stefano.quer@polito.it
Dipartimento di Automatica e Informatica,
Politecnico di Torino,
Torino, Italy

Abstract

Scheduling, or planning, is widely recognized as a very important step in several domains such as high level synthesis, real-time systems, and every-day applications. Given a problem described by a number of actions and their relationships, finding a schedule, or a plan, means to find a way to perform all the actions minimizing a specific cost function.

The goal of this paper is to develop, analyze and compare different scheduling techniques on a new scheduling/planning problem. The new application domain is aircraft maintenance. It shares with previous ones the underlying problem definition, but it also unveils brand new challenging characteristics, and a different optimization target. We show how to model the problem in a suitable way, and how to solve it with different methodologies going from Satisfiability solvers and Binary Decision Diagrams, to Timed Automata and Coloured Petri Nets. New ideas are put forward in the different domains having efficiency and scalability as main targets. Experimental results stress the different techniques, showing their application range and limits, and defining advantages and disadvantages of the underlying models. Overall, general-purpose tools have been easily applied to our problem, but failed as far as efficiency was concerned. The satisfiability-based approach proved to be faster and more scalable, being able to solve instances $3-4$ times larger.

To sum up, our contributions range from modeling the aircraft maintenance problem as a scheduling instance, to coding this problem with home-made and general-purpose tools, to dovetailing exact and heuristic techniques, and comparing these techniques in terms of efficiency and scalability.

Keywords: *Boolean satisfiability, SAT-solvers, timed automata, Petri nets, scheduling, planning*

Submitted February 2007; revised March 2008; published June 2008

1. Introduction

Scheduling is one of the key tasks in high-level synthesis, and it has gained broad recognition in real-time systems and in alternative application domains. Given a set of operations with data dependency relations, the scheduling problem associates a control step to each operation such that certain constraints are satisfied. Many of the practical scheduling problem instances are known to be NP–Complete [18].

There exist a plethora of heuristic scheduling techniques in the literature, e.g., [29, 13], aiming to provide solutions for practical problem instances. Although many of these heuristics can deal with large problems, they usually fail to find high quality solutions, especially in tightly constrained formulations, where early pruning decisions may exclude candidates eventually leading to superior results. For example, Dain et al. [13] presented ARGOS, i.e., a heuristic search-based optimization tool to find ship construction scheduling. The tool works by heuristically constructing a schedule, and then looking for changes in the schedule able to reduce the scheduling cost. Solutions are sub-optimal, but the tool is able to deal with huge problems obtained from commercial shipyards, which, for a single hull, may consist in several thousands activities spanning some working years. Moreover, in these cases a scheduling must be evaluated also for its ability to overcome environmental problems such as task delays, labor shortages, etc. For that reason, heuristic tasks are sometimes combined with simulation activities, see for example [14], where a stochastic shipyard simulator is designed to evaluate the performance of a scheduling given environmental problems.

Integer Linear Programming (ILP) methods (e.g., [24]) can solve scheduling exactly. However, the ILP complexity significantly increases by considering control constraints, and thus it can lead to unacceptable execution times.

More recently, symbolic methods [22, 23, 27, 11] have been proved effective in finding exact solutions in highly constrained problem formulations. In these cases scheduling constraints are represented as Boolean functions, and all solutions are implicitly enumerated by means of Binary Decision Diagrams (BDDs). Post-process pruning is used to apply additional constraints which may not have an efficient formulation within heuristic approaches and ILP. Moreover, symbolic methods yield a very efficient formulation of control dependencies and environmental timing constraints.

The goal of this paper is to analyze a particular scheduling problem within the field of aircraft maintenance. Maintenance is a very important phase of an aircraft life-cycle as, to grant safety and reliability, it has become more and more detailed and refined. An aircraft maintenance consists of a large number of tasks, often recursively expressed in terms of sub-tasks, that have to be executed respecting specific temporal constraints. These constraints are usually expressed as precedences and mutual-exclusions between main tasks or even sub-tasks. Each operation is executed by personnel with specific skills (and then different hourly salaries) and with suitable tools and materials (with different costs). The target of the problem is not simply to minimize the total time necessary to complete the task, but to optimize the maintenance cost, i.e., the amount of money necessary to maintain the aircraft. This, in turn, is usually computed as a function of the time spent by each group of employees to do their job on the plane and their hourly salaries.

The above problem is somehow considerably different, and more complex, than other scheduling puzzles available in the literature:

- Tasks are usually defined in terms of other tasks or sub-tasks. On the contrary, in other fields, i.e., in hardware scheduling, each task, e.g., computing the sum of two values, is considered as an atomic operation. This feature implies a pre-processing of the problem, in order to decompose each task into atomic operations, and to define their relationships in terms of time, mutual exclusion, etc. Similarly, a final post-processing manipulation is necessary to obtain the task scheduling starting from the plan of atomic operations.

- Operations may require a wide range of time to be completed, i.e., from minutes to hours, whereas other scheduling models often consider tasks lasting just one or a few clock cycles. This characteristic makes the problem harder, as keeping track of large elapsed time is a major hurdle.
- Tasks often require more than one resource at the same time, whereas in hardware scheduling one resource is usually sufficient to complete a certain operation. This feature increases the number of constraints that have to be considered to solve the problem.
- In certain occasions, tasks that have already been done do not need to be repeated, unless the complementary operation has been executed in the meantime. This implies that certain operations can be factorized, i.e., performed just once, among different similar activities, whereas they have to be repeated in other occasions. For example, we can empty-out the hydraulic system and perform several fixing operations on tubes and connectors of the hydraulic system itself. This means that the operation "empty-out the hydraulic system" has been factorized among all fixing operations. On the contrary, if we want to fully check the hydraulic system, we have to store its pressure back, and in case of a failure we have to empty-it-out again. This implies that the operation "empty-out the hydraulic system" cannot be factorized among different complete checks of the system.
- Employees are paid even if they stand in the airfield doing nothing, whereas hardware functional units may be idle without increasing the cost of the problem. This would require a different optimization target similar to the one used to reduce the power consumption on multi-processor systems [6].

These considerations motivate further investigation to evaluate the efficiency and scalability of formal techniques on maintenance activities.

Our strategy can be divided into two main phases. In the first phase, we translate the airplane maintenance model into a Data Flow Graph (DFG) including all required information on the original problem. A DFG is a directed graph describing the set of tasks to be performed and their temporal relationships. This transformation takes into account all constraints and costs. In the second phase, we solve the problem by adopting different techniques. First, we describe the problem using a symbolic approach based on breadth-first reachability and Binary Decision Diagrams. Second, we solve the problem adopting Bounded Model Checking (BMC) and a satisfiability (SAT) solver as underlying engine. Novel solutions are used to deal with the new features of the problem. We aim at optimizing long operations and analyze different coding methodologies for the SAT tool. Given the specific issues of our application, we also modify the standard BMC technique in order to increase its capacity. Third, we use Timed Automata as the way to appropriately describe functional units with a wide range of execution times. This model is implemented within the UPPAAL and UPPAAL-CORA [5, 4] tools. Finally, we model the problem using Petri Nets. More specifically, to appropriately describe times, precedences, and mutual-exclusions, we adopt Colored Petri Nets, and the CPN-tools [25]. We use different running examples, throughout the text, to clarify our models and methods.

Overall, our contributions are the following:

- Modeling the aircraft maintenance problem as a scheduling instance, appropriately coding the avionic data base describing the process.
- Expressing the scheduling instance with BDD- and SAT-based techniques, and with general-purpose tools based on Timed Automata and Coloured Petri Net.
- Contrasting SAT-based BMC and heuristic techniques, to trade-off capacity for accuracy.
- Comparing, on the BMC problem, different coding techniques and several state-of-the-art SAT solvers.

Our experimental results concentrate on comparing advantages and disadvantages of the different models and strategies adopted, finding strengths and weaknesses for each of them. Real problems, coming from the avionic industry, are used to stress the methods.

The paper is organized as follows. Section 2 introduces some preliminary notions on scheduling. Section 3 describes the avionic maintenance problem we deal with. Section 4 introduces the first phase of our algorithm, in which we convert our aircraft model into a Data Flow Graph. Section 5 describes the second step, the one in which we model the problem adopting different techniques. Finally, Section 6 discusses the experiments we performed, and Section 7 concludes with a few summarizing remarks.

2. Background

We assume that the reader is familiar with BDDs, and SAT. Timed Automata and Coloured Petri Nets are defined in Section 5.2 and 5.3, respectively. As a consequence we briefly review only the basic concepts that are relevant for our application framework.

In our notation, B indicates the Boolean space. Symbols $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ are used for Boolean conjunction (AND), disjunction (OR), negation (NOT), implication, and co-implication, respectively.

The automata we address are usually represented implicitly by Boolean formulas. For our purposes, an *automaton* is a triple $\mathrm{A} = (\mathsf{I}, \mathsf{TR}, \mathsf{T})$, where I is the set of initial states, TR is the transition relation between the states, and T is the target set of states.

