A Novel SAT-Based Approach to the Task Graph Cost-Optimal Scheduling Problem

A Novel SAT-Based Approach to the Task Graph Cost-Optimal Scheduling Problem

Sergio Nocco and Stefano Quer

***Abstract*—The task graph cost-optimal scheduling problem consists in scheduling a certain number of interdependent tasks onto a set of heterogeneous processors (characterized by idle and running rates per time unit), minimizing the cost of the entire process. This paper provides a novel formulation for this scheduling puzzle, in which an optimal solution is computed through a sequence of binate covering problems, hinged within a bounded model checking paradigm. In this approach, each covering instance, providing a min-cost trace for a given schedule depth, can be solved with several strategies, resorting to minimum-cost satisfiability solvers or pseudo-Boolean optimization tools. Unfortunately, all direct resolution methods show very low efficiency and scalability. As a consequence, we introduce a specialized method to solve the same sequence of problems, based on a traditional all-solution SAT solver. This approach follows the "circuit cofactoring" strategy, as it exploits a powerful technique to capture a large set of solutions for any new SAT counter-example. The overall method is completed with a branch-and-bound heuristic which evaluates lower and upper bounds of the schedule length, to reduce the state space that has to be visited. Our results show that the proposed strategy significantly improves the blind binate covering schema, and it outperforms general purpose state-of-the-art tools.**

***Index Terms*—Formal methods, SAT, satisfiability, scheduling, symbolic techniques.**

I. Introduction

IN THE DESIGN of integrated circuits, *scheduling* [1]–[3] consists in a manual or automatic process, during which the designer decides *when* computational operations and communication transactions take place. Scheduling is often performed with *binding*, that establishes on *which* resource each operation or transaction has to be run. The decisions taken by the scheduler and the binder are constrained by operand dependences, resource availability, control decisions, and global costs. The target of this process is usually to minimize the total number of time steps, i.e., the so called *latency* (or *makespan*), necessary to complete all tasks in the given system.

A. Related Works

Symbolic manipulations, based on both binary decision diagrams (BDDs) and satisfiability solving (SAT), have recently attained interesting results in the framework of hardware scheduling [4]–[9], as an alternative to integer linear programming [10] and heuristic techniques [11], [12]. The key idea of the symbolic approach is to use a set of nondeterministic finite automata to describe the behavior of each operation in the system. The automata are then composed, and the resulting state space is visited adopting state-of-the-art model checking techniques. In the simplest case of systems without control choices (i.e., *if-then-else* constructs), a schedule is a path. Therefore, symbolic scheduling works just like invariant checking with counterexample extraction, i.e., forward or backward traversals provide a scheduling solution as a trace connecting initial and terminal states. However, in control-dependent behavior *fork* and *join* nodes are introduced to represent scheduling choices, depending on values of control operands. This model requires a specific BDD-based backward traversal procedure (called *validation* in [13]), which does not directly correspond to standard model checking procedures.

In [6], [8], [14], and [9] Cabodi *et al.* adapted the previous model to support conventional model checking procedures for control-dependent systems. More specifically, alternative sub-traces were transformed into concurrent behaviors, subsequently followed in parallel. The resulting schedule was always a path (instead of a tree) connecting initial and final states. As a by-product, the authors exploited a traditional SAT-based bounded model checking paradigm to find the scheduling solution.

At the same time, several attempts have been made to apply tools designed for real-time and hybrid systems, usually exploiting timed automata [15], to solve realistic scheduling problems [16], [17]. In this area, the timed automata formalism can be enriched by allowing the accumulation of costs during behavior [18]–[20]. This leads to priced timed automata models and consequently to a different optimization target, i.e., minimizing the overall cost associated with a feasible schedule instead of the latency. One concrete example for this problem is the task graph cost-optimal scheduling, consisting in mapping a number of interdependent tasks (e.g., performing some arithmetic operations) onto a set of heterogeneous processors. Each processor is annotated with both idle and running costs (representing, for example, power dissipation). The target is then to reduce the cost of the entire schedule trace, taking into account the execution and idle rates of all resources. This concept of optimality is normally unsupported by standard verification algorithms. Indeed, although the task

Manuscript received January 28, 2010; revised May 4, 2010; accepted June 17, 2010. Date of current version November 19, 2010. This paper was recommended by Associate Editor W. Kunz.

The authors are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin 10129, Italy (e-mail: sergio.nocco@polito.it; stefano.quer@polito.it).

Digital Object Identifier 10.1109/TCAD.2010.2061631

graph cost-optimal scheduling problem has several applications in the electronic design automation (EDA) domain, only a few methodologies are currently available for solving it exactly. Among them, we mention mixed integer linear programming [21], [22], and priced timed automata [23]–[25].

B. Proposed Methodology

In this paper, we propose a novel formulation of the task graph cost-optimal scheduling problem, as an alternative to previously used formalisms. Starting from the symbolic SAT-based approach to high-level synthesis scheduling proposed by Cabodi *et al.* [6], [8], [14], we:

1) allow each task to be executed by several (but usually not all) resource units;
2) incorporate the idle and running costs for each resource;
3) shift the target from minimizing the global latency to optimizing the total scheduling cost.

In our approach, an optimal solution is obtained through a sequence of binate covering instances [26]–[28]. Each of these instances can be solved by both minimum-cost satisfiability [29], [30] and pseudo-Boolean optimization [31]–[33] solvers. Nevertheless, our experiments prove that modern tools for the binate covering problem still provide a limited scalability. As a consequence, we present a novel resolution technique, based on conventional SAT engines, that drastically improves the binate covering schema. The approach consists in solving the problem using a SAT solver, by inserting a blocking clause for each discovered counter-example, and re-running the SAT routine ("all-solution" SAT [34]–[36]). However, following the "circuit-cofactoring" idea presented by Ganai *et al*. [37], in order to avoid a complete solution enumeration we enlarge each single counter-example to represent a large sub-set of the entire solution space. This optimization leads to a very effective state space pruning, thus drastically reducing the number of enumeration steps. Finally, to further prune the state space that has to be visited, we adopt an efficient branch-and-bound strategy, following similar ideas adopted in other domains [38], [39]. We show how it is possible to compute on-the-fly tight lower and upper bounds for the latency, and to use them to limit the number of binate covering instances that have to be solved.

We present controlled experiments to demonstrate the power of the previous ideas on both synthetic and standard benchmarks. More specifically, we first show that our initial solution, based on blindly solving a sequence of binate covering problems, though feasible, is still less efficient than state-of-the-art priced timed automata tools. Then, we give evidence to the fact that our all-solution SAT procedure is by far more efficient and scalable than the other approaches. Lastly, we detail the pruning power of our branch-and-bound technique.

C. Contributions

This paper includes the following contributions.

1) A new SAT-based formulation for the task graph cost-optimal scheduling problem. This includes the definition of a new automaton model to map tasks onto different units, and a new strategy to represent the accumulation of costs.
2) A complete methodology to solve cost-optimal scheduling problems through bounded model checking, where each instance actually corresponds to a binate covering problem.
3) A fast resolution technique based on an all-solution SAT strategy, with a novel optimization able to capture a large set of solutions for any new SAT counter-example. Such an optimization allows to effectively prune the search space and drastically reduces the number of iterations that must be performed by the algorithm.
4) A branch-and-bound heuristic to dynamically evaluate the latency lower and upper bounds, to further reduce the state space that has to be visited.

As far as we know, this is the first time such extensions are proposed and adopted. Our method shows an edge over state-of-the-art general purpose tools, with a discrete modeling power and generality. The experimental data support these claims, showing an improvement of more than two orders of magnitude in terms of speed-up in some cases.

D. Roadmap

This paper is organized as follows. Section II introduces some background notions on the adopted models, model checking, and scheduling. It also includes an instance of the task graph cost-optimal scheduling problem used as a running example in this paper. Section III describes how to model minimum latency scheduling problems by adopting a SAT-based approach. Section IV presents our contributions to model a task graph cost-optimal scheduling problem and to solve it through a bounded model checking strategy, where each instance is a binate covering problem. Section V shows how to adopt an all-solution SAT method to make the resolution process faster. Section VI introduces our branch-and-bound technique. Section VII discusses experimental results. Finally, Section VIII concludes this paper with some summarizing remarks.

II. Background

A. Model and Notation

In our notation, $B = \{0, 1\}$ indicates the set of Boolean values. Symbols $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ are used to represent the Boolean conjunction (AND), disjunction (OR), negation (NOT), implication, and co-implication, respectively. Finally, the symbol $\times$ denotes the Cartesian product.

The automata we address are usually represented implicitly by Boolean formulas. For our purposes, an automaton is defined as follows.

Definition 1: 1 An *Automaton* is a triple $(\mathsf{I}, \mathsf{TR}, \mathsf{T})$, where:

1) $\mathsf{I} \subset B^m$ is the set of initial states;
2) $\mathsf{T} \subset B^m$ is the set of final (or target) states;
3) $\mathsf{TR} \subseteq B^m \times B^m$ is the transition relation of the system.

The present (next) state space of an automaton is defined by an indexed set of m Boolean variables $P = \{p_1, \ldots, p_m\}$ ($N = \{n_1, \ldots, n_m\}$). The behavior of an automaton is described by its transition relation $\mathsf{TR}(P, N)$, which contains all

pairs (present state p, next state n) such that there is a valid transition from p to n.

B. Satisfiability and Minimum-Cost Satisfiability

Given a Boolean formula f depending on n variables $(x_1, x_2, \ldots, x_n)$, the Boolean satisfiability problem [40], usually known as SAT, consists in finding, if it exists, an assignment to the variables of f, namely $(v_1, v_2, \ldots, v_n)$, such that f is true, i.e., $f(v_1, v_2, \ldots, v_n) = 1$.

