Efficient SAT Encodings for Hierarchical Planning

Dominik Schreiber[1,2], Damien Pellier[1], Humbert Fiorino[1] and Tomáš Balyo[2]

[1]*Karlsruhe Institut für Technologie, Karlsruhe, Germany*

[2]*Université Grenoble-Alpes, Grenoble, France*
{*damien.pellier, humbert.fiorino*}*@imag.fr*, {*dominik.schreiber, tomas.balyo*}*@kit.edu*

Abstract: Hierarchical Task Networks (HTN) are one of the most expressive representations for automated planning problems. On the other hand, in recent years, the performance of SAT solvers has been drastically improved. To take advantage of these advances, we investigate how to encode HTN problems as SAT problems. In this paper, we propose two new encodings: GCT (Grammar-Constrained Tasks) and SMS (Stack Machine Simulation), which, contrary to previous encodings, address recursive task relationships in HTN problems. We evaluate both encodings on benchmark domains from the International Planning Competition (IPC), setting a new baseline in SAT planning on modern HTN domains.

1 INTRODUCTION

HTN (*Hierarchical Task Network*) planning (Georgievski and Aiello, 2015) is one of the most efficient and widely used planning techniques in practice. It is based on expressive languages allowing to specify complex expert knowledge for real world domains. Unlike classical planning (Fikes and Nilsson, 1971), in HTN planning, the goal is expressed as a set of tasks to achieve, and to which it is possible to associate different kinds of constraints: the set of constraints and tasks is a *Task Network*.

Boolean satisfiability (SAT) solving is a generic problem resolution method which has already been successfully applied to classical planning before, e.g., (Rintanen, 2012). Given a propositional logic formula F, the objective of SAT solving is to find an assignment to all occurring Boolean variables such that F evaluates to *true*, or to report unsatisfiability if no such assignment exists. When applying SAT solving to planning problems, the full procedure features four steps: (1) enumerating and instantiating all the possible actions, (2) encoding the instantiated planning problem into propositional logic, (3) finding a solution with a SAT solver, e.g., (Eén and Sörensson, 2003a; Audemard and Simon, 2009; Biere, 2013), and (4) decoding the found variable assignment back to a valid plan. In conventional SAT planning (Kautz and Selman, 1992; Kautz et al., 1996), each fact and each action at each step is commonly represented by a Boolean variable. The logic of how actions are ex-

ecuted and how they transform the world state is encoded with general clauses, while the initial state and the goal are provided as single literals (unit clauses). As the number of necessary actions is generally unknown in advance, the planning problem is iteratively re-encoded for a growing amount of steps, until a solution is found or the computation timed out. The SAT approach to planning is appealing because all planning is performed by SAT solvers using very efficient domain-independent heuristics; thus, any advances towards more efficient SAT solvers can also improve SAT-based planning.

The technique of *incremental SAT solving* has recently gained popularity for planning purposes (Gocht and Balyo, 2017). While conventional SAT solving processes a single formula in an isolated manner, incremental SAT solving allows for multiple solving steps while successively modifying the formula (Eén and Sörensson, 2003b): New clauses can be added between steps, and single literals can be *assumed*, i.e. they are enforced for a single solving attempt and dropped afterwards. A central advantage of incremental SAT solving is that SAT solvers can learn conflicts in assignments from past solving attempts and thus find a solution more efficiently (Nabeshima et al., 2006).

Compared to classical planning, significantly less research has been done on using SAT solvers for HTN planning (Mali and Kambhampati, 1998). One of the reasons is that enumerating and instantiating methods in HTN planning is challenging. Recently, efficient

instantiation procedures in HTN planning have been proposed (Ramoul et al., 2017), which allows us to investigate effective SAT-based approaches.

In this paper, our contributions are two new SAT encodings for HTN problems: GCT (Grammar-Constrained Tasks) and SMS (Stack Machine Simulation). Contrary to previous encodings (Mali and Kambhampati, 1998), GCT and SMS fully address recursive task relationships in HTN problems, and SMS is specifically designed for incremental SAT solving.