The (present) state space of the automaton is defined by an indexed set of m Boolean variables $P = \{p_1, \ldots, p_m\}$. We also indicate next state variables with $N = \{n_1, \ldots, n_m\}$, disjoint from P.

TR is the transition relation containing all couples (present state P, next state N) such that there is at least one input value that lets the system go from state P to state N.

A scheduling, or planning, problem may be understood in terms of a certain number of objects, each one associated with various distinguishing attributes. Standard attributes are object resources, operand dependencies, and control decisions. The possible schedules, or plans, solving the problem are described by a number of actions, the execution of which may depend on and affect the values of (some of) the objects attributes. We are interested in finite scheduling, i.e., in a finite sequence of actions that takes the system from an initial to a final configuration.

The following example describes a *traditional* hardware scheduling problem involving only data operations (and no decisions) which last one single time unit. Our real problem, and the main differences with this one, will be introduced in Section 3.

Example 1. *Let us suppose to have* 5 *tasks, namely A, B, C, D, and E, such that A and B have to be executed before D, and C and D have to be executed before E. Moreover, let us suppose that all tasks can be executed in a single time unit and that they require different operators/resources, i.e., tasks A, B and C are performed by the operator $op1$, D by $op2$ and E by $op3$. Given this sequence of tasks, we characterize the problem by the Data Flow Graph (DFG) of Figure 1.*

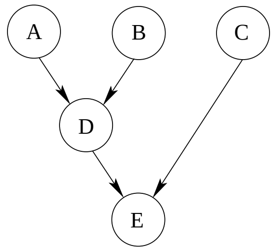


Figure 1. A Data Flow Graph (DFG) example.

As mentioned in the introduction, a DFG is a directed graph describing the set of tasks to be performed (represented as the nodes of the graph), and their temporal relationships (indicated by the edges of the graph). Given the DFG, several scheduling solutions can be found depending on:

- *The number of operators/resources allocated for each type of operation, namely $op1$, $op2$ and $op3$.*
- *The type of resources that are possible to adopt for each operation, e.g., we can suppose to use a particular tool (or employee), namely op, able to complete more than one kind of operation.*

Figure 2 shows some of the possible scheduling instances, with different set of resources ($op1$, $op2$, $op3$) available:

- *In Figure 2(a) there is no resource limit. All operations needing $op1$ can run in the first time step as there are enough resources to execute them in parallel. The total time necessary to complete all operations, usually called latency, is equal to* 3 *time units.*
- *In Figure 2(b) two resources are available for operator $op1$. As a consequence the third task needing the operator $op1$ has to be delayed until the second time unit. The latency is again equal to* 3 *time units.*
- *In Figure 2(c) one single resource is available for each operation. Then, just a single $op1$ operator can be used in each single time unit. The latency increases to* 4 *time units.*

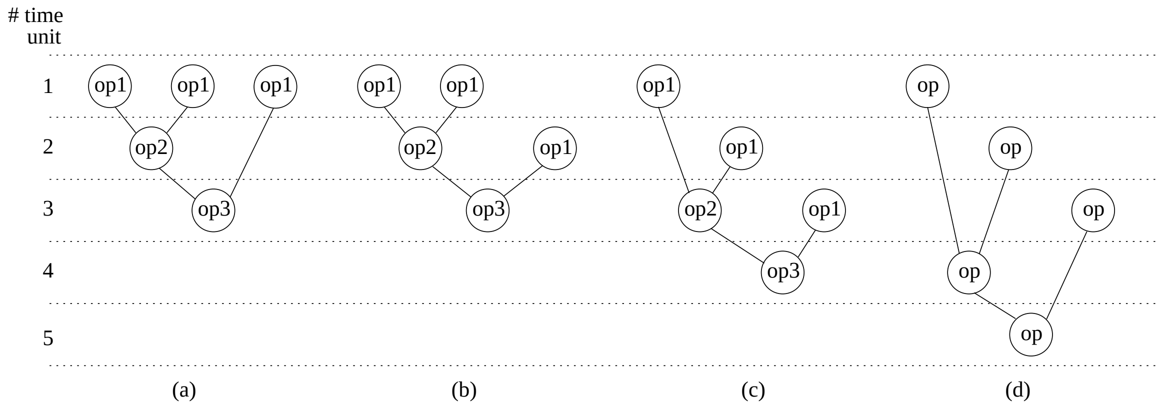


Figure 2. Scheduling solutions for the DFG of Figure 1.

- *In Figure 2(d) only one resource op is available, but that resource is able to complete all possible tasks. In this case, the latency is equal to* 5 *time units.*

Notice that in our analysis we will have to increment the above model in different directions. Not only each operation will require a certain number of resources (not just a single one), but we also will have mutual exclusions among operations. Moreover, operations will require different times (and costs) to be completed, thus complicating the planning solution.

3. Problem Description

Our scheduling problem is coded in a data base coming from the avionic industry and containing the following information. Each maintenance operation takes the name of Data Module (DM). Each DM may be recursively defined in terms of other DMs. An atomic DM is a DM that represents an atomic operation. Such a DM is also called Reference (R). A DM is completely executed when all atomic references defining it are executed in the precise order in which they are defined within the DM.

Each elementary operation, i.e., a reference, requires a specific time to be completed, must be performed by definite personnel, and needs particular tools. Operation times have a wide range of values (from minutes to several hours), and this characteristic greatly differentiates the problem from standard hardware scheduling, where operations usually require a single or few time units. A limited number of resources can be used by each DM, such as materials, tools, work benches, skilled staff, etc.

Before a given DM can be executed, its prerequisites, that are the conditions necessary to start the task, need to be satisfied. These prerequisites are called Required Conditions (RC) and usually are other DMs.

To complicate the overall task, two DMs may be incompatible, i.e., they cannot be executed at the same time and have to be executed in mutual exclusion.

Example 2. *The following example describes a simple scheduling problem. For the sake of readability, we consider only data operations, precedence relations, and mutual exclusions without representing any resource limitation.*

Figure 3 reports a simple initial configuration.

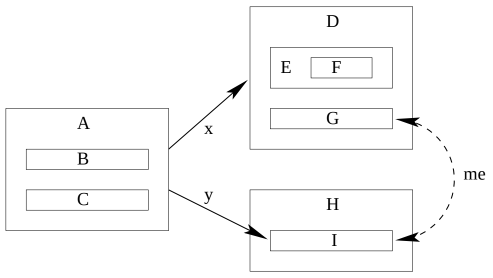


Figure 3. The input model: DM structure and dependency.

In this schema A, D, E and H are non-atomic DMs, whereas B, C, F, G and I are references. DM A has to be completed before D and I (see the precedence edges named `x` *and* `y`*). The references G and I must be executed in mutual exclusion (see the incompatibility edge named* `me`*). Moreover, DM A is made up of references B and C, DM D is composed of data module E (which in turn contains reference F) and reference G, and H contains I.*

4. Creating a DFG Model

The first step of our algorithm is to create a DFG starting from the original data base. The core idea is to recursively decompose each data module into references, i.e., atomic operations, and, at the same time, appropriately keep into consideration precedences, resource constraints, and mutual exclusions. The pseudo-code of this step is presented in Figure 4.

```
createDFG(DB)
    ADB ← $\phi$
    for each m ∈ DB
        ADB ← ADB ∪ createADB(m)
    for each a ∈ ADB
        expandExplicitRC(a)
    for each a ∈ ADB
        expandImplicitRC(a)
    return (ADB)
```

Figure 4. Modeling recursively defined tasks.

DB is the original data base. ADB is the atomic data base (the DFG with only atomic operations involved) we want to create. Initially, ADB is empty. Then, for each DM m in the original data base DB, function createADB recursively expands m in the sequence of its references. Renaming is performed to maintain the unicity of references at the level of the atomic data base ADB. After that, we have to take care of all required conditions. This in done in two separate steps:

- Function expandExplicitRC expands each required condition explicitly contained in the original non-expanded data base. It generates a new set of required conditions for each single required condition (i.e., a precedence edge) present in the original data base DB.

- Function expandImplicitRC generates all the required conditions due to the expansion of the original data base into atomic operations.

Notice that these two functions also take into consideration all resource constraints and mutual exclusions among data modules and references.

Example 3. *Figure 5 shows the DFG corresponding to the original aircraft maintenance problem represented in Figure 3.*

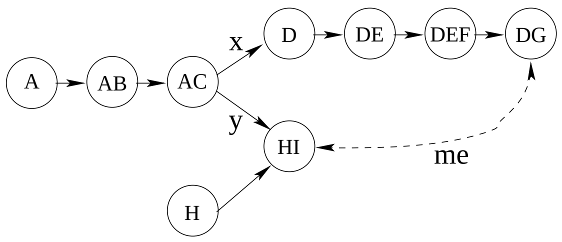


Figure 5. Modeling recursively defined tasks: A first step.