The minimum-cost satisfiability problem (min-cost SAT) [29], [30] is a SAT problem which minimizes the cost of the satisfying assignment. Given a set of costs $c_1, \ldots, c_n$, with each $c_i \geq 0$ associated to a variable x_i, the goal is to find a variable assignment that satisfies f and minimizes

$$\sum_{i=1}^{n} c_i \cdot v_i \tag{1}$$

where each $v_i \in \{0, 1\}$ is the value assigned to a variable x_i in the counter-example. A SAT problem can be viewed as the special case of a min-cost SAT instance with all $c_i = 0$.

C. Covering Problems

The covering problem [26]–[28] has been widely studied, and it has many applications in logic synthesis and Boolean minimization [1]. Given a set of n Boolean variables $(x_1, x_2, \ldots, x_n)$, it consists in minimizing a cost value [expressed as in (1)], provided that a set of linear constraints are satisfied. The ith constraint of a covering instance C can be represented as

$$\sum_{j=1}^{n} a_{ij} \cdot x_j \quad \geq \quad b_i. \tag{2}$$

If each a_{ij} is in the set $\{0, 1\}$ and $b_i = 1$, then C is an instance of the unate covering problem. If every a_{ij} is in the set $\{-1, 0, 1\}$ and $b_i = 1 - |\{a_{ij} : a_{ij} = -1, 1 \leq j \leq n\}|$, then C is an instance of the binate covering problem.

It is worth noticing that there is a strong relationship between the binate covering problem and min-cost SAT. Given the restrictions on the a_{ij} and b_i coefficients, it is indeed possible to prove that every constraint specified by (2) can be expressed as a propositional clause, and vice-versa. This means that a solution for a binate covering problem can be obtained through a min-cost SAT solver, and that a min-cost SAT instance can be actually solved by means of any other approach developed for the binate covering case.

In turn, the covering problem can be viewed as a special case of the more general pseudo-Boolean optimization problem [31]–[33]. In fact, a pseudo-Boolean optimization problem can be formulated as a covering problem, with the only difference that b_i and all a_{ij} are integer numbers without restrictions. Thus, any covering instance can be solved through a pseudo-Boolean solver.

The above observations will be taken into account in Section VII for the experimental evaluation, but they do not influence the underlying theory. Henceforth, without loss of generality, in the sequel we will just mention min-cost SAT instances, implicitly referring to binate covering problems that could be solved by both min-cost SAT and pseudo-Boolean optimization solvers.

D. Bounded Model Checking

Finding transition sequences with SAT [41]–[43] has been applied to model checking since the introduction of SAT-based bounded model checking (BMC) [44]. BMC builds a propositional formula that is satisfiable *iff* there is a path from I to T of bounded length k. More specifically, a BMC run of depth k unfolds the transition relation k times

$$\mathsf{TR}^k(S_0, \ldots, S_k) \quad = \quad \bigwedge_{0 \leq j < k} \mathsf{TR}(S_j, S_{j+1})$$

and uses a SAT solver to check the satisfiability of the formula

$$path^k(S_0, \ldots, S_k) = \mathsf{I}(S_0) \wedge \mathsf{TR}^k(S_0, \ldots, S_k) \wedge \mathsf{T}(S_k). \tag{3}$$

If such a formula is unsatisfiable, there is no path of length k connecting I and T, and a larger value of k should be tried.

E. Priced Timed Automata

Priced timed automata (PTA) were introduced, independently, in [18] and [19]. They are an extension of timed automata [15], where edges and locations are further labeled with non-negative costs and cost rates. More formally, the following definition may be given.

Definition 2: A *priced timed automaton* is a tuple $(\mathsf{X}, \mathsf{L}, l_0, \mathsf{A}, \mathsf{E}, \mathsf{I}, \mathsf{C})$, where:

1) X is a set of clock variables;
2) L is a finite set of states (or *locations*, in the tradition of the timed automata formalism);
3) $l_0 \in \mathsf{L}$ is the initial location;
4) A is a set of possible actions that could be performed by the automaton while making a transition;
5) $\mathsf{E} \subset \mathsf{L} \times \mathsf{X} \times \mathsf{A} \times \mathsf{L}$ is the set of edges connecting the locations;
6) $\mathsf{I} \subset \mathsf{L} \times \mathsf{X}$ is a function assigning invariants to locations;
7) $\mathsf{C} \subset \mathsf{L} \times \mathsf{E} \times \mathbb{N}_0$ assigns positive integer cost rates and costs to locations and edges, respectively.

Intuitively, a clock is a non-negative real valued variable that can be reset to zero and grows at fixed rate with the passage of time. A timed automaton over a set of clocks X is thus an annotated directed graph with a vertex set L, an edge set E and a distinguished vertex l_0. Edges are labeled with guard expressions and an action set. A guard g is the conjunction of simple constraints $x \bowtie k$, x being a clock in X, k a non-negative integer value, and $\bowtie \in \{<, \leq, =, \geq, >\}$. An edge is enabled when its guard g evaluates to true and the source location is active. An action set $a \subset \mathsf{A}$ may include resetting a sub-set of the clocks in X or modifying the value of some variables. According to the PTA semantics, all actions associated to an edge are performed when the edge is taken. All locations may be labeled with invariants. An invariant i is the conjunction of simple conditions $x \prec k$, x being a clock in X, k a non-negative integer value, and $\prec \in \{<, \leq\}$. The semantics of (priced) timed automata requires that an invariant must evaluate to true whenever its location is active. Finally, both edges and locations may be annotated with costs and cost rates, respectively. When an edge is taken, it contributes to the total cost with its own cost; when a location is active,

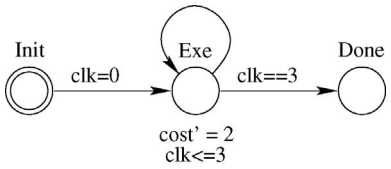

Fig. 1. Very simple PTA.

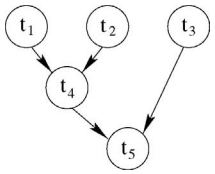

Fig. 2. Example of data flow graph.

it contributes to the total cost with an amount equal to the product of its own cost rate and the number of time steps it is active.

The following example describes a very simple PTA. A more complex case, showing how a task graph cost-optimal scheduling problem can be modeled through PTA, is illustrated in Example 3.

Example 1: Fig. 1 shows a very simple PTA with three states. The variable *clk* is set equal to 0 on the transition between state *Init* and state *Exe*, whereas it has to be equal to 3 on the transition from state *Exe* to state *Done*. Costs, with rates indicated by the variable *cost′*, are accumulated in state *Exe*. As a consequence, the *Done* state can be reached with minimum cost equal to 6 ($3 \cdot 2$). □

F. High-Level Synthesis Scheduling

As described in the introduction, scheduling [1]–[3] consists in deciding *when* computational operations and communication transactions take place.[1] Such decisions are constrained by hardware resource availability, operand dependences and control decisions. The following example shows a very simple scheduling problem, in which we target latency minimization.

Example 2: Let us suppose to have the set of operations (or *tasks*) represented by the data flow graph (DFG) of Fig. 2, where it is assumed that all tasks can be executed in one time step by computational units of the same type. Notice that the edges in Fig. 2 represent data dependences, so that, for instance, task t_4 cannot start executing until both tasks t_1 and t_2 have completed. The same consideration holds for task t_5 with t_3 and t_4.

Then, depending on the number of resources used to execute the operations, it is evident that several planning solutions can be found. Fig. 3 shows some of the possible scheduling traces, obtained with a different resource availability.

1) In Fig. 3(a), there is no resource limit and the operations execution is controlled by the data dependences only. Thus, all tasks without dependences can be performed in the first time step, as there are enough computational units to execute them in parallel. The minimal amount of time necessary to complete all operations, i.e., the latency, is equal to three time units.
2) In Fig. 3(b) only two resource units are available. As a consequence, in order to obtain the minimum latency, t_3 has to be delayed until the second time step. The latency is again equal to three time units.
3) In Fig. 3(c) one single computational unit is available. Therefore, only one operation can be performed in each single time step. In this case, the latency is equal to five time steps. □

[1]Notice that the term "planning" usually indicates the process of deciding *which* operations are necessary to perform. As a consequence, most existing planners concentrate on action selection, and most existing schedulers focuses on action execution. Nevertheless, in this paper we use the two terms without distinction.

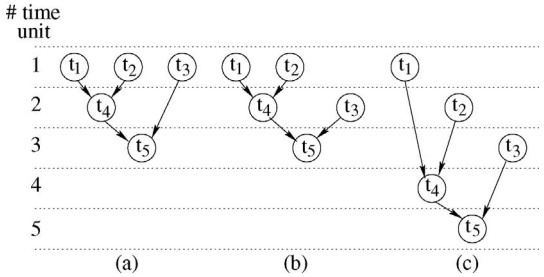

Fig. 3. Scheduling solutions for the DFG of Fig. 2. (a) When there is no resource limit. (b) When only two resource units may be exploited. (c) When just one computational unit is available.

Processor	Idle	Exe
p_1	2	5
p_2	1	4

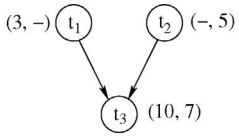

Fig. 4. Simple task graph.

G. Task Graph Cost-Optimal Scheduling

A specific class of planning problems is represented by *task graph cost-optimal scheduling* [45], [46]. Like high-level synthesis scheduling, it consists in planning a set of tasks related with each other by data dependences. Nevertheless, every task may be performed (with distinct execution times) by several (types of) resources, or *processors*, each of which is annotated with idle and running costs. The target is to find a schedule such that the total associated cost is minimized. Notice that, in our problems, task graphs are acyclic in their nature. The following example shows a very simple case of task graph scheduling, targeting the optimization of the global cost.

Example 3: Fig. 4 shows simple a task graph, taken verbatim from [20], where three tasks (t_1, t_2, and t_3) must run on two processors (p_1 and p_2), with idle and running rates (per time unit) represented in the table. Each task is annotated with the required execution times on the processors, that is, t_1 can only execute on p_1 (in 3 time steps), t_2 only on p_2 (in five time units), and finally t_3 can execute on both p_1 (in ten time units) and p_2 (in 7).