2 HTN PLANNING

In this section we introduce the foundations of HTN planning. A *fact* is an atomic logical proposition. A *state* s is a consistent set of positive facts. An *operator* o is a tuple $o = (name(o), pre(o), eff(o))$ where: $name(o)$ is a syntactic expression of the form $n(x_1, \ldots, x_n)$ with n being the name of the operator and $x_1, \ldots, x_n$ its parameters; and $pre(o)$ and $eff(o)$ are two sets of facts which define respectively the preconditions that must hold to apply the operator and the effects that must hold after the application of o.

An *action* is a ground operator, i.e. it has no free parameters. Action a is *applicable* to a state s if $pre(a) \subseteq s$. The resulting state s' of the application of a in a state s is defined as follows: $s' = \gamma(s, a) = (s \setminus eff^-(a)) \cup eff^+(a)$, where eff^+ (eff^-) represent the positive (negative) facts in eff. The application of a sequence of actions is recursively defined by $\gamma(s, \langle \rangle) = s$ and $\gamma(s, \langle a_0, a_1, \ldots, a_n \rangle) = \gamma(\gamma(s, a_0), \langle a_1, \ldots, a_n \rangle)$.

A *method* $m = (name(m), pre(m), subtasks(o))$ is a tuple where: $name(m)$ is a syntactic expression of the form $n(x_1, \ldots, x_n)$ with n being the name of the method and $x_1, \ldots, x_n$ its parameters; $pre(m)$ defines the preconditions, i.e., facts that must hold to apply m, and $subtasks(m)$ is the sequence of subtasks that must be executed in order to apply m. A *reduction* is a ground method. A reduction r is applicable in a state s if $pre(r) \subseteq s$.

A *task* t is a syntactic expression of the form $t(x_1, \ldots, x_n)$ where t is the task symbol and $x_1, \ldots, x_n$ its parameters. A tasks is *primitive* if t is the name of an operator; otherwise the task is *non-primitive*.

An action a accomplishes a primitive task t in a state s if $name(a) = t$ and a is applicable in s. Similarly, a reduction r accomplishes a non-primitive task t in a state s if $name(r) = t$ and r is applicable in s. Moreover, tasks can have different reductions.

An *HTN planning problem* is a 5-tuple $P = (s_0, g, T, O, M)$ where s_0 and g are respectively the initial state and the goal defined by logical propositions, T is an ordered list of tasks $\langle t_0, \ldots, t_{k-1} \rangle$, O is a set of operators, and M is a set of methods defining the possible reductions of a task.

A *solution plan* for a planning problem $P = (s_0, g, T, O, M)$ is a sequence of actions $\pi = \langle a_0, \ldots, a_n \rangle$ such that $g \subseteq \gamma(s_0, \pi)$. Intuitively, it means that there is a reduction of T into π such that π is executable from s_0 and each reduction is applicable in the appropriate state of the world. The recursive formal definition has three cases. Let $P = (s_0, g, T, O, M)$ be an HTN planning problem. A *plan* $\pi = \langle a_0, \ldots, a_n \rangle$ is a *solution* for P iff:

Case 1. T is an empty sequence of tasks. Then the empty plan $\pi = \langle \rangle$ is the solution for P if $g \subseteq s_0$.

Case 2. The first task t_0 of T is primitive. Then π is a solution for P if there is an action a_0 obtained by grounding an operator $o \in O$ such that (1) a_0 accomplishes t_0, (2) a_0 is applicable in s_0 and (3) $\pi = \langle a_1, \ldots, a_n \rangle$ is a solution plan for the HTN planning problem:

$$P' = (\gamma(s_0, a_0), g, \langle t_1, \ldots, t_{k-1} \rangle, O, M)$$

Case 3. The first task t_0 of T is non-primitive. Then π is solution if there is a reduction $d = subtasks(t_0)$ obtained by grounding a method $m \in M$ such that d accomplishes t_0 and π is a solution for the HTN planning problem:

$$P' = (s_0, g, \langle subtasks(t_0), t_1, \ldots, t_{k-1} \rangle, O, M)$$

3 GCT ENCODING

As Mali et al. (Mali and Kambhampati, 1998) already noted, their general encoding approach is equivalent to supplementing a classical planning encoding with HTN-specific constraints, which essentially enforce a valid grammar over the sequence of actions. A noteworthy restriction of their encoding is that each task is assumed to have some fixed maximal amount of primitive actions when fully reduced. Under this assumption, it is easily possible to "allocate" a fixed amount of propositional variables to each initial task in the plan, knowing exactly the point where a task, and ultimately the entire plan, will certainly have finished. In contrast, many common HTN problems can expand indefinitely due to recursive relationships. As such, it is difficult to decide in advance how many actions a given task will take, depending on which reductions are chosen and which facts hold before. The GCT encoding overcomes this limitation. For each step of the plan to be found, we assign one Boolean variable to the execution of some action as well as to the beginning and the end of some task or reduction. Then, we express all logical constraints which

the occurrence of a certain action, task, or reduction will imply at some step, such as the preconditions of an action or the possible reductions of a task. The encoding generally allows for recursive task relationships, as we do not set any upper limit on a single task's amount of primitive actions (except for the total plan length).

3.1 Rules of Encoding

All facts from the specified initial state hold at the beginning:

$$\bigwedge_{p \in s_0} holds(p, s_0) \wedge \bigwedge_{p \notin s_0} \neg holds(p, s_0) \qquad (1)$$

Likewise, all facts of the goal hold in the last state s_n:

$$\bigwedge_{p \in g} holds(p, s_n) \qquad (2)$$

The preconditions of an action a must hold for it to be executed at s_i, and the execution of that action implies its effects at the next state s_{i+1}:

$$execute(a, s_i) \Rightarrow \bigwedge_{p \in pre(a)} holds(p, s_i) \qquad (3)$$

$$execute(a, s_i) \Rightarrow \bigwedge_{p \in add(a)} holds(p, s_{i+1}) \wedge \bigwedge_{p \in del(a)} \neg holds(p, s_{i+1}) \qquad (4)$$

Facts only change their logical value if an action is executed which has this change as an effect:

$$\neg holds(p, s_i) \wedge holds(p, s_{i+1}) \Rightarrow \bigvee_{p \in add(a)} execute(a, s_i) \qquad (5)$$

$$holds(p, s_i) \wedge \neg holds(p, s_{i+1}) \Rightarrow \bigvee_{p \in del(a)} execute(a, s_i)$$

Exactly one action is executed. For each distinct pair of actions a and b, $b \neq a$ we have:

$$\neg execute(a, s_i) \vee \neg execute(b, s_i) \qquad (6)$$

The following clauses ensure the ordering and sequential execution of initial tasks. The first task t_0 begins at s_0. When a task t_j ends, then the next task t_{j+1} begins at the next state. The last task t_k ends at state s_{n-1} (such that its final effects hold in s_n):

$$taskStarts(t_0, s_0) \qquad (7)$$

$$taskEnds(t_j, s_i) \Leftrightarrow taskStarts(t_{j+1}, s_{i+1}) \qquad (8)$$

$$taskEnds(t_k, s_{n-1}) \qquad (9)$$

Clauses in (10) essentially provide a grammar for valid start and end points of tasks. Any start point of a task must precede a corresponding end point and vice versa.