The sequence of operations A, AB, and AC, for instance, derives from the single DM A of Figure 3. Notice that it is necessary to specify A in the DFG, beyond AB and AC, as every DM in the original data base always includes some atomic operations even when it is defined in terms of other DMs and/or references.

Function expandExplicitRC *recreates precedences named* `x` *and* `y` *and the incompatibility edge* `me`*. Function* expandImplicitRC *generates all the other precedences (implicitly present in the original data base).*

Notice that at this stage we do not factorize any common operation, because the industrial source data base does not code the necessary information, i.e., when a DM may or may not be executed just once. In other words (see Figure 6) if two DMs A and B have C as common reference (Figure 6(a)) we generate the ADB represented in Figure 6(b).

Nevertheless, it would be possible, at least in some occasions, to factorize C and generate the ADB represented in Figure 6(c). We do not perform such an optimization because at the moment the original data base does not report enough information about factorisable operations and their counterpart (e.g., turning-off an electronic device can be factorized, i.e., it can be done only once, at least until it is turned-back-on).

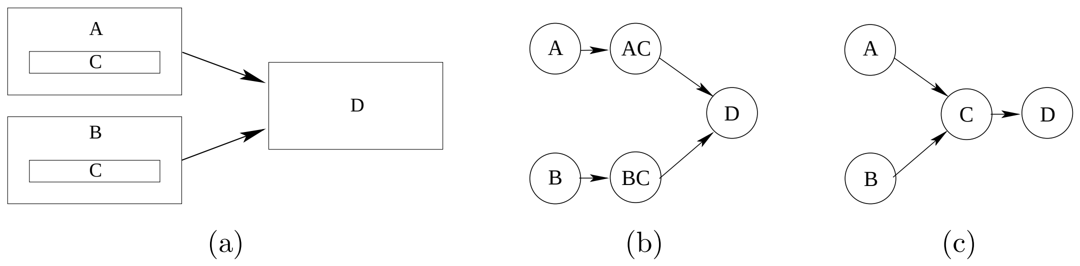


Figure 6. Factorize common operations.

5. Scheduling Methodologies

Scheduling of tasks in real-time systems has traditionally been conducted using purely algorithmic approaches [9]. Heuristic approaches have been adopted for their simplicity and efficacy on lightly constrained problems [33, 34, 10].

In this section, we analyze the application of several symbolic techniques to this domain. Model Checking, based on both Binary Decision Diagrams and SAT solvers, is definitely interesting and has been increasingly adopted for scheduling problems. Timed Automata [2] (or even Priced Timed Automata [6]) are another possible modeling approach. Finally, Petri Nets have been selected for their capability to model and analyze real-time systems.

5.1 BDD and SAT Based Scheduling

In [22, 23, 30] the authors proposed a BDD-based symbolic technique, able to produce the optimal latency for resource-constrained scheduling. The key idea was to model the input DFG as a non-deterministic automaton, so that symbolic model checking methodologies could be applied.

SAT-based model checking was independently formulated in [27] and [11]. In the first work, given a target latency, a complete 1-hot encoding [28] was used. In this encoding a Boolean variable x_{ijk} was introduced to indicate (when $x_{ijk} = 1$) that operation i is scheduled in time frame j on resource k. A set of constraints was imposed over these variables in order to obtain valid scheduling traces. In [11], the authors adapted to SAT the automaton encoding introduced for BDDs in [22]. In the sequel, we will show how the automaton is constructed following this approach.

5.1.1 Expressing atomic operations of the DFG

Each DFG operation is modeled through a non-deterministic automaton. For every single-time-step operation of the given DFG, the automaton provides information about the execution of the associated operation. If we indicate with 0 the state in which an operation has not yet been scheduled, and with 1 the state in which the operation has been scheduled, the automaton may be represented as in Figure 7.

More formally, as the automaton has only two states, its transition relation may be encoded with exactly two Boolean variables, i.e., $P = \{p_1\}$ for the present state and $N = \{n_1\}$ for the next state. The meaning of the possible automaton transitions is hence the following:

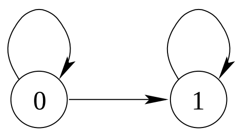


Figure 7. Basic scheduling automaton.

- $p_1 = 0$ and $n_1 = 0$: The operation has not been scheduled in previous steps and will not be scheduled in the next one.
- $p_1 = 0$ and $n_1 = 1$: The operation has not been scheduled in previous steps but it is going to be scheduled in the next one.
- $p_1 = 1$ and $n_1 = 1$: The operation has been previously scheduled.

The automaton transition relation is encoded as:

$$\begin{aligned}\mathsf{TR}_{op}(P,N) \quad = \quad & (\ (p_1 = 0) \Rightarrow ((n_1 = 0 \vee n_1 = 1)\ )\ \wedge \\ & (\ (p_1 = 1) \Rightarrow (n_1 = 1)\ )\end{aligned}$$

which may be simplified to:

$$\mathsf{TR}_{op}(P,N) \quad = \quad (\ (p_1 = 1) \Rightarrow (n_1 = 1)\ )$$

When it is necessary to work with operations requiring more than one time step to be performed (this is one of the specific issues of our application), the previous representation must be extended to the automaton shown in Figure 8, where l is the given operation latency.

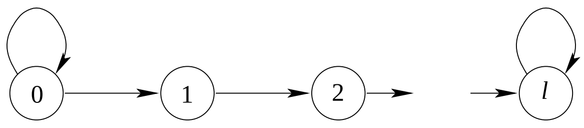


Figure 8. Basic scheduling automaton for latency l.

Notice that the automaton may start its execution non-deterministically, but then it must proceed to the next state at every time step. The physical meaning for this behavior is that an operation, once started, cannot be stopped and resumed sometimes later. The overall behavior of TR_{op} is the conjunction of all transitions between each couple of states. Although it is not formally correct (as P and N are sets of variables), we will model the fact that the automaton is in a present (next) state numbered i as $P = i$ ($N = i$). For instance, the behavior of the automaton starting from state 0 (see Figure 8) is represented as $(P = 0) \Rightarrow (N = 0 \vee N = 1)$, meaning that from the state 0 it is possible to progress to state 0 or state 1.

The complete expression of TR_{op} is hence the following one:

$$\begin{aligned}\mathsf{TR}_{op}(P,N) \quad = \quad & ((P=0) \Rightarrow (N=0 \vee N=1)) \wedge \\ & ((P=1) \Rightarrow (N=2)) \wedge \\ & ((P=2) \Rightarrow (N=3)) \wedge \\ & \dots \\ & ((P=l-1) \Rightarrow (N=l)) \wedge \\ & ((P=l) \Rightarrow (N=l))\end{aligned} \tag{1}$$

A scheduling activity occurs whenever the value of the next state is different from the current one. This condition can be also expressed as $(P \neq l) \wedge (N \neq 0)$. For a single time step operation, $l = 1$, there are only two states (0 and 1), and one present and one next state variables are sufficient. Then the previous relation is reduced to $(p_1 \neq 1) \wedge (n_1 \neq 0)$, that is just $(p_1 = 0) \wedge (n_1 = 1)$.

The Boolean meaning of the notations $P = 0$, $N = 0$, etc., as well as the number of Boolean variables necessary to express these equations, depends on the strategy adopted to encode the problem. In this paper, we analyze two different encodings, showing the advantages and disadvantages for both of them.

Logarithmic Encoding The *logarithmic encoding* emulates the encoding used for binary counters. Given a positive value l, a number $m = \lceil log_2(l) \rceil$ of Boolean variables is used, and all the values in the range $[0, l]$ are represented as binary numbers over them. As a consequence, the term $((P = 0) \Rightarrow (N = 0 \vee N = 1))$, in Equation 1 becomes:

$$\begin{aligned}((p_m=0) \wedge \ldots \wedge (p_2=0) \wedge (p_1=0)) \quad \Rightarrow \quad & ((n_m=0) \wedge \ldots \wedge (n_2=0) \wedge (n_1=0)) \vee \\ & ((n_m=0) \wedge \ldots \wedge (n_2=0) \wedge (n_1=1))\end{aligned}$$

All the other terms of TR_{op} can be expressed following the same principles.

This type of encoding requires the minimum possible number of state variables, and hence it is very beneficial when the scheduling problem is fully solved adopting BDDs [21]. However, when a SAT-solver is used, this type of encoding has several drawbacks. First of all, unit propagation under this encoding is not effective [32]. Moreover, each of the implications appearing in TR_{op}, as defined by Equation 1, requires the generation of m clauses, each of which is made up of $1 + m$ literals. The total number of clauses we need to express in TR_{op} is hence, approximatively, $l \cdot m$. Even more importantly, the logarithmic encoding severely impacts the Boolean expression of the other constraints imposed among different operations (see Section 5.1.2), making them much more complex to be translated into the CNF representation. This is because, in principle, any automaton state is identified by a specific Boolean value over *all* the state variables.