Fig. 5 shows the priced timed automata model for such a task graph. It includes five PTA, three for the tasks and two for the processors, which synchronize by message passing over

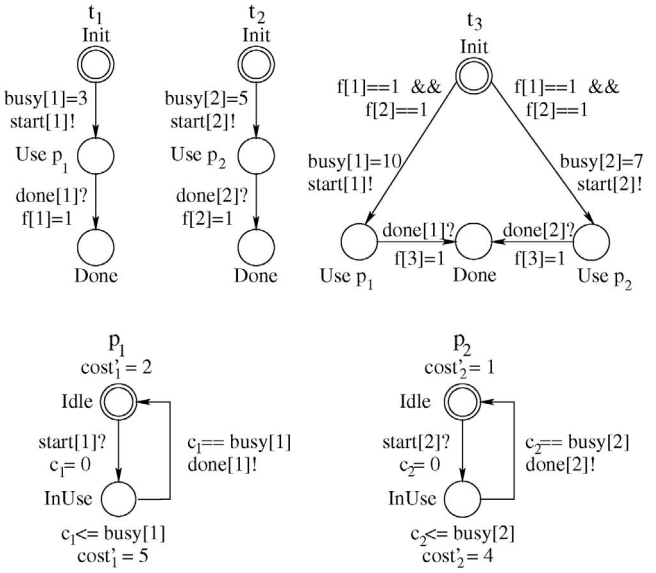

Fig. 5. PTA model for the task graph of Fig. 4.

the global arrays *start*, *busy*, *done*, and f.
In more detail, we obtain the following.

1) The automata for the processors (p_1 and p_2) are formally identical, consisting of only two states (*Idle* and *InUse*), both labeled with the proper cost rate ($cost_i'$). The ith resource automaton moves from the *Idle* to the *InUse* location when a request $start[i]$ is received. In the same transition, the associated clock c_i is also reset. Then, the automaton remains in the *InUse* state for an amount of time equal to $busy[i]$. Finally it returns to the *Idle* state while emitting the $done[i]$ message.
2) The automata for the tasks (t_1, t_2, and t_3) differ according to the number of processors they may execute on. Thus, the PTA for tasks t_1 and t_2 may move from the *Init* to the *Done* locations only through one intermediate state, which models the fact that the task is currently executing on the right processor. On the other hand, the automaton for task t_3 may reach the *Done* state through two different paths, according to the resource used to execute it. In all cases, as soon as the ith automaton moves to its *Done* state, the corresponding $f[i]$ value is set to 1. This action correctly models the data dependences existing between (t_1, t_2) and t_3 (the automaton for t_3 may leave its *Init* state only when both $f[1]$ and $f[2]$ have been set).

An optimal schedule trace corresponds to a path in the resulting composed priced timed automaton which satisfies the property $EF(t_1.Done \wedge t_2.Done \wedge t_3.Done)$, i.e., all three tasks must be in the *Done* state, and minimizes the overall cost. A minimum-cost solution for this task graph problem will be provided in Examples 6 and 7. □

III. SAT-Based Scheduling

As a starting point for solving a planning problem, such as the one represented in Example 2, by means of SAT (see [7]–[9]), each operation is modeled through a non-deterministic automaton, also called basic scheduling automaton (BSA). Each BSA provides information about the execution of the associated operation.

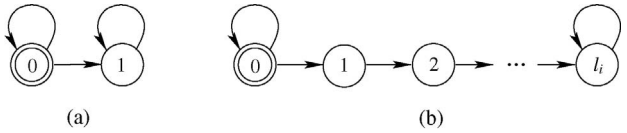

Fig. 6. BSA for an operation i with latency (a) 1 and (b) l_i.

More specifically, for any single-time-step operation i of the given DFG, the related automaton may be represented as in Fig. 6(a), where 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. More formally, as the automaton has only two states, its transition relation may be encoded with exactly two Boolean variables, i.e., $P_i = \{p\}$ for the present state, and $N_i = \{n\}$ for the next state. The meaning of the possible automaton transitions is hence the following.

1) $p = 0$ and $n = 0$: the operation has not been scheduled in previous steps and will not be scheduled in the next one.
2) $p = 0$ and $n = 1$: the operation has not been scheduled in previous steps but it is going to be scheduled in the next one.
3) $p = 1$ and $n = 1$: the operation has been previously scheduled.

The automaton transition relation is thus[2]

$$\mathsf{TR}_i = [(p = 0) \Rightarrow (n = 0 \vee n = 1)] \wedge [(p = 1) \Rightarrow (n = 1)]$$

which reduces to

$$\mathsf{TR}_i = [(p = 1) \Rightarrow (n = 1)].$$

When it is necessary to work with operations requiring more than one time step, the previous representation must be extended to the automaton shown in Fig. 6(b), which models a task i with latency equal to l_i. The operation has not yet started in state 0 and it has been completed in state l_i.

The automaton may start its execution non-deterministically, but then it proceeds with a new state at every time step, until it reaches the final state.[3] With abuse of notation (as P and N denote sets of variables), we will represent the fact that such an automaton is in a present (next) state numbered j as $P_i = j$ ($N_i = j$).[4] Thus, in this case the automaton transition relation

[2]For the sake of readability, in this section we leave the support variables implicit. We thus simply write TR_i (TR_{xx}) instead of $\mathsf{TR}_i(P_i, N_i)$ ($\mathsf{TR}_{xx}(P, N)$, respectively).

[3]Note that pre-emptive scheduling can be achieved by letting every operation proceed to the next state, like in Fig. 6(b), or remain in the same state. This behavior can be obtained from the one analyzed in this paper with few minor modifications. We do not consider it in the sequel for the sake of simplicity.

[4]The Boolean meaning of such a notation actually depends on the strategy adopted to encode the problem. Previous experience [9] with this model has shown that the *thermometric encoding* is usually a good choice.

can be expressed as

$$\begin{aligned}\mathsf{TR}_i = {} & [(P_i = 0) \Rightarrow (N_i = 0 \vee N_i = 1)] \wedge \\ & \textstyle\bigwedge_{k=1}^{l_i-1} [(P_i = k) \Rightarrow (N_i = (k+1))] \wedge \\ & [(P_i = l_i) \Rightarrow (N_i = l_i)].\end{aligned} \tag{4}$$

It is worth noticing that a scheduling activity occurs *iff* the automaton next state differs from its current state. This condition can also be expressed as $(P_i \neq l_i) \wedge (N_i \neq 0)$.

Once the basic automata for all operations in the problem have been generated, they are combined together by means of a Cartesian product. The resulting automaton, further restricted by several constraints, incorporates all the allowed system behaviors, and can indeed be used for solving the scheduling problem. Following [8] and [9], the overall formulation is

$$\mathsf{TR} = \mathsf{TR}_{op} \wedge \mathsf{TR}_{dd} \wedge \mathsf{TR}_{rc} \tag{5}$$

where TR, i.e., the transition relation of the entire system, is computed as the conjunction of three terms.

1) TR_{op} describes the behavior of the basic scheduling automata. It is the Cartesian product (Boolean conjunction) of all the BSA relations given by (4)

$$\mathsf{TR}_{op} = \textstyle\bigwedge_{i \in BSA} \mathsf{TR}_i.$$

2) TR_{dd} represents the constraint introduced by data dependences (*dd*) or required conditions, i.e., it is illegal to start an operation with a predecessor that has not yet been completed. Hence, the expression $(P_i \neq l_i) \wedge (N_j \neq 0)$ is illegal for any $i \rightarrow j$ expressing a data dependence

$$\mathsf{TR}_{dd} = \textstyle\bigwedge_{(i \rightarrow j) \in dd} [(P_i = l_i) \vee (N_j = 0)].$$

3) TR_{rc} represents the resource constraints. Let b_r be the number of instances available of a given resource class r in the set of classes R, and ρ_r the set of operations competing for such a resource set. Then, it is illegal to schedule more than b_r concurrent operations from ρ_r. In other words, for any sub-set $\alpha_r \subseteq \rho_r$ such that $|\alpha_r| > b_r$, it is illegal to have all operations in α_r active at the same time

$$illegal_r = \textstyle\bigvee_{\alpha_r \subseteq \rho_r, |\alpha_r| > b_r} \bigwedge_{i \in \alpha_r} [(P_i \neq l_i) \wedge (N_i \neq 0)]$$

and so

$$\mathsf{TR}_{rc} = \textstyle\bigwedge_{r \in R} \neg illegal_r.$$

Notice that the expression of TR_{rc}, being basically an "at-most" constraint, can be actually encoded in CNF in several ways [47], [48].

Finally, the description of the composed automaton is completed by defining its initial state I, i.e., the state in which no operation has been scheduled, and final state T, i.e., the one in which all tasks have been scheduled. In practice, I and T are the Boolean conjunction of all BSA initial and final states.

Given this information, a valid schedule trace can be obtained by looking for paths connecting I and T through TR. This can be efficiently achieved with a BMC analysis. For instance, a shortest execution latency scheduling trace is given by the smallest value of k for which the Boolean formula $path^k$ of (3) is satisfiable.

IV. Min-Cost SAT-Based Scheduling

To extend the previous model to solve minimum-cost planning problems through min-cost SAT, we have to face two main problems.

1) While in Section III each operation is implicitly assumed to be performed only by a computational unit of a specific resource class, now tasks may be executed by several (different types of) resource units.
2) While in Section III the total cost is measured in terms of the length of the schedule trace, now each computational unit is characterized by idle and running cost rates.

These two problems will be respectively analyzed in the next two sub-sections.