Thermometric Encoding The drawbacks of the logarithmic encoding are overcome with the second encoding strategy which adopts more state variables. This new encoding is often called *thermometric* [20][1]. In this strategy, both the P and N sets are represented with a number of Boolean variables equal to the operation latency l. Then, for every state i,

1. The term *thermometric* comes from old mercury thermometers, where one side is always filled with mercury and the other one is empty. Notice that the thermometric encoding is called “Full Regular” in [3], where it is used to translate a CSP problem into a SAT instance.

exactly the initial i consecutive state variables take the 1 value, whereas all the others are set to 0.

For instance, in Equation 1, writing $P = 2$ (respectively $P = 3$) actually means $\{p_l, \dots, p_1\} = \{0, \dots, 0, 0, 1, 1\}$ (respectively $\{p_l, \dots, p_1\} = \{0, \dots, 0, 1, 1, 1\}$), up to the final state l for which the encoding is a sequence of all ones $\{1, \dots, 1, 1, 1, 1\}$. The behavior is the same as that of a shift register with l bits, all initialized with 0, and accepting a value equal to 0 for a certain number of time steps and then a value equal to 1.

The advantages of the thermometric encoding are two fold. First, given the previous considerations, TR can be represented in a very concise way:

$$\begin{aligned}\mathsf{TR}_{op}(P, N) \quad = \quad & ((p_1 = 1) \Rightarrow (n_1 = 1)) \wedge \\ & ((n_2 = 1) \Leftrightarrow (p_1 = 1)) \wedge \\ & ((n_3 = 1) \Leftrightarrow (p_2 = 1)) \wedge \\ & \dots \\ & ((n_l = 1) \Leftrightarrow (p_{l-1} = 1))\end{aligned}$$

The implication described on the first line indicates that, once the first bit is set to 1, it remains at 1 forever. This corresponds to the fact that from state 0 it is possible to stay in the same state or to reach state 1, as shown in Equation 1. The following co-implications represent the shift-register-like behavior previously described: The value of the bit in position i in the next clock cycle is given by the value of the bit in position $i - 1$ in the current clock cycle. As a consequence, expressing TR_{op} into CNF format requires a number of clauses which is only linear with l. Furthermore, each clause is made up of exactly two literals.

The second advantage of the thermometric encoding derives from the fact that, for the given automaton, the invariant $(p_i = 1) \Rightarrow (p_j = 1)$ is true for all $j < i$, i.e., if the bit in position i is equal to 1, then all bits in previous positions are also equal to 1. Dually, the invariant $(p_i = 0) \Rightarrow (p_j = 0)$ is true for all $j > i$. This means that the automaton initial and final states can be identified through the condition $p_1 = 0$ and $p_l = 1$, respectively. The consequence is that the expression of the constraints discussed in Section 5.1.2 can be kept compact even when translated into clauses, with the length of each clause being *independent* from the operation latencies. We will show its effectiveness with respect to the logarithmic encoding in the experimental results section. Moreover, let us remark here that, compared to the 1-hot encoding, it allows to express implications between different bit configurations, i.e., states, in a more compact way.

5.1.2 Expressing the entire system

Once the single automata are represented, the complete DFG scheduling automaton is the Cartesian product of all automata, restricted by several constraints, each of which represents a particular allowed behavior. The overall formulation is the following:

$$\mathsf{TR}(P, N) \quad = \quad [\ \textstyle\bigwedge_i \mathsf{TR}_{op_i}(P, N)\ ] \wedge \mathsf{TR}_{dd}(P, N) \wedge \mathsf{TR}_{me}(P, N) \wedge \mathsf{TR}_{rc}(P, N) \qquad (2)$$

where:

- TR is the Transition Relation of the entire system, involving the original basic automata, refined by constraints for data dependencies, resource limits, and mutual exclusions.

- TR_{op_i} describes the single automata behavior, following Equation 1.

- TR_{dd} represents data dependencies (dd) or required conditions. It is illegal to schedule an operation j ($N_j \neq 0$) with a predecessor i that has not yet been completed ($P_i \neq l_i$). In other words, for any data dependency $i \rightarrow j$ between operations i and j, the expression $(P_i \neq l_i) \wedge (N_j \neq 0)$ represents an illegal condition. TR_{dd} can be expressed as the conjunction of all legal conditions:

$$\mathsf{TR}_{dd}(P, N) \quad = \quad \bigwedge\nolimits_{(i \rightarrow j) \in \mathsf{dd}} \; \big((P_i = l_i) \vee (N_j = 0)\big)$$

  The number of clauses generated by TR_{dd} is linear with the number E of edges representing dependencies in the given DFG. When using the thermometric encoding, such a number is exactly E, and each clause contains exactly 2 literals.

- TR_{me} represents the mutual exclusions (me). Given two operations i and j in mutual exclusion, it is illegal to have any schedule activity involving them simultaneously. In other words, when operations i and j are in mutual exclusion it is not possible to have operation i active ($(P_i \neq l_i) \wedge (N_i \neq 0)$, as described in Section 5.1.1) when operation j is active ($(P_j \neq l_j) \wedge (N_j \neq 0)$). TR_{me} can be expressed as the conjunction of all legal conditions:

$$\begin{aligned}\mathsf{TR}_{me}(P, N) \quad &= \quad \bigwedge\nolimits_{(i,j) \in \mathsf{me}} \neg\big[\big((P_i \neq l_i) \wedge (N_i \neq 0)\big) \wedge \big((P_j \neq l_j) \wedge (N_j \neq 0)\big)\big] \\ &= \quad \bigwedge\nolimits_{(i,j) \in \mathsf{me}} \;\; \big[\big((P_i = l_i) \vee (N_i = 0)\big) \vee \big((P_j = l_j) \vee (N_j = 0)\big)\big]\end{aligned}$$

  The number of clauses necessary to translate TR_{me} in CNF depends on the number M of mutual exclusions specified in the problem. When using the thermometric encoding, each mutual exclusion can be expressed as a single clause with exactly 4 literals.

- TR_{rc} represents resource constraints (rc). Let b_R be the number of instances available of a given resource class $R \in \mathsf{rc}$, and ρ_R the set of operations competing for such a resource set. It is illegal to schedule more than b_R concurrent operations from ρ_R. In other words, for any subset $\alpha_R \subseteq \rho_R$ such that $|\alpha_R| > b_R$, it is illegal to have all operations in α_R active at the same time. As in the previous cases, TR_{rc} is expressed as the conjunction of all the legal terms:

$$\mathsf{TR}_{rc}(P, N) \quad = \quad \bigwedge\nolimits_{R \in \mathsf{rc}} \neg\big[\bigwedge\nolimits_{\alpha_R \subseteq \rho_R, \, |\alpha_R| > b_R} \bigwedge\nolimits_{i \in \alpha_R} \big((P_i \neq l_i) \wedge (N_i \neq 0)\big)\big]$$

  The number of clauses generated by TR_{rc} depends on how such a constraint is expressed. In [21], it was proved that each illegal term making TR_{rc}, when represented as a BDD instead of a two level form, has a size (in terms of BDD nodes) proportional to $b_R \cdot |\rho_R|$, i.e., to the number of resource units available for the class times the number of operations competing for that resource class. By adopting the translation methodology from BDDs to CNF proposed in [12], the CNF representation can be easily obtained with similar complexity (in terms of generated clauses and added auxiliary variables).

Notice that, in traditional hardware scheduling, each operation is definitely mapped onto a single resource class, i.e., executed by a functional unit of that class. In our application, instead, any DM may require *several* resource instances belonging to different classes. The given formulation, however, naturally covers this case. Given a DM requiring one resource unit of R different classes, any scheduling activity on that DM will be accounted for by the expression of TR_{rc} for each of the involved R resource classes. In this case, however, some specific optimizations, aiming at reducing the BDD size of TR_{rc}, are possible[2]. These optimizations have been performed through the use of the BDD *restrict* operator.

The modeling automaton described by TR encapsulates all legal execution sequences of the system. Once TR is computed, the description of the complete automaton is fully specified with the definition of its initial and final states:

- The initial state I is the state in which no operation has been scheduled.
- The final, or target, state T is the one in which all the operations have been scheduled.

In practice, I and T are the Cartesian products of the basic automata initial and final states.

Given this information, we want to find the shortest possible path connecting I and T. This can be done with both a BDD-based and a SAT-based approach. We analyze these two methodologies in the following two subsections.

5.1.3 BDD-based Model Checking Formulation

To solve our problem, the first possibility is to perform BDD-based symbolic breadth-first reachability analysis starting from I and ending as soon as T is reached.

The set of states reachable at the i-th clock cycle may be computed by a standard iterative image computation:

$$\begin{aligned} S_i(P) &= \textsc{Img}\ (\mathsf{TR}, S_{i-1}) \\ &= \exists_P \big[\mathsf{TR}\ (P,N) \wedge\ S_{i-1}(P)\big] \end{aligned}$$

starting with $S_0 = \mathsf{I}$. Valid schedules are represented by state paths that reach the final set of states T, in which all terminal operations have been scheduled. If the target is minimum execution latency, the search may stop as soon as T is reached.

As we will show, adopting BDDs gives rise to the state explosion problem, even when the logarithmic encoding is used.

5.1.4 SAT-based Bounded Model Checking Formulation

The second possibility is to solve the related Bounded Model Checking (BMC) problem with the target of finding the smallest possible bound (representing the optimal schedule latency) for which the generated propositional formula is satisfiable.

Standard BMC would imply the following steps. First of all, to describe a sequence of transitions from s_0 to s_l (through $s_1, s_2, \ldots, s_{l-1}$), the transition relation TR (see Equation 2) has to be unrolled l times:

$$path(s_0, \ldots, s_l) \quad = \quad \mathsf{TR}(s_0, s_1) \wedge \ldots \wedge \mathsf{TR}(s_{l-1}, s_l)$$

2. When two DMs compete for a common subset of the total resource classes, the scheduling activity for both of them will be accounted for within *all* the shared resource classes. It is possible to demonstrate that such a repetition can be avoided, by considering a simultaneous transition of the related automata only once.

After that, the so-called *exact* l problem, which looks for paths of length exactly equal to l, has to be generated:

$$(s_0 = \mathsf{I}) \wedge path(s_0, \ldots, s_l) \wedge (s_l = \mathsf{T})$$

Finally, the problem is expressed in CNF form and solved with a state-of-the-art SAT-solver to prove (or disprove) the reachability between the initial and final states in l steps. Notice that in standard BMC l usually starts from 1, and it is increased until the problem is solved or computation resources are exceeded.

This approach showed scalability problems on our benchmarks. To make it more scalable, the following considerations are possible. First of all, our scheduling problems always have a solution. As a consequence, our BMC problems sooner or later will deliver a SAT result. An upper bound on the total scheduling latency can be quickly found by using any heuristic technique (ASAP, ALAP, force-directed, path-based, etc.). Thus, a second strategy is to start from the highest bound, for which a scheduling solution has been already found, and to decrease it in order to find the first unsatisfiable instance. A third strategy, as proposed in [11], is to perform a binary search of the optimal latency. Although the number of analyzed CNF problems is minimum, the drawback of the binary search is that some of the generated problems are hard-to-solve unsatisfiable problems. For this reason Cabodi et al. [11] exploited a SAT run with abort, i.e., they gave the SAT solver was given a (small) time limit to perform the search. This limitation, however, could lead to sub-optimal results, since each time overflow was interpreted like an unsatisfiable instance, thus possibly missing some shorter solutions. In this paper, we follow the second strategy, but we incorporate in the SAT instances also some further information obtained with the heuristic solution. The heuristic scheduling is used not only for the initial estimate of the total latency, but also to predict a complete scheduling with shorter latency. The prediction is made by adapting the scheduling time in the previous schedule solution, *without taking into account any of the scheduling constraints*. We then use the SAT solver to validate the predicted scheduling. The overall approach is represented in Figure 9.

The procedure receives the original data base DB and the system resource limits, i.e., time and memory, that the verification process has to satisfy. Initially (line 3) , function createDFG creates the atomic data base ADB (see Section 4). This data base is used by function createAutomata, which builds the automaton model of the DFG, returning its transition relation TR, as well as its initial and final state sets, following Sections 5.1.1 and 5.1.2. Function heuristicScheduling provides the initial heuristic scheduling S and the upper bound of the global latency l. Then, a loop (lines $9 - 23$) is entered. At each iteration, a CNF problem is built, expressing the mutual reachability between I and T along a path made up of l steps, which is then checked through a SAT solver. More precisely, the CNF expression we construct is generated by simplifying the exact problem with the information coming from the predicted scheduling (line 11). In practice, we allow the SAT solver to trigger an operation only in a window of time steps, centered in the operation predicted scheduling time, with width $2 \cdot \Delta$. This is done by function simplify. For each operation i, this function computes the predicted scheduling time t_i starting from the heuristic scheduling, and then it forces the operation automaton to be in the initial state for all time steps $t < t_i - \Delta$, and in the final state for all time steps $t > t_i + l_i + \Delta$. This is done in the following way. First, we add to the problem a set of unit clauses, expressing

```
BMCScheduling(DB, limits)
    // Initial setting
    ADB = createDFG(DB)
    (TR, I, T) = createAutomata(ADB)
    (S, $l$) ← HeuristicScheduling(ADB)
    $l \leftarrow l - 1$
    $\Delta \leftarrow 1$
    // Main loop
    while (TRUE)
        unroll ← $(s_0 = \mathsf{I}) \wedge path(s_0, \ldots, s_l) \wedge (s_l = \mathsf{T})$
        cnf ← simplify(unroll, S, $\Delta$)
        result ← sat(cnf, limits)
        if (result = OVERFLOW)
            return (S, $l + 1$)
        if (result = SAT)
            S ← trace(cnf)
            $l \leftarrow l - 1$
            $\Delta \leftarrow 1$
        if (result = UNSAT)
            if ($\Delta > l$)
                return (S, $l + 1$)
            else
                $\Delta \leftarrow 2 \cdot \Delta$
```

Figure 9. The top-level BMC procedure.

$P_i = 0$ ($P_i = l_i$) for time step $t = t_i - \Delta - 1$ ($t = t_i + l_i + \Delta + 1$). Then, we generate the CNF problem by simplifying the expression of the unrolling, i.e., by propagating the added unit clauses, and then removing all satisfied clauses.

The SAT run may end up with three possible results:

- If the SAT solver is not able to determine the instance satisfiability due to time/memory constraints, the process stops and the last valid schedule is returned.

- If the problem is proved to be satisfiable, a new (shorter) schedule has been found. Thus, the original heuristic schedule is replaced by the newer one (computed by function TRACE from the SAT counter-example), and a new value for the latency is tried.

- If the problem is unsatisfiable, there are two sub-cases. If we are analyzing the exact problem ($\Delta > l$, no simplification is performed), then no schedule shorter than the last one can be found, so that the process stops. Otherwise, the unsatisfiability may be due to the simplification introduced by the prediction. In this case, we increase the value of Δ, and we re-run the SAT analysis.

As a final remark, notice that, in the real implementation, the algorithm in Figure 9 is made effective by applying the incremental SAT paradigm. The CNF problem corresponding to the initial (highest) bound is loaded once and for-all within the SAT solver structure. After that, when a valid scheduling for bound l is found and a shorter one (with latency $l - 1$) has to be looked-for, only a set of unit clauses, asserting $s_{l-1} = \mathsf{T}$, is added to the previous problem. The effect of these assignments is causing the SAT solver to remove all the clauses generated for the last time frame, i.e., the ones expressing $\mathsf{TR}(s_{l-1}, s_l)$, as they are immediately satisfied.

We adopt the same paradigm within the window based simplification of function SIMPLIFY. The job performed by this function is actually to provide the SAT solver with a set of unit assumptions [17]. Although the clauses satisfied by these assumptions are not deleted from the solver database, they do not participate in the SAT search, thus obtaining the same simplification effect.

5.2 Timed Automata

Since their introduction [2], timed automata have established themselves as a modeling formalism for describing real-time system behavior. More recently they have been adapted to target time-optimal scheduling and planning problems [6, 1].

A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. Clocks can be reset at certain transitions, and their value can be used as conditions to enable or disable transitions.

Systems are modeled as networks of timed automata, i.e., by representing each component of the system as a timed automata and adopting parallel composition to simulate the overall behavior. Handshake is used to synchronize the components. When two automata have to synchronize, one of them will generate a signal and the other one will wait for it. More specifically, if an automaton has one transition labelled $s!$, it generates the signal s and waits on that transition till the signal is received by another automaton. If an automaton has one transition labelled $s?$, it waits on that transition the signal s generated by another automaton. If there is more than one possible choice for communication channels, the selection is made non-deterministically. To force transitions without delay, the concept of *committed* locations [5] can be adopted.

In this framework, an optimal schedule corresponds to a shortest path in the resulting timed automata. Current timed automata tools are able to deliver both optimal and pseudo-optimal results. Optimal methods are often implemented as standard model checking procedures [5]. Pseudo-optimal methods are often implemented as branch and bound algorithm for optimal reachability analysis. In UPPAAL CORA [6], for instance, branching is based on various search strategies such as breadth-first, depth-first, best-first, random, random-restart, etc. Bound is based on a user-supplied, lower-bound estimate of the remaining cost to reach the goal.

Example 4. *Let us suppose to have the scheduling problem presented in Figure 10, where the references (Rs) A, B, and C require T_A, T_B, and T_C time units to be completed, respectively.*

Figure 11 shows a possible representation of the problem through timed automata.