A. Compound Scheduling Automata

We introduce the concept of compound scheduling automata to map each task over different types of resources. A compound scheduling automaton (CSA) represents an operation that can be performed by a number $n \geq 1$ of computational units belonging to different resource classes. It consists of exactly n independent BSA, which may run in parallel. Roughly speaking, each BSA within a compound represents the execution of the operation over *one* of the functional units that can be used, i.e., a possible binding. Thus, the transition relation of a CSA is the Cartesian product of the transition relations of all the BSA in the compound itself. Moreover, the initial and final states of a CSA can be derived from the physical meaning of the compound.

1) When the operation represented by the CSA has not yet started, *all* BSA within the compound must be in their initial states. In other words, the initial state of the CSA is given by the Boolean conjunction of the initial states of the included BSA.
2) When the task represented by the CSA has been completed, it has been executed by *at least one* of the functional units able to perform it. This means that the CSA final state is given by the Boolean disjunction of the final states of the included BSA.

Once the compound automata have been introduced in the model, extending the relations analyzed in Section III is straightforward. As far as TR_{op} is concerned, conceptually nothing changes. For TR_{dd} it is enough to replace the expressions identifying the BSA initial and final states with the corresponding ones for the involved CSA. The same is true for the initial and final states of the composed automaton. Finally, the correct expression of TR_{rc} can be obtained by working with all the *flattened* BSA present in the problem.

Example 4: Let us consider the task graph introduced within Example 3. For such a system, three CSA are defined, as represented in Fig. 7. Two of them actually coincide with the BSA for tasks t_1 and t_2 (namely t_1^1 and t_2^2). The last one contains two distinct BSA (t_3^1 and t_3^2), one for each binding of

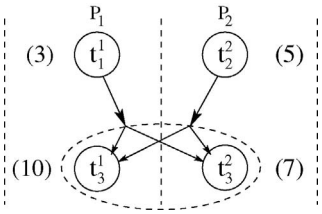

Fig. 7. CSA model for the task graph of Fig. 4.

t_3 over the two processors. For every BSA, a transition relation is built according to (4). For instance, for t_1^1 we have

$$\begin{aligned}\mathsf{TR}_{t_1^1} = {} & [(P_{t_1^1} = 0) \Rightarrow (N_{t_1^1} = 0 \vee N_{t_1^1} = 1)] \wedge \\ & [(P_{t_1^1} = 1) \Rightarrow (N_{t_1^1} = 2)] \wedge \\ & [(P_{t_1^1} = 2 \vee P_{t_1^1} = 3) \Rightarrow (N_{t_1^1} = 3)].\end{aligned}$$

Concerning the data dependences, we have that t_3 depends on both t_1 and t_2. The generated TR_{dd} term is hence

$$\begin{aligned}\mathsf{TR}_{dd} = {} & [P_{t_1^1} = 3 \vee (N_{t_3^1} = 0 \wedge N_{t_3^2} = 0)] \wedge \\ & [P_{t_2^2} = 5 \vee (N_{t_3^1} = 0 \wedge N_{t_3^2} = 0)].\end{aligned}$$

Then, there are two pairs of BSA in the model sharing the same resource, i.e., (t_1^1, t_3^1) and (t_2^2, t_3^2). The TR_{rc} component is hence[5]

$$\begin{aligned}\mathsf{TR}_{rc} = {} & [P_{t_1^1} = 3 \vee N_{t_1^1} = 0 \vee P_{t_3^1} = 10 \vee N_{t_3^1} = 0] \wedge \\ & [P_{t_2^2} = 5 \vee N_{t_2^2} = 0 \vee P_{t_3^2} = 7 \vee N_{t_3^2} = 0].\end{aligned}$$

Finally, the expressions for initial and final states of the composed automaton are

$$\begin{aligned}\mathsf{I} &= P_{t_1^1} = 0 \wedge P_{t_2^2} = 0 \wedge (P_{t_3^1} = 0 \wedge P_{t_3^2} = 0) \\ \mathsf{T} &= P_{t_1^1} = 3 \wedge P_{t_2^2} = 5 \wedge (P_{t_3^1} = 10 \vee P_{t_3^2} = 7).\end{aligned}$$

□

B. Modeling Execution and Idle Costs

To take into account the resource costs, we initially assume only one functional unit available for each resource class (this assumption will then be removed). First, all the BSA in the system (after flattening all CSA components) are associated with a new set M of *monitor* variables. Each Boolean variable $m_i \in M$ records the related automaton activity during the current step. More precisely, m_i is true *iff* the associated automaton is currently being scheduled

$$m_i \quad \Leftrightarrow \quad (P_i \neq l_i) \wedge (N_i \neq 0). \tag{6}$$

After that, we introduce another set E of *execution* variables, associated to the resources, in order to get an expression for the state (idle or busy) of the available computational units. In more detail, for each class $r \in R$, let ρ_r be the set of BSA competing for such a resource. Then, the related e_r variable is defined as a function of the sub-set of M corresponding to the BSA in ρ_r, by the following relation:

$$e_r \Leftrightarrow \bigvee_{i \in \rho_r} m_i. \tag{7}$$

It is easy to see that the meaning of the E variables is the following. $e_r = 1$ witnesses a busy state for the functional unit in class r, whereas $e_r = 0$ means that the unit is idle. The intuition behind the definitions of (6) and (7) is that each e_r variable provides an automated way to check whether a given functional unit is idle or busy. Therefore, we can now attach to every e_r variable the corresponding execution cost, and consequently compute the cost of a schedule trace following Behrmann *et al.* [20].

Unfortunately, this is not enough in order to obtain the correct optimal cost as a result of a min-cost SAT instance. This is because, in a min-cost SAT problem, each variable may have only one cost value, whereas we have two costs (busy and idle). For this reason, similarly to the E set, we finally introduce a new set I of *idle* variables, whose meaning is the reverse of the execution variables

$$i_r \Leftrightarrow \neg e_r$$

and properly attach an idle cost to each of them.

In summary, if we define the term TR_{exe} as

$$\begin{aligned}\mathsf{TR}_{exe} = {} & \textstyle\bigwedge_{i \in BSA} [m_i \Leftrightarrow (P_i \neq l_i) \wedge (N_i \neq 0)] \wedge \\ & \textstyle\bigwedge_{r \in R} [(e_r \Leftrightarrow \bigvee_{i \in \rho_r} m_i) \wedge (i_r \Leftrightarrow \neg e_r)]\end{aligned} \tag{8}$$

we can conjoin TR_{exe} with the system transition relation (5) without modifying the satisfiability of the BMC checks generated during the search for the optimal latency. The minimum-cost problem can be then obtained by defining the costs c_i for any CNF variable x_i in the following way.

1) The cost c_r^e of the running state for computational units in resource class r, if x_i corresponds to one of the unrolled CNF variables for e_r.
2) The cost c_r^i of the idle state for resources in class r, if x_i is one of the unrolled CNF variables for i_r.
3) Zero in all the other cases.

Example 5: With respect to the system presented in Example 3, the transition relation specified in Example 4 must be extended (conjoined) with the following expression:

$$\begin{aligned}\mathsf{TR}_{exe} = {} & [m_{t_1^1} \Leftrightarrow (P_{t_1^1} \neq 3 \wedge N_{t_1^1} \neq 0)] \wedge \\ & [m_{t_2^2} \Leftrightarrow (P_{t_2^2} \neq 5 \wedge N_{t_2^2} \neq 0)] \wedge \\ & [m_{t_3^1} \Leftrightarrow (P_{t_3^1} \neq 10 \wedge N_{t_3^1} \neq 0)] \wedge \\ & [m_{t_3^2} \Leftrightarrow (P_{t_3^2} \neq 7 \wedge N_{t_3^2} \neq 0)] \wedge \\ & [e_{p_1} \Leftrightarrow (m_{t_1^1} \vee m_{t_3^1})] \wedge [i_{p_1} \Leftrightarrow \neg e_{p_1}] \wedge \\ & [e_{p_2} \Leftrightarrow (m_{t_2^2} \vee m_{t_3^2})] \wedge [i_{p_2} \Leftrightarrow \neg e_{p_2}].\end{aligned}$$

The CNF costs are: 5 for variables derived for e_{p_1}, 4 for e_{p_2}, 2 for i_{p_1}, 1 for i_{p_2}, and 0 for all the others. □

The given formulation is fine when, for any resource class r, the number of available function units is $b_r = 1$. When $b_r > 1$, it fails because the term on the right side of (7) is true whenever *at least* one of the involved computational units is

[5]Our tool [8] is able to detect that t_1^1 and t_3^1 cannot execute at the same time because of the dependence constraint, and similarly for t_2^2 and t_3^2. In such a case, TR_{rc} would actually be computed as the constant one.

```
1   MINCOSTSATSCHEDULING (TG, R)
2       (TR, I, T) ← CREATEAUTOMATA (TG, R)
3       (lb, ub) ← ESTIMATEBOUNDS (TG, R)
4       k ← lb - 1
5       do
6           k ← k + 1
7           cex ← SAT (path^k)
8       while (cex = ∅)
9       cost ← ∞
10      while (k ≤ ub)
11          tmpCost ← MINCOSTSAT (path^k)
12          if (tmpCost < cost)
13              latency ← k
14              cost ← tmpCost
15          k ← k + 1
16      return (latency, cost)
```

Fig. 8. Top-level scheduling procedure based on min-cost SAT.

used, but we do not know the exact number. The problem can be overcome in two possible ways.

The most straightforward one consists in a pre-processing step. Each resource class r with $b_r > 1$ is broken into b_r distinct sub-classes, each of which with only one functional unit available. Then, operations initially mapped onto class r will have the choice for execution among the generated b_r sub-classes, thus reducing the problem to the previous formulation. This strategy, however, increases the complexity of the problem, as the total number of BSA in the global automaton becomes larger.

The second strategy consists in taking into account all possible legal resource states, distinguishing among the cases of 0, 1, ..., b_r functional units used for class r. This can be effectively performed through the use of the *concurrency cliques* presented in [8]. Hence, for any class r, exactly b_r execution (and idle) variables are introduced, each of which expresses a different resource exploitation.

C. Overall Algorithm

As mentioned in Section III, valid scheduling traces can be obtained by applying a BMC paradigm, i.e., by SAT solving the $path^k$ Boolean formula [see (3)] for various bounds (latencies) k. Similarly, a minimum-cost planning trace for a given bound can be found by exploiting a min-cost SAT solver over the same formula, with the only difference that the system TR must be extended as previously discussed. Unfortunately, in general, the optimal minimum-cost schedule trace does not correspond to the shortest execution latency. As a consequence, the cost-optimal scheduling problem requires the exploration of a range of possible bounds.

The high-level algorithm to solve this puzzle is presented in Fig. 8. Function MinCostSatScheduling receives the original task graph TG and the set of available resources R. It starts generating the automaton model of the DFG (line 2), following Sections III and IV. After that, it estimates the range of latencies that should be tried (i.e., the lower, lb, and upper, ub, bounds), in such a way that the latency corresponding to a minimum-cost trace is in the range [lb, ub]. This topic will be better analyzed in Section VI. After that, the procedure executes two consecutive cycles.

1) The purpose of the first one (lines 5–8) is to discover the shortest execution latency for the system, as the lower bound estimate returned by function EstimateBounds may be too coarse and deliver some unsatisfiable instances. Notice that we exploit standard SAT within this loop, since SAT solvers are much more efficient than min-cost SAT solvers. More in general, the loop may be actually replaced by any other algorithm or search strategy [49] delivering the minimum execution latency.
2) Once the first satisfiable instance has been detected, the second loop (lines 10–15) performs a min-cost SAT analysis on all the next instances, keeping track of the optimal cost and the corresponding latency.

V. ALL-SOLUTION SAT APPROACH

The algorithm of Fig. 8 is guaranteed to find an optimal cost scheduling trace. However, due to the limited efficiency of min-cost SAT solvers, it cannot be practically used even in medium-size cases, as we will show in Section VII. Thus, we propose in this section an improved all-solution SAT approach, which is able to overcome this problem.

In principle, a min-cost SAT routine can be seen as (and substituted by) an all-solution SAT search, where the following occurs for each counter-example.

1) Its corresponding cost is computed and possibly saved, if it represents the new minimum.
2) A blocking clause is added to the clause database, forbidding that solution to appear again [34]–[36].
3) The search is restarted in order to find a new solution.

Unfortunately, adopting this crude approach in Fig. 8 will not provide any advantage, since this algorithm (further enhanced with specific optimizations [30]) is already used by a class of min-cost SAT solvers. Nonetheless, we can have an edge over general min-cost solvers using some high level information gathered while generating a SAT instance. In particular, we can observe that all k-latency schedule traces involving the same resource profile are characterized by the same cost. In other words, if we want to find a minimum-cost schedule for a given bound k, we do not need to explore all traces (SAT solutions) feasible for every bound. On the contrary, we only have to analyze all "representative" solutions, i.e., the ones involving a different resource exploitation, and hence delivering a different cost.

In general, this cannot be trivially achieved. For every SAT solution, we should count the number of time steps each processor is running and somehow impose such a number to be different in all the subsequent solutions for at least one processor. However, we can easily obtain almost the same effect by working on the set of BSA that have been scheduled in the current solution. Indeed, at every new SAT counter-example, we can do the following.

1) Collect the set of BSA that have been scheduled. If more than one BSA related to the same CSA have been scheduled, we select the one which completed first.[6]

[6]Notice that selecting any other BSA in the involved CSA may not guarantee the validity of the schedule itself.

```
1   AllSatScheduling (TG, R)
2       (TR, I, T) ← createAutomata (TG, R)
3       (lb, ub) ← estimateBounds (TG, R)
4       k ← lb - 1
5       do
6           k ← k + 1
7           cex ← Sat (path^k)
8       while (cex = ∅)
9       cost ← ∞
10      valid ← TRUE
11      while (k ≤ ub)
12          do
13              cex ← Sat (path^k ∧ valid)
14              if (cex ≠ ∅)
15                  tmpCost ← computeCost (R, cex)
16                  if (tmpCost < cost)
17                      latency ← k
18                      cost ← tmpCost
19                  valid ← valid ∧ blockSolution (cex)
20          while (cex ≠ ∅)
21          k ← k + 1
22      return (latency, cost)
```

Fig. 9. Top-level scheduling procedure based on all-solution SAT.

2) Impose at least one of those BSA to be not scheduled in all the subsequent SAT solutions. Given the behavior of the BSA (see Section III), this constraint can be generated by forcing any of the involved BSA to be *not* in its final state *only in the last time step of the schedule*. This will intrinsically forbid all traces where the same sub-set of BSA has completed, even in different time steps from the current solution.

Furthermore, we can notice that all traces found at a given latency can also be "replicated" at larger depths, by a simple insertion of an "idling" time step at the end (or at the beginning, or even somewhere in the middle) of the schedule. All these traces are of no interest, because the related cost is certainly not the optimal one. Thus, such traces can be immediately ruled-out, and this can be achieved by means of the same set of constraints added for the smaller latencies. This means that all the blocking constraints can be accumulated and re-used for the next values of the bound.

This way to manage the SAT solutions, enlarging and re-using them, is probably the major contribution of this paper, as it represents a key factor for efficiency. The mechanism is conceptually similar to the one presented in [37], as each single SAT counter-example is used to derive (and prune out) a potentially large sub-set of the entire solution space. To this respect, though other techniques [34]–[36] have been presented to solve the all-solution SAT problem by making the blocking clauses stronger, in our case (as in [37]) the adopted constraints are obtained through a high-level reasoning on the problem. As such, they cannot be automatically derived by a general-purpose all-solution SAT engine.

The improved algorithm is reported in Fig. 9. The first part of the pseudo-code (lines 2−9) is identical to the one presented in Fig. 8. The second part implements the all-solution SAT algorithm just described and further detailed in the following example.

Example 6: Let us consider again the task graph presented in Example 3. Following the algorithm of Fig. 9, we have that

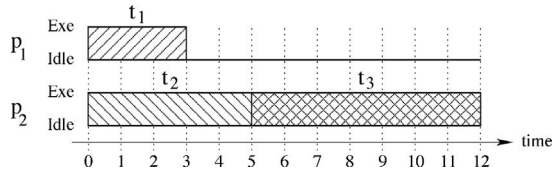

Fig. 10. Shortest schedule trace for the problem of Example 3.

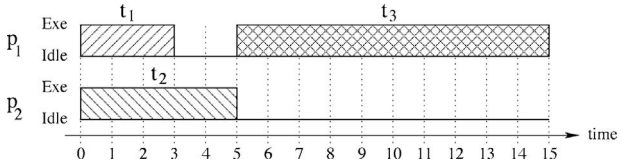

Fig. 11. Alternative schedule trace for the problem of Example 3.

the first satisfiable instance is found for bound 12. A possible schedule trace, obtained through the first SAT call for this latency, is represented in Fig. 10.

When this solution is returned by the SAT solver, the blocking expression built by function BlockSolution is

$$(P_{t_1^1} \neq 3) \vee (P_{t_2^2} \neq 5) \vee (P_{t_3^2} \neq 7)$$

meaning that we now look for a schedule where at least one of the three BSA involved in the previous trace does not complete. After this constraint is added at time frame 12, no more solutions can be found for bound 12. In fact, even though two other valid schedule traces exist (i.e., the ones in which task t_1 is shifted in time of 1 or 2 time-frames), both violate this constraint, and the SAT solver thus reports an UNSAT result. Similarly, for bound 13 (and 14) the number of valid schedule traces is equal to 11 (and 26), but all of them violate the previous condition when expressed at time frame 13 (and 14). Hence, they are immediately ruled-out and not reported by the SAT solver. In other words, for these bounds the nested loop of Fig. 9 consists in just one iteration, that simply detects the unsatisfiability of the instance.

Finally, coming at latency 15, the solver discovers one alternative solution, depicted in Fig. 11, as another 50 traces similar to the previous ones are skipped again. In this case, the expression built by function BlockSolution is

$$(P_{t_1^1} \neq 3) \vee (P_{t_2^2} \neq 5) \vee (P_{t_3^1} \neq 10).$$

Once this is also added to the clause database, all the subsequent CNF instances become UNSAT, no matter the bound we consider and the actual number of schedule traces that could be obtained. □

VI. Latency Lower and Upper Bound Estimations

A common operation to both the algorithms of Figs. 8 and 9 is the estimation of the latency range that must be explored. This is a crucial task, as estimates must be conservative (i.e., keep the optimal cost latency within the range), and tight (as loose estimates generate large numbers of iterations).

Lower and upper bounds on the value of a cost function are often identified by branch-and-bound algorithms [30], [39],

and the search is pruned when the lower bound estimation is higher than or equal to the upper bound.

As far as the lower bound is concerned, we compute it in two possible ways.

1) The sum over all tasks of the smallest time (l) required to execute the task itself, divided by the number of available processors ($|R|$)

$$lb_R = \left\lceil \sum_{i \in CSA} l_i / |R| \right\rceil . \quad (9)$$

2) The latency provided by an ASAP algorithm when the number of processors is assumed to be boundless

$$lb_\infty = ASAP(TG, \infty). \quad (10)$$

In general, none of the two estimates is superior to the other. The first one is more suitable when the problem can be decomposed into several independent connected components and the number of available resources is small. The second is better when dependence constraints dominate all the others. The final value of the lower bound is taken as the maximum between the previous estimates.