For each reference (A, B, and C), one automaton (Figure 11 (a), (b) and (c), respectively) is created. For each automaton the variable t is the local clock. The local clock is initially set to 0. The invariant $t \leq T$ makes the automaton stay in the Exe state for T time units. On the contrary, the guard $t > T$ allows it to move to the next state when this condition is triggered. The S labels identify synchronization channels; two channels

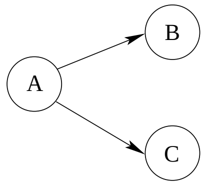


Figure 10. DM dependency: An example.

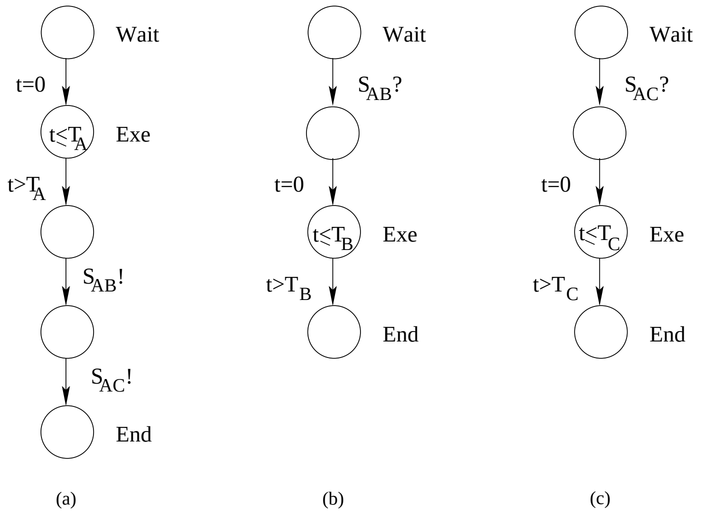


Figure 11. An example: The Timed Automata model.

(AB and AC with signals S_{AB} and S_{AC}) allow a complete synchronization among the three processes.

The evolution of the three automata can be described in the following way (we do not represent the composed automaton for sake of simplicity). The reference A can be executed without delay, as it does not need any synchronization to start its execution (see Figure 10). When it starts, the local time t is set to 0. The automaton stays for T_A time units in the Exe state, and it moves to the state following Exe afterward (through the edge labelled $t > T_A$). On the following two transitions, this automaton enables the other two automata, i.e., the ones for the references B and C, with the two synchronization commands $S_{AB}!$ and $S_{AC}!$, respectively. When B and C receive their own synchronization signals ($S_{AB}?$ and $S_{AC}?$, respectively), they start their execution for T_B and T_C time units. The goal state of the composed behavior is represented by the condition in which the three automata A, B and C are all in their final state End. A breadth-first visit of the composed automaton starting with all references in their initial $Wait$ states, and ending in their End states, gives the smallest latency scheduling, minimizing the time required to perform the entire planning problem. To express this visit, it is possible to adopt model checking by expressing the following property:

$$E\ (A.End\ \wedge\ B.End\ \wedge\ C.End)$$

which requires that a path leading to the End state of each automaton exists. A breadth-first analysis of the system gives the exact solution. A branch-and-bound analysis estimates it.

5.3 Coloured Petri Nets

Coloured Petri Nets (CPNs) are a modeling language developed for systems in which communication, synchronization and resource sharing play an important role. Typical examples of application areas are communication protocols, distributed and embedded systems, automated production systems, and work flow analysis. CPNs combine the strengths of ordinary Petri Nets with the strengths of a high-level programming language. Petri Nets provide the primitives for process interaction, while the programming language supplies the primitives for the definition of data types and the manipulations of data values.

A CPN model consists of a set of modules containing a network of places (represented by circles), transitions (represented by rectangles), and edges. Each place contains a set of markers called tokens. In contrast to low-level Petri Nets (such as place/transition nets) each token carries a data value which belongs to a given type. Therefore, they can be distinguished from each other. To be able to occur, a transition must have enough tokens on its input places, and these tokens must have values that match the corresponding edge expressions.

The modules interact with each other through a set of well-defined interfaces, in a similar way as known from many modern programming languages. The graphical representation makes it easy to see the basic structure of a complex CPN model, i.e., understand how the individual processes interact with each other.

Simulation of CPN models is often used for early investigation of the design, while more formal analysis methods are used for validation. In an automatic simulation the steps are selected by the random CPN simulator which enables and fires the selected transitions. Formal analysis is usually based on occurrence or reachability graphs, in which a node of the graph is built for each reachable marking.

Example 5. *Figure 12 reports the CPN model corresponding to Figure 10. As the different automata are represented by tokens with different characteristics, one single module is sufficient to represent the overall behavior.*

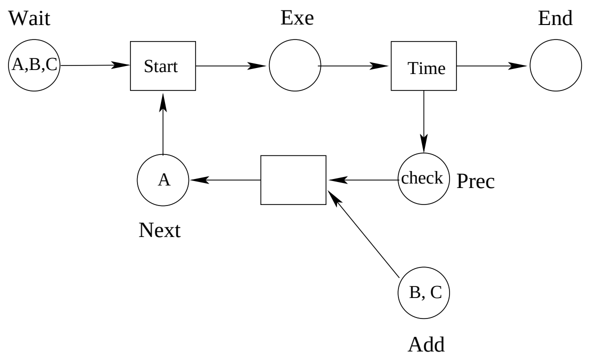


Figure 12. An example: The CPN model.

At the beginning of the process, the places:

- *Wait contains one token for each reference, i.e., tokens A, B and C.*
- *Next contains one token for each reference which does not have any required conditions, i.e., A.*
- *Add contains one token for each reference with at least one required condition, i.e., B and C.*

At the beginning only DM A is enabled (fired) by the transition Start. As a consequence, it executes in the Exe place while transition Time evaluates its running time (i.e., it evaluates T_A, T_B and T_C for each passing token A, B and C). After the transition Time the tokens head to their final place End, but while doing that their passage is checked by place Prec. This place checks for all required conditions of all references and enables the right token to move from the place Add to the place Next where it enables a new token to move from the initial Wait place.

6. Experimental Results

In this section we present our results on real cases of aircraft maintenance obtained from Alenia Aerospace. We compare the SAT-based tool with the BDD-based one and with Timed Automata and Petri Nets. The comparison involves both formal methods and pseudo-optimal strategies.

To fully grade the previously described techniques, we evaluate them on different maintenance processes. These are obtained considering two different airplanes (denoted as A and B in the sequel), accounting for several maintenance activities. In both cases, the original data bases are made up of several hundreds of DMs and we extract from them two scalable families of experiments with increasing size and complexity. Maintenance activities of model B are more constrained than the ones for model A. Table 1 gives some meaningful data on a selected subset of the benchmarks we generated. It reports the number of DMs appearing in the problem (column # DM), the average latency time for DMs (column Latency), the average number of required conditions for DMs (# RC), the total number of mutual exclusions (# ME), and the number of resource classes involved in the problem (# Res). As the resource availability is known only when the maintenance activities are performed,

Table 1. Details on a few problems from aircraft maintenance. # DM: Number of DMs; Latency: Average latency time; # RC: Average number of required condition; # ME: Total number of mutual exclusions; # Res: Number of resource classes.

# DM	DB A				DB B			
	Latency	# RC	# ME	# Res	Latency	# RC	# ME	# Res
20	15.8	1.2	14	5	20.6	1.8	32	6
60	16.0	1.3	40	8	21.7	1.8	122	8
100	16.2	1.4	106	10	22.1	1.9	180	12
200	16.1	1.5	142	14	21.4	2.0	387	18
300	16.2	1.4	190	21	21.8	1.9	522	26

we handle the worst possible situation. For each resource class, only one functional unit is assumed to be available, resulting in harder scheduling problems, i.e., with the highest solution latency.

Table 2 shows the advantages of the thermometric encoding with respect to the logarithmic one when a SAT approach is applied (see Section 5.1.1). For each encoding, we provide the size of the transition relation in terms of total number of CNF variables (# Variable), clauses (# Clause), and average clauses' length (Length). The given data clearly show that the thermometric encoding represents the constraints introduced in Sections 5.1.1 and 5.1.2 in a much more compact way. The logarithmic encoding reduces the number of state variables[3], but this advantage is paid through a much larger amount of long clauses when the transition relation is represented in CNF format.

Table 2. Statistics for the logarithmic and the thermometric encoding for TR. # DM: Number of DMs; # Variable: Total number of CNF variables; # Clause: Total number of clauses; Length: Average clauses' length.