As far as the upper bound is concerned, a possible initial estimate is given by the sum over all tasks of the largest time (L) needed to execute the task itself

$$ub_L = \sum_{i \in CSA} L_i. \quad (11)$$

However, we may dynamically infer much closer estimates starting from cost considerations. Given a specific value k for the latency, we can under-estimate the cost for that latency in the following way. For each task, we select the resource able to execute it with the smallest cost. Let be l_r the total number of time steps each resource r is running according to the previous selection criterion. Then, an under-estimate c of the total cost for latency k is given by the sum over all resources of their execution cost c_r^e for l_r steps and their idle cost c_r^i for the remaining ones

$$c = \sum_{r \in R} [c_r^e \cdot l_r + c_r^i \cdot (k - l_r)].$$

Finally, if we have at hand an upper bound C for the total cost, we can exploit the previous expression in order to derive an over-estimate ub_C of the latency, by imposing $c < C$

$$\sum_{r \in R} [c_r^e \cdot l_r + c_r^i \cdot (ub_C - l_r)] \quad < \quad C$$

which leads to

$$ub_C \; < \; \left[C - \sum_{r \in R} (c_r^e - c_r^i) \cdot l_r \right] / \sum_{r \in R} c_r^i. \quad (12)$$

Equation (12) provides an important relation, as it allows us to dynamically update the latency upper bound ub as long as new values C for the minimum cost are found during the SAT search (this detail was omitted in the previous pseudo-codes for the sake of simplicity). The initial cost estimate C is found by running a heuristic ASAP algorithm complying with the resource constraints.

Example 7: Let us consider once more the task graph scheduling problem presented in Example 3. Following (9) and (10), our estimates for the lower bound are

$$\begin{array}{rcl} lb_R & = & \lceil (3+5+7)/2 \rceil \quad = \quad 8 \\ lb_\infty & = & ASAP(TG, \infty) \quad = \quad 12 \end{array}$$

and hence

$$lb = \max(lb_R, lb_\infty) \quad = \quad 12.$$

According to (11), we obtain the following estimate for the upper bound:

$$ub_L = 3 + 5 + 10 \quad = \quad 18.$$

However, by running the ASAP algorithm, we find the schedule trace depicted in Fig. 10, which is characterized by a cost equal to 81. Therefore, by applying (12)

$$ub_C < [81 - (5-2) \cdot 3 - (4-1) \cdot 12)]/(2+1) = 36/3 = 12$$

we obtain an upper bound ub smaller than the lower bound. This means that, at least in this trivial case, the solution returned by the ASAP scheduling algorithm corresponds to the optimal one. Thus, the process terminates without even resorting to SAT. □

VII. Experimental Results

We implemented the novel techniques presented in this paper in our SAT-based scheduling tool [8], extending it from a minimum latency search to a cost-optimal analysis. We adopted the Minisat [50] tool (version p1.14) as underlying SAT solver and MinCostChaff [30] as min-cost SAT engine. Furthermore, as all min-cost SAT instances can be represented as binate covering problems, and in turn as pseudo-Boolean optimization problems, for performance evaluation we investigated both Minisat+ [50], [51] (version 2007 January 05) and SCIP [52], [53] (version 1.2.0). Minisat+ is a SAT-based solver including all main generalizations of SAT to PB, such as multiple ways of translating pseudo-Boolean constraints to clauses. On the other hand, SCIP combines several solving strategies, such as branching, cutting, pricing, and propagation. Moreover, it won five categories of the 2009 pseudo-Boolean competition [54] and ranked second in two more categories.

All our experiments run on a 3.0 GHz Dual Core workstation, equipped with 3 GB of main memory and running Debian Linux. Time and memory limits were always set to 3600 s and 1 GB, respectively. We compare the min-cost SAT (possibly in its pseudo-Boolean version) strategy of Section IV with the improved all-solution SAT technique described in Section V. Furthermore, the state-of-the-art priced timed automata tool UPPAAL CORA [20], [55] (version 4.0.6) is used as an alternative approach to solve the same problem. Notice that UPPAAL CORA is able to deliver both optimal and pseudo-optimal results. We compare our methods against its most efficient exact strategy.

TABLE I
SYNTHETIC BENCHMARKS CHARACTERISTICS

Name	Tasks		#P	Latency			Cost
	#CSA	#BSA		Min	Opt	Max	
f01	5	8	3	19	20	22	548
f02	10	17	3	32	34	35	1273
f03	15	29	3	45	47	52	1608
f04	20	33	4	67	70	71	2435
f05	30	42	4	98	101	103	3776
f06	40	60	4	120	122	124	4700
f07	50	72	5	141	144	146	6384
r01	5	12	3	10	11	12	455
r02	10	23	3	24	26	29	1624
r03	15	30	4	46	48	49	2210
r04	20	39	4	59	65	65	3060
r05	30	46	4	83	87	92	4495
r06	40	71	4	118	119	125	6041

We present results on both automatically generated problems, and a large set of instances derived from standard benchmarks [56].

As far as the synthetic problems are concerned, all task graphs consist in a single connected component. However, this is not a real limitation, as several connected components can be managed in the same way. The set includes two families of designs, named f*i* and r*i*, which differ in their graph shape. More specifically, every f*i* instance has a large number of operations without predecessors or successors, i.e., inputs and outputs of the task graph, whereas the number of tasks at intermediate levels is quite small. Conversely, each r*i* problem has a small number of inputs and outputs and a large number of tasks at intermediate levels. Furthermore, the tasks in the r family may execute, on average, over a larger number of resources. Table I shows the main characteristics of such benchmarks. For each model, it reports the number of tasks (column #CSA) the total number of BSA (#BSA), the number of the available processors (#P), the shortest (Min), optimal (Opt), and maximum (Max) execution latency, and the total cost for the optimal solution (Cost). Notice that the value Max−Min+1 represents the number of min-cost SAT (or pseudo-Boolean) calls performed by our algorithms.

Table VII shows the results we collected with all the discussed solving strategies. MCS, PBO, UPC and IAS denote the results for the min-cost SAT model, its pseudo-Boolean optimization counter-part (Minisat+ and SCIP), UPPAAL CORA, and the improved all-SAT strategy, respectively. For all those strategies, the table reports the total CPU time and the peak memory required to find the optimal solution of each problem. When a time/memory overflow occurred, we report (between parentheses) the memory/time used up to that moment. From our data, we can draw the following conclusions. The min-cost SAT approach is poorly scalable, as it could solve only the simplest problems. Although some of its inefficiencies might be overcome (e.g., we did not implement an incremental SAT [57] approach, because the source code of the solver is not available), this strategy does not seem powerful enough to solve complex instances. Almost the same considerations hold for the cases where a pseudo-Boolean solver is exploited (with SCIP with a discrete edge over Minisat+), as both the engines could solve only the smallest instances. Interestingly, we can notice that, although designed to solve a more general problem, SCIP behaves slightly better even than MinCostChaff. We argue this could be due to some recent improvements in pseudo-Boolean solving [58], [59] that have been not incorporated into the min-cost SAT solver. Anyhow, even with these advances, the PBO version of the min-cost SAT strategy is also not competitive. The priced timed automata model, as implemented in UPPAAL CORA, currently appears to be a superior approach. However, among all the engines, our improved all-solution SAT technique is by far the most efficient and scalable method, as it is the only able to solve the largest instances. Moreover, while the limiting factor for the other strategies is often the amount of available memory, this is absolutely not a problem for the all-SAT strategy.

As far as standard instances are concerned, we adapted[7] to our needs the benchmarks taken from [56]. This web-site hosts a huge set of instances, designed to evaluate multiprocessor scheduling algorithms targeting minimum latency, when a set of t tasks (having arbitrary precedence constraints and arbitrary processing times) are assigned to p processors of the same capability. The graph shapes have been generated according to four different methods, fully described in [56]. The number of tasks t varies between 50 and 2700, whereas the number of processors p is arbitrarily selected.

For our purposes, since even the instances with 50 tasks were very hard to solve when expressed as task graph cost-optimal scheduling problems,[8] we extracted our benchmarks by cutting the instances in this suite to the first n tasks, with n equal to 10, 20, 30, and 40. After that, for each value of n, we generated a problem instance with a number of processors p equal to 2, 3, and 4. While doing that, we enriched the description with (randomly generated) information about the costs of the available processors and the association between every task and the given set of resources. To be more specific, for the (idle and running) costs, we generated integer values ranging from 2 to 15 (with the idle cost always smaller than the running one). Moreover, each task was (randomly) associated to a number of processors varying from 1 to p. Depending on the mean and variance values of these parameters, we were able to generate instances with distinct latency and costs, and implying a different effort to be solved. To sum up, the process outlined above allowed us to obtain a total of 2160 test cases.

For these benchmarks we limited our analysis to the two most efficient techniques outlined by Table VII, i.e., the UPPAAL CORA priced timed automata model (UPC) and the improved all-SAT method (IAS).

The results are summarized by the scattered plots of Fig. 12, which contrast the CPU time and memory requirements for UPPAAL CORA and our tool. For a fair comparison, the plot of Fig. 12(a) [Fig. 12(b)] does not report those cases in which a memory (time) overflow has been hit by any of the tools. For this reason, only 1288 (2032) points appear in Fig. 12(a) [Fig. 12(b)].

As far as the time comparison is concerned, our tool shows a fair edge on UPPAAL CORA in the vast majority of the

[7]Just like the authors of [24], we were not able to find a set of publicly available benchmarks suited for the problem we are targeting.

[8]This fact is not really surprising, as the task graph cost-optimal scheduling problem is far more complex than the original simple minimum latency puzzle.

TABLE II
RESULTS FOR THE TWO SYNTHETIC BENCHMARK FAMILIES

Name	MCS		PBO				UPC		IAS	
			Minisat+		SCIP					
	Time [s]	Mem [MB]	Time [s]	Mem [MB]	Time [s]	Mem [MB]	Time [s]	Mem [MB]	Time [s]	Mem [MB]
f01	9	19	159	25	2	21	0	0	0	0
f02	124	176	*ovf*	(80)	80	84	1	22	1	12
f03	*ovf*	(832)	*ovf*	(144)	*ovf*	(170)	243	162	5	17
f04	(2934)	*ovf*	*ovf*	(367)	*ovf*	(256)	466	311	8	20
f05	(1878)	*ovf*	*ovf*	(511)	*ovf*	(483)	(2242)	*ovf*	27	28
f06	(1743)	*ovf*	*ovf*	(783)	*ovf*	(639)	(1628)	*ovf*	240	40
f07	(1190)	*ovf*	*ovf*	(924)	*ovf*	(772)	(1521)	*ovf*	2829	54
r01	14	23	134	31	15	31	0	0	0	0
r02	*ovf*	(257)	*ovf*	(105)	3332	121	44	59	1	13
r03	(1462)	*ovf*	*ovf*	(198)	*ovf*	(176)	216	183	2	17
r04	(1151)	*ovf*	*ovf*	(412)	*ovf*	(220)	(2090)	*ovf*	48	19
r05	(1235)	*ovf*	*ovf*	(581)	*ovf*	(303)	(1874)	*ovf*	1163	27
r06	(986)	*ovf*	*ovf*	(853)	*ovf*	(640)	(1979)	*ovf*	3351	55

ovf indicates a time (3600 s) or memory (1 GB) overflow. MCS, PBO, UPC, and IAS denote min-cost SAT, pseudo-Boolean optimization, UPPAAL CORA, and the improved all-SAT strategy, respectively.

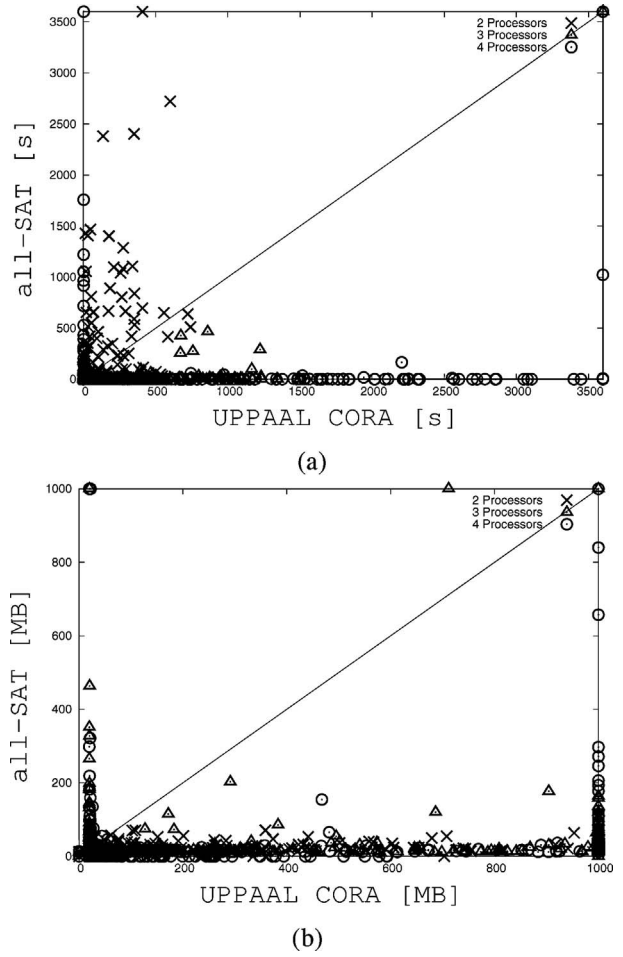

Fig. 12. (a) CPU time and (b) memory usage for standard benchmarks. All-SAT vs. UPPAAL comparison.

cases (more than 1000 points are located below the diagonal). We can also observe that UPPAAL CORA is faster than our approach in a non-negligible number of benchmarks (about 260), though almost all of them could be solved as well with our method within half of the time limit. For these test cases, we observed that a considerable amount of time has been spent by our tool to prove the unsatisfiability of some CNF instances while searching for the minimum execution latency (first loop of Fig. 9). To this respect, the exploitation of an up-to-date SAT solver may further improve the performance of our strategy. Anyway, the presence of such cases witness that the two approaches we are comparing are orthogonal and somehow complementary.

The comparison on memory is even more clear. With the exception of some benchmarks, that UPPAAL CORA solved very quickly, the memory requirements of our tool are consistently smaller than the priced timed automata engine. In most of the cases, a few MB are enough to complete the experiments with our method, whereas several hundred MB are usually required by UPPAAL CORA.[9]

The previous results can be further appreciated when considering the overall statistics. Our tool was able to solve a total of 2026 benchmarks, i.e., more than 90% of all instances, while hitting 113 overflows on time and 21 overflows on memory. On the other hand, UPPAAL CORA could solve only 1291 test cases (less than 60%), hitting 21 and 859 overflows on time and memory, respectively. The total run-time (including all overflows) has been almost 252 h for our tool, as opposed to 350 h taken by UPPAAL CORA. This means that our tool solved about 50% more benchmarks than UPPAAL CORA, while requiring slightly more than 70% of its running time. Clearly, these data confirm that our approach is more robust and scalable. On the other hand, the cases in which UPPAAL CORA outperformed our tool are apparently due to the cost-optimal branch-based reachability analysis implemented in that tool. Given the nature of the two compared methods, it seems reasonable that UPPAAL CORA shows some advantage on specific instances.

Finally, it is important to remark the impact of the latency estimates (see Section VI) in our algorithm. As it can be noticed from Table I, the maximum latency values that had to be explored are very close to the optimal ones, meaning that

[9]To this respect, it should be further noticed that we always ran UPPAAL CORA using the on-line interface, thus avoiding the graphical-user interface. This method is obviously faster and less memory consuming.

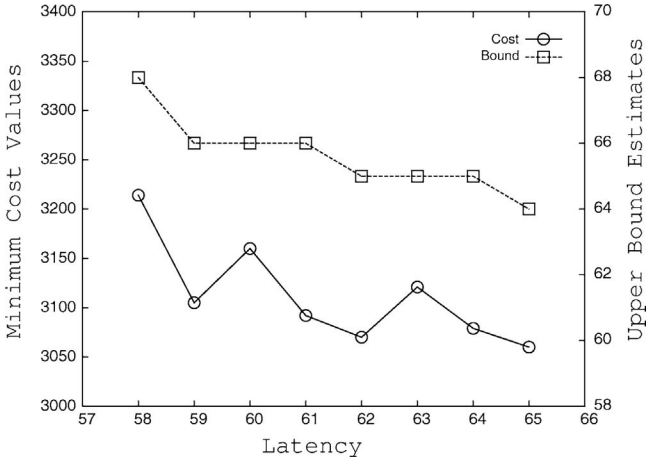

Fig. 13. Cost and upper bound estimate as a function of the latency.

our estimates are precise and avoid useless BMC iterations. Fig. 13 further highlights the role played by (12) in our framework. It plots the minimum cost and the upper bound estimation as a function of the latency for the r04 benchmark of Tables I and VII. The initial cost provided by the heuristic ASAP algorithm (equal to 3214) corresponds to a latency estimate equal to 68 (these two values have been reported in the graph for a latency equal to 58). As shown in Table I, the first satisfiable instance is obtained at latency 59. For this value, our all-SAT routine provides a minimum cost equal to 3105 and consequently an upper bound estimation equal to 66. We proceed in a similar way until the latency becomes equal to 65. For this bound, we found a new minimum cost (equal to 3060) and the estimate of the latency is updated to 64. This confirms that the optimal solution has been found and terminates the execution.

VIII. Conclusion

This paper described the application of min-cost SAT and all-solution SAT in order to solve the task graph cost-optimal scheduling problem. To sum up, our work included the following contributions: 1) a new SAT-based model of the problem; 2) a complete strategy to solve those models, based on minimum-cost satisfiability or pseudo-Boolean optimization and bounded model checking; 3) an improved all-solution SAT-based strategy to capture a large set of solutions for any new SAT counter-example, thus drastically reducing the number of iterations that must be performed by the algorithm; and (4) a branch-and-bound heuristic to dynamically evaluate and use new latency bounds as long as the solving process proceeds.

As far as we know, the model and the related example-specific optimizations have never been proposed before. Experimental results, performed on both synthetic and standard problems, show that the proposed approach is much more efficient than existing techniques.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. Hightstown, NJ: McGraw-Hill, 1994.

[2] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 1, pp. 85–93, Jan. 1991.

[3] C.-T. Hwang and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 4, pp. 464–475, Apr. 1991.

[4] I. Radivojevic and F. Brewer, "A new symbolic technique for control-dependent scheduling," *IEEE Trans. Comput.-Aided Design*, vol. 15, no. 1, pp. 45–47, Jan. 1996.

[5] S. Haynal and F. Brewer, "Automata-based scheduling for looping DFGs," Univ. California, Santa Barbara, Tech. Rep. EC99_14, Oct. 1999.

[6] G. Cabodi, M. Lazarescu, L. Lavagno, S. Nocco, C. Passerone, and S. Quer, "A symbolic approach for the combined solution of scheduling and allocation," in *Proc. 15th ACM/IEEE ISSS*, 2002, pp. 237–242.

[7] S. O. Memik and F. Fallah, "Accelerated SAT-based scheduling of control/data flow graphs," in *Proc. Int. Conf. Comput. Design*, 2002, pp. 395–400.

[8] G. Cabodi, A. Kondratyev, L. Lavagno, S. Nocco, S. Quer, and Y. Watanabe, "A BMC-based formulation for the scheduling problem of hardware systems," *Int. J. STTT*, vol. 7, no. 2, pp. 102–117, Jan. 2005.

[9] V. Bruno, L. Garcia, S. Nocco, and S. Quer, "Stressing symbolic scheduling techniques within aircraft maintenance optimization," *Int. JSAT*, vol. 5, nos. 1–4, pp. 77–104, Jun. 2008.

[10] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 4, pp. 464–475, Apr. 1991.

[11] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Design*, vol. 8, no. 6, pp. 661–679, Jun. 1989.

[12] O. Dain, D. Etherington, M. Ginsberg, O. Keenan, and T. Smith, "Automatic scheduling to minimize shipbuilding cost," in *Proc. 12th Int. Conf. Comput. Applicat. Shipbuilding*, 2005, pp. 41–54.