# DM		Logarithmic			Thermometric		
		# Variable	# Clause	Length	# Variable	# Clause	Length
	20	636	2591	4.1	758	892	2.0
	60	1899	8659	4.1	2684	2912	2.1
A	100	3161	13807	4.2	4502	4856	2.1
	200	6605	30076	4.2	8462	9778	2.1
	300	9539	43400	4.2	13291	14478	2.1
	20	725	3784	4.2	902	1084	2.1
	60	2322	14497	4.3	2744	3573	2.2
B	100	3903	25042	4.3	4783	5918	2.2
	200	8991	65226	4.4	9286	12834	2.3
	300	13348	98337	4.4	14569	19309	2.3

To implement the timed automata model, we used the UPPAAL and UPPAAL CORA tools [6, 5, 4]. UPPAAL [5] is a tool box for modeling, simulation and verification of real-time systems, based on constraint-solving and on-the-fly techniques, developed jointly by the Uppsala and the Aalbor University. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and shared variables. The model is further extended with bounded discrete variables that are part of the state and are used as in programming languages. A state of the system is defined by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge separately or synchronize with another automaton, which leads to a new state. UPPAAL uses a continuous time model. The model-checker is based on the theory of timed automata and its modeling language offers additional features such as bounded integer variables and urgency. The query language of UPPAAL, used to specify properties to be checked, is

3. Notice that, the numbers reported in the table include the auxiliary variables introduced by the conversion process from BDDs to CNF, see Section 5.1.2.

a subset of CTL (Computation Tree Logic) and it is sufficient to implement the check described in Section 5.2.

For the Coloured Petri Nets approach we used the CPN-tools [25, 26]. It has been developed at the University of Aarhus, Denmark. The tools combine the strength of Petri Nets with the strength of programming languages. Petri Nets provide the primitives for the description of the synchronization of concurrent processes, while programming languages provide the primitives for the definition of data types and the manipulation of data values. This representation is the foundation for the definition of the different behavioral properties and the analysis methods. CPN models can be made with or without explicit reference to time. Un-timed CPN models are usually used to validate the functional/logical correctness of a system, while timed CPN models are used to evaluate the performance of the system. CPN also offers formal verification methods, such as state space analysis and invariant analysis, to prove that a system has a certain set of behavioral properties.

For the BDD-based and SAT-based approaches we implemented a home-made symbolic scheduling tool. The BDD-based approach is built on top of the Colorado University Decision Diagram (CUDD, version 2.4.1) package [31]. The SAT-based approach uses the Minisat [16] tool (version p_v1.14) as default (linked) SAT solver.

Our experiments were run on a Pentium IV 1.7 GHz Workstation with 1 GByte of main memory. Notice that while UPPAAL, UPPAAL CORA and CPN-tools run under Win-

Table 3. Results (in seconds) on aircraft maintenance provided by exact engines. A dash (−) means overflow on time (1800 s) or memory (1 GB). When the time limit is reached, the latency time of the last solution found is reported between parenthesis.

# DM		Latency	UPPAAL	CPN-tools	BDDs		SAT	
					Log	Therm	Log	Therm
	20	78	1	1	12	1070	37	1
	40	95	22	5	368	–	832	11
	60	128	134	37	–	–	1800 (151)	47
	80	163	581	183	–	–	1800 (239)	115
A	100	188	–	1279	–	–	–	276
	150	231	–	–	–	–	–	611
	200	267	–	–	–	–	–	1229
	250	302	–	–	–	–	–	1800 (304)
	300	329	–	–	–	–	–	1800 (345)
	20	104	3	2	84	–	168	5
	40	152	73	44	955	–	1800 (207)	59
	60	187	419	248	–	–	–	174
	80	219	1452	1021	–	–	–	392
B	100	253	–	–	–	–	–	668
	150	296	–	–	–	–	–	1105
	200	334	–	–	–	–	–	1748
	250	389	–	–	–	–	–	1800 (407)
	300	413	–	–	–	–	–	1800 (458)

dows, the home-made BDD and SAT-based package run under the Debian GNU/Linux 4.0 operating system.

The first comparison we perform concerns the results given by the symbolic optimal methods. Table 3 presents data collected by applying the different exact strategies we have analyzed, in terms of CPU time, with a time limit of 1800 seconds. The table is divided in two parts, as it shows data for both the aircraft processes (A and B). More specifically, it provides the number of DMs appearing in each instance, the value for the optimal latency, and then the CPU time (in seconds) required by each tool to find the solution of the problem. For the BDD and SAT approaches, we report results for both the logarithmic (column Log) and the thermometric (column Therm) encoding.

The following observations can be made. Though all the engines are able to solve the easy instances, the problems become very soon infeasible for BDDs, due to the state explosion problem, even when the number of state variables is controlled through the logarithmic encoding. The methodologies offered by CPN-tools and UPPAAL behave better, but they give up when the number of DMs is more than one hundred. The SAT technique exploiting the thermometric encoding seems to be the only able to go beyond this threshold, delivering the exact solution even with 200 DMs. For larger instances (i.e., 250 and 300 DMs), the SAT method is not able to find the optimal solution within the given time limit, thus we report this time (1800 seconds) and the latency (between parenthesis) of the last solution found by the algorithm in Figure 9. On the other side, the SAT approach is absolutely not effective when the logarithmic encoding is used. This fact definitely proves that the thermometric encoding is better than the logarithmic one when solving this kind of problems through SAT.

The previous results can be partially explained, however, by considering that both the CPN-tools and UPPAAL software are all-purpose tools which have been designed to model generic behaviors, whereas our tool, with the underlying SAT formulation, is dedicated to face exactly this problem.

In the second set of experiments performed, we compare on our benchmarks a few SAT tools. Results are reported in Table 4, where we compare the solvers in terms of CPU time and memory efficiency, on the same set of experiments reported in Table 3 for the data base B.

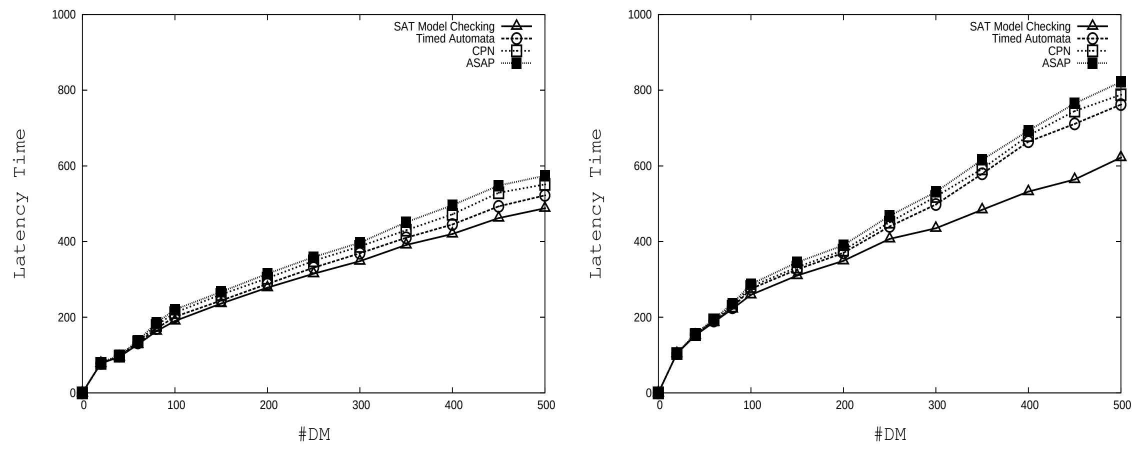
Table 4. SAT solver comparison on the harder of the two data bases (B). A dash (−) means overflow on time (3 hours) or memory (1 GB).

# DM	Minisat		PicoSAT		RSat		March_KS	
	Mem	Time	Mem	Time	Mem	Time	Mem	Time
20	16.3	22	15.3	20	17.6	24	100.4	172
40	32.5	206	30.3	132	36.2	97	219.4	524
60	93.7	696	110.9	1188	119.1	414	332.9	1480
80	117.7	1566	117.3	1668	133.2	1820	559.3	8852
100	186.4	3430	214.7	3301	211.4	2935	784.7	–
150	394.1	6630	440.7	5568	457.7	4944	–	–
200	489.4	10236	493.1	8908	620.2	7754	–	–

Given the very bad performance obtained with the logarithmic encoding, we concentrate with the thermometric encoding. We used the best ranking solvers [7], namely Minisat [16] (version 2.0), PicoSAT [8] (version 535), RSat [19] (version 2.01) and March_KS [15] (version 06.03.2007). All SAT solvers are exploited as external tools, throughout a file-based interface. This implies that we could not adopt the incremental SAT paradigm, which explains the worse performance of the SAT tools compared to the ones given in Table 3. Therefore, we increased the time limit to 3 hours to perform these experiments. Overall, the generated experiments seem hard-to-solve even for modern optimized SAT tools. RSat seems to behave slightly better. Between Minisat and PicoSAT there is no clear winner. March_KS is the slower of the group and it runs out of time already with 100 DMs.