[13] S. Haynal and F. Brewer, "Efficient encoding for exact symbolic automata-based scheduling," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1998, pp. 477–481.

[14] G. Cabodi, A. Kondratiev, L. Lavagno, S. Nocco, S. Quer, and Y. Watanabe, "A BMC-formulation for the scheduling problem in highly constrained hardware systems," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 623–638, Jul. 2003.

[15] R. Alur and D. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[16] A. Fehnker, "Scheduling a steel plant with timed automata," in *Proc. 6th Int. Conf. RTCSA*, 1999, pp. 280–286.

[17] T. Hune, K. G. Larsen, and P. Pettersson, "Guided synthesis of control programs using UPPAAL," *Nordic J. Comput.*, vol. 8, no. 1, pp. 43–64, 2001.

[18] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Petterson, J. Romijn, and F. Vaandrager, "Minimum-cost reachability for priced timed automata," in *Proc. 4th Int. Workshop Hybrid Syst.: Computat. Contr.*, LNCS 2034. 2001, pp. 147–161.

[19] R. Alur, S. L. Torre, and G. Pappas, "Optimal paths in weighted timed automata," in *Proc. 4th Int. Workshop Hybrid Syst.: Comput. Control*, LNCS 2034. 2001, pp. 49–62.

[20] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," in *SIGMETRICS Performance Eval. Rev.*, vol. 32, no. 4, pp. 34–40, 2005.

[21] F. Gruian and K. Kuchcinski, "Low-energy directed architecture selection and task scheduling for system-level design," in *Proc. EUROMICRO Conf.*, vol. 1. 1999, p. 1296.

[22] D. Abramson, J. E. Beasley, M. Krishnamoorthy, and Y. M. Sharaiha, "Scheduling aircraft landings: The static case," *Transport. Sci.*, vol. 34, no. 2, pp. 180–197, 2000.

[23] G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn, "As cheap as possible: Efficient cost-optimal reachability for priced timed automata," in *Proc. Comput.-Aided Verification*, 2001, pp. 493–505.

[24] K. G. Larsen, J. I. Rasmussen, and K. Subramani, "Resource-optimal scheduling using priced timed automata," in *Tools and Algorithms for the Construction and Analysis of Systems* (LNCS Series, vol. 2988). Berlin, Germany: Springer, 2004, pp. 220–235.

[25] K. G. Larsen, J. I. Rasmussen, and K. Subramani, "On using priced timed automata to achieve optimal scheduling," *Formal Methods Syst. Design*, vol. 29, no. 1, pp. 97–114, Jul. 2006.

[26] S. W. Jeong and F. Somenzi, "A new algorithm for the binate covering problem and its application to the minimization of a

Boolean relations," in *Proc. Int. Conf. Comput.-Aided Design*, 1992, pp. 417–420.
[27] O. Coudert, "On solving covering problems," in *Proc. 33rd Annu. Design Automat. Conf.*, 1996, pp. 197–202.
[28] V. M. Manquinho and J. P. Marques-Silva, "Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem," *IEEE Trans. Comput.-Aided Design*, vol. 21, no. 5, pp. 505–516, May 2002.
[29] X. Y. Li, "Optimization algorithms for the minimum-cost satisfiability problem," Ph.D. dissertation, Dept. Comput. Sci., North Caroline State Univ., Raleigh, 2004.
[30] Z. Fu and S. Malik, "Solving the minimum cost satisfiability problem using SAT based branch and bound search," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2006, pp. 852–859.
[31] W. E. Adams, A. Billionnet, and A. Sutter, "Unconstrained 0-1 optimization and Lagrangian relaxation," *Discrete Appl. Math.*, vol. 29, nos. 2–3, pp. 131–142, 1990.
[32] D. Chai and A. Kuehlmann, "A fast pseudo-Boolean constraint solver," *IEEE Trans. Comput.-Aided Design*, vol. 24, no. 3, pp. 305–317, Mar. 2005.
[33] O. Roussel and V. M. Manquinho, "Pseudo-Boolean and cardinality constraints," in *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009, pp. 695–733.
[34] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Proc. Comput.-Aided Verific.*, LNCS 2404. 2002, pp. 250–264.
[35] H. S. Jin and F. Somenzi, "Prime clauses for fast enumeration of satisfying assignments to boolean circuits," in *Proc. Design Automat. Conf.*, 2005, pp. 750–753.
[36] H. S. Jin, H. J. Han, and F. Somenzi, "Efficient conflict analysis for finding all satisfying assignments of a boolean circuit," in *Tools and Algorithms for the Construction and Analysis of Systems* (LNCS, vol. 3440). Berlin, Germany: Springer, 2005, pp. 287–300.
[37] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based unbounded symbolic model checking using circuit cofactoring," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2004, pp. 510–517.
[38] P. Barth, "A Davis–Putnam enumeration algorithm for linear pseudo-Boolean optimization," Max Plank Inst. Comput. Sci., Tech. Rep. MPI-I-95-2-003, 1995.
[39] V. M. Manquinho and J. P. Marques-Silva, "Effective lower bounding techniques for pseudo-Boolean optimization," in *Proc. Design Automat. Test Europe Conf.*, Mar. 2005, pp. 660–665.
[40] C. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," in *Handbook of Knowledge Representation*. San Diego, CA: Elsevier, 2008, pp. 89–134.
[41] H. Kautz and B. Selman, "Planning as satisfiability," in *Proc. Eur. Conf. Artif. Intell.*, 1992, pp. 359–363.
[42] H. Kautz and B. Selman, "Pushing the envelope: Planning, propositional logic, and stochastic search," in *Proc. 13th Nat. Conf. AAAI*, Aug. 1996, pp. 1194–1201.
[43] J. Rintanen and J. Hoffmann, "An overview of recent algorithms for AI planning," *Künstliche Intell.*, vol. 2, no. 1, pp. 5–11, 2001.
[44] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. 36th Design Automat. Conf.*, Jun. 1999, pp. 317–320.
[45] H. El-Rewini, H. H. Ali, and T. G. Lewis, "Task scheduling in multiprocessor systems," *IEEE Trans. Comput.*, vol. 28, no. 12, pp. 27–37, Dec. 1995.
[46] Y. Abdeddaïm, A. Kerbaa, and O. Maler, "Task graph scheduling using timed automata," in *Proc. 17th IPDPS*, 2003, p. 237.2.
[47] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of Boolean cardinality constraints," in *Proc. Int. Conf. Principles Pract. Constr. Programm.*, LNCS 2833. 2003, pp. 108–122.
[48] C. Sinz, "Toward an optimal CNF encoding of boolean cardinality constraints," in *Proc. Int. Conf. Principles Pract. Constr. Programm.*, Oct. 2005, pp. 827–831.
[49] M. J. Streeter and S. F. Smith, "Using decision procedures efficiently for optimization," in *Proc. Int. Conf. Automat. Plan. Schedul.*, 2007, pp. 312–319.
[50] N. Eén and N. Sörensson. (2009, Apr.). *The Minisat SAT Solver* [Online]. Available: http://minisat.se
[51] N. Eén and N. Sörensson, "Translating pseudo-Boolean constraint into SAT," *Int. JSAT*, vol. 2, nos. 1–4, pp. 1–26, 2006.
[52] *The SCIP (Solving Constraint Integer Programs) Solver* [Online]. Available: http://scip.zib.de
[53] T. Achterberg, "SCIP: Solving constraint integer programs," *Math. Programm. Computat.*, vol. 1, no. 1, pp. 1–41, Jul. 2009.
[54] V. Manquinho and O. Roussel, "Results of the fourth pseudo Boolean competition," in *Theory and Applications of Satisfiability Testing*, 2009 [Online]. Available: http://www.cril.univ-artois.fr/PB09
[55] G. Behrmann, A. David, and K. G. Larsen. (2006). *The Uppaal Tool* [Online]. Available: http://www.uppaal.com
[56] Kasahara Laboratory, Waseda University. *Standard Task Graph Set* [Online]. Available: http://www.kasahara.elec.waseda.ac.jp/schedule
[57] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," in *Proc. 1st Int. Workshop BMC*, Jul. 2003, pp. 543–560.
[58] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean SAT solver," *Int. JSAT*, vol. 2, nos. 1–4, pp. 165–189, 2006.
[59] V. Manquinho and J. P. Marques-Silva, "On using cutting planes in pseudo-Boolean optimization," *Int. JSAT*, vol. 2, nos. 1–4, pp. 209–219, 2006.

Sergio Nocco received the M.S. degree in electrical engineering and computer science in 2001 and the Ph.D. degree in information and system engineering in 2005, both from Politecnico di Torino, Turin, Italy.

During his studies, he performed several summer internships with companies working in the electronics area. He was with NEC, Tokyo, Japan, in 2001, Cadence Berkeley Laboratories, Berkeley, CA, in 2002, and Intel, Santa Clara, CA, in 2003, 2004, and 2005. Currently, he is a Post-Doctoral Fellow with the Formal Methods Group, Dipartimento di Automatica e Informatica, Politecnico di Torino. His current research interests include symbolic techniques, based on both binary decision diagrams and satisfiability solvers, and their application to hardware and software formal verification, high level synthesis, and game theory.

Stefano Quer received the M.S. degree in electrical engineering and computer science from Politecnico di Torino, Turin, Italy, in 1991, and the Ph.D. degree in information and system engineering in 1996.

In 1994, he was with the Department of Electronic Engineering and Computer Science, University of California, Berkeley. In 1998, he collaborated with the Advanced Technology Group, Synopsys, Inc., Mountain View, CA, and in 1999 with the Alpha Development Group, Compaq, Shrewsbury, MA. He has been a Consultant for Compaq Computer Corporation, and an Assistant Professor with the Dipartimento di Automatica e Informatica, Politecnico di Torino. He is currently an Associate Professor with the Dipartimento di Automatica e Informatica, Politecnico di Torino. His current research interests include hardware description languages, logic synthesis, formal verification, simulation, and testing of digital circuits and systems.