The third set of experiments done (presented in Figure 13) focuses on the pseudo-optimal (heuristic) techniques coming with all the previous tools. It is well known [29, 13] that heuristic approaches are able to cover even very large instances, usually quickly finding a good solution for lightly constrained problems. Here, we provide the results obtained with the branch and bound scheme of UPPAAL, with the simulative mode of CPN-tools, with the As Soon As Possible (ASAP) heuristic strategy, and with a pseudo-optimal SAT-based method exploiting the thermometric encoding. More precisely, for the SAT approach we present data obtained by running the algorithm introduced in [11], instead of the one given in Section 5.1.4. As previously mentioned, in this case a binary search for the optimal latency is performed. However, every SAT run is executed under a (small) time limit, interpreting an overflow result just like unsatisfiability. Thus, a possible solution for a given bound may be missed (due to the time limit for the search), forcing the algorithm to return a sub-optimal result. For this work, we have set the time limit for each SAT search to 1 minute.

Figure 13. Results on aircraft maintenance provided by pseudo-optimal engines.



Aircraft Model A Aircraft Model B

Figure 13 plots the obtained values for the latency time (for model A and B) with respect to the number of data module in a range up to 500. Again, the graphs show the advantages of the SAT method, even in its original formulation, in terms of the accuracy of the results. This is paid in terms of the CPU time and memory necessary to find the solution. In fact, while the heuristic procedures required only a few seconds (at most) and a few MBytes of memory in all cases, the SAT scheduling technique needed several hundred MBytes for the largest instances, with total run-times up to 10 minutes.

Finally, we provide an experimental evaluation of the simplification effect obtained with the method introduced in Section 5.1.4. As described in that section, function SIMPLIFY enables each operation in a window centered in the operation predicted scheduling time, and of width $2 \cdot \Delta$. Table 5 shows, for problems with an increasing number of DMs, the number of clauses alive, i.e., not immediately satisfied, after the initial propagation of unit clauses and assumptions. The total number of clauses of the exact case is drastically reduced for the window-based search even when adopting quite large windows. For example, for the last experiment of the data base B, with a window of width 128 the number of clauses of the problem is reduced from about 9 millions to about 1.9 million. As the bound in that case is equal to 460 and $2 \cdot \Delta = 256$, this means that we reduce the number of problem clauses by a factor of almost 5 by allowing each operation to be scheduled in more than 50% of the time steps. In general, the number of clauses may be easily reduced by 5–10 times.

Table 5. Simplification effect due to the window search.

# DM	DB A				DB B			
	Bound	Exact	Window		Bound	Exact	Window	
		# Clauses	Δ	# Clauses		# Clauses	Δ	# Clauses
20	80	53412	4	9828	110	96207	4	23523
60	150	436948	16	68269	200	714826	16	121680
100	210	1012074	32	188913	270	1598102	32	298814
200	280	2159540	64	342758	350	3664972	64	588636
300	370	5357568	128	1398236	460	9076272	128	1890672

7. Conclusions

In this paper, we presented a new application of standard scheduling and planning algorithms to the field of aircraft maintenance.

We modeled the problem adopting different techniques, varying from heuristic scheduling, symbolic BDD-based and SAT-based scheduling, Priced Timed Automata and Petri Nets. We specifically concentrate on SAT-based scheduling, presenting an algorithm that combines exact and heuristic approaches, trading-off scalability and optimality of the results. We used the tools to target both optimal and sub-optimal (approximated) responses to our problems, depending on their size, total latency, and overall complexity. We compared the different models, in terms of description power, efficiency, and accuracy of the results. We also compared different state-of-the-art SAT solvers in terms of CPU time and memory.

Experiments on real, very large, and highly constrained problems enabled the following considerations. General purpose tools, albeit easy to apply to the problem, proved to have a low scalability. On the contrary, ad-hoc SAT-based solutions were in general more efficient and scalable. Our technique was able to solve instances $3 - 4$ times larger in terms of operations to be scheduled.

Some interesting results have also been put forward regarding the performances of the adopted SAT solvers on the specific verification instances generated by our problem. As far as SAT solvers are concerned, on our benchmarks we showed that RSat has a small edge on all other SAT tools.

Acknowledgments

The authors would like to thank the Alenia Aerospace, ISLT – Logistic Engineering, Torino, Italy, for the industrial aircraft maintenance databases used in this paper.

References

[1] Y. Abdeddaim, A. Kerbaa, and O. Maler. Task Graph Scheduling using Timed Automata. In *IPDPS*, 2003.

[2] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, **126**(2):183–235, 1994.

[3] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers)*, **3542** of *LNCS*, pages 1–15. Springer-Verlag, 2004.

[4] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, **3185** of *LNCS*, pages 200–236. Springer-Verlag, 2004.

[5] G. Behrmann, A. David, and K. G. Larsen. The Uppaal Tool, http://www.uppaal.com/, 2006.

[6] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal Scheduling using Priced Timed Automata. In *SIGMETRICS Performance Evaluation Rev.*, **32**(4), pages 34–40, 2005.

[7] D. Le Berre, O. Roussesl, and L. Simon. The International SAT Competitions Web Page, http://www.satcompetition.org/, 2007.

[8] A. Biere. Picosat SAT Solver, http://fmv.jku.at/picosat/, 2006.

[9] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering*, **6**(3):116–128, 1991.

[10] A. Burns and A. J. Wellings. The Notion of Priority in Real-Time Programming Languages. *Computer Languages*, **15**(3):153–162, 1990.

[11] G. Cabodi, A. Kondratyev, L. Lavagno, S. Nocco, S. Quer, and Y. Watanabe. A BMC-Based Formulation for the Scheduling Problem of Hardware Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, January 2005.

[12] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals. In *Proc. Design Automation & Test in Europe Conf.*, pages 898–903, Munich, Germany, March 2003. IEEE Computer Society.

[13] O. Dain, D. Etherington, M. Ginsberg, O. Keenan, and T. Smith. Automatic Scheduling to Minimize Shipbuilding Cost. In *12th International Conference on Computer Applications in Shipbuilding*, pages 41–54, Busan, Korea, 2005.

[14] O. Dain, M. Ginsberg, E. Keenan, J. Pyle, T. Smith, A. Stoneman, and I. Pardoe. Stochastic Shipyard Simulation with SimYard. In L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, editors, *Winter Simulation Conference*, pages 1770–1778, 2006.

[15] M. Dufour, M. Heule, J. van Zwieten, and H. van Maaren. March SAT Solver, http://www.st.ewi.tudelft.nl/sat/download.php, 2006.

[16] N. Eén and N. Sörensson. Minisat SAT Solver, http://minisat.se/, 2003.

[17] N. Eén and N. Sörensson. Temporal induction by incremental sat solving. In *BMC'03: First International Workshop on Bounded Model Checking*, Boulder, Colorado, July 2003.

[18] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-completeness*. Freeman, 1979.

[19] UCLA Automated Reasoning Group. RSat SAT Solver, http://reasoning.cs.ucla.edu/rsat/, 2006.

[20] B. Guyot and J.-M. Janik. Simplified method of binary/thermometric encoding with an improved resolution, U. S. Patent No. 649676, 2002.

[21] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California Santa Barbara, USA, December 2000.

[22] S. Haynal and F. Brewer. Efficient Encoding for Exact Symbolic Automata–Based Scheduling. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 477–481, San Jose, California, November 1998.

[23] S. Haynal and F. Brewer. Automata-Based Scheduling for Looping DFGs. Technical report, University of California Santa Barbara, USA, October 1999.

[24] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A Formal Approach to the Scheduling Problem in High-Level Synthesis. *IEEE Trans. on Computer-Aided Design*, **10**:464–475, April 1991.

[25] K. Jensen. Coloured Petri Nets Tools, http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

[26] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science, Springer-Verlag, 1997.

[27] S. O. Memik and F. Fallah. Accelerated SAT-based Scheduling of Control/Data Flow Graphs. In *Proc. Int'l Conf. on Computer Design*, pages 395–400, 2002.

[28] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, Hightstown, New Jersey, 1994.

[29] P. G. Paulin and J. P. Knight. Force–Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, **8**:661–679, June 1989.

[30] I. Radivojevic and F. Brewer. A New Symbolic Technique for Control-Dependent Scheduling. *IEEE Trans. on Computer-Aided Design*, **15**(1):45–47, January 1996.

[31] F. Somenzi. CUDD: CU Decision Diagram Package – Release 2.4.1, http://vlsi.colorado.edu/~fabio/CUDD/, 2005.

[32] Toby Walsh. Sat v csp. In Rina Dechter, editor, *CP*, **1894** of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

[33] W. Zhao and K. Ramamritham. Distributed Scheduling Using Bidding and Focussed Addressing. In *6th IEEE Real-Time Systems Symposium*, pages 103–111, December 1985.

[34] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive Scheduling under Time and Resource Constraints. *IEEE Trans. on Computers*, **38**(8):949–960, 1987.