$$taskStarts(t, s_i) \Rightarrow \bigvee_{i' \geq i} taskEnds(t, s_{i'}) \qquad (10)$$

$$taskEnds(t, s_i) \Rightarrow \bigvee_{i' \leq i} taskStarts(t, s_{i'})$$

The following clauses are added only for non-primitive tasks t. They define reduction variables in order to uniquely refer to some particular task reduction beginning or ending at some computational step. Remember, $D(t)$ is the set of all the possible reductions of t:

$$taskStarts(t, s_i) \Rightarrow \bigvee_{d \in D(t)} reducStarts(t, d, s_i) \qquad (11)$$

$$taskEnds(t, s_i) \Rightarrow \bigvee_{d \in D(t)} reducEnds(t, d, s_i)$$

To provide a meaning to the newly defined variables, it is enforced that the first subtask begins whenever the reduction begins, and the last subtask ends whenever the reduction ends:

$$reducStarts(t, \langle t', \dots \rangle, s_i) \Rightarrow taskStarts(t', s_i) \quad (12)$$

$$reducStarts(t, \langle a, \dots \rangle, s_i) \Rightarrow execute(a, s_i)$$

$$reducEnds(t, \langle \dots, t' \rangle, s_i) \Rightarrow taskEnds(t', s_i)$$

$$reducEnds(t, \langle \dots, a \rangle, s_i) \Rightarrow execute(a, s_i)$$

All subtasks (both primitive (13) and non-primitive (14)) of any occurring non-primitive task t need to be completed within the execution of t.

$$reducStarts(t, \langle \dots, a, \dots \rangle, s_i) \Rightarrow \bigvee_{i' \geq i} execute(a, s_{i'}) \qquad (13)$$

$$reducEnds(t, \langle \dots, a, \dots \rangle, s_i) \Rightarrow \bigvee_{i' \leq i} execute(a, s_{i'})$$

$$reducStarts(t, \langle \dots, t', \dots \rangle, s_i) \Rightarrow \bigvee_{i' \geq i} taskStarts(t', s_{i'}) \qquad (14)$$

$$reducEnds(t, \langle \dots, t', \dots \rangle, s_i) \Rightarrow \bigvee_{i' \leq i} taskEnds(t', s_{i'})$$

The preconditions of a reduction need to hold.

$$reducStarts(t, d, s_i) \Rightarrow \bigwedge_{p \in pre(d)} holds(p, s_i) \qquad (15)$$

Again, an additional set of variables is introduced denoting that some task t' at step i' is part of some reduction of task t at step i. They are required in order to reference the subtask relationship between two tasks in an unambiguous manner in the following subtask

ordering constraints.

$$reducStarts(t, \langle \ldots, t', \ldots \rangle, s_i) \Rightarrow$$
$$\Rightarrow \bigvee_{i' \geq i} subtaskStarts(t, s_i, t', s_{i'}) \quad (16)$$
$$subtaskStarts(t, s_i, t', s_{i'}) \Rightarrow taskStarts(t', s_{i'})$$
$$subtaskStarts(t, s_i, a, s_{i'}) \Rightarrow execute(a, s_{i'})$$
$$reducEnds(t, \langle \ldots, t', \ldots \rangle, s_i) \Rightarrow$$
$$\Rightarrow \bigvee_{i' \leq i} subtaskEnds(t, s_i, t', s_{i'}) \quad (17)$$
$$subtaskEnds(t, s_i, t', s_{i'}) \Rightarrow taskEnds(t', s_{i'})$$
$$subtaskEnds(t, s_i, a, s_{i'}) \Rightarrow execute(a, s_{i'})$$

Now, the following clauses enforce that the subtasks of any given task are totally ordered. Assume that a reduction of t is $d = \langle t_1', \ldots, t_k' \rangle$ and $1 \leq j < k$:

$$reducStarts(t, d, s_i) \wedge subtaskEnds(t, s_i, t_j', s_{i'}) \Rightarrow$$
$$\Rightarrow subtaskStarts(t, s_i, t_{j+1}', s_{i'+1}) \quad (18)$$
$$reducStarts(t, d, s_i) \wedge subtaskStarts(t, s_i, t_{j+1}', s_{i'+1}) \Rightarrow$$
$$\Rightarrow subtaskEnds(t, s_i, t_j', s_{i'}) \quad (19)$$

At this point, if one would just use variables $taskStarts(t', s_{i'})$ instead of the more explicit $subtaskStarts(t, s_i, t', s_{i'})$, then the clauses may lead to unresolvable conflicts. This is because some task t' may then in fact not belong to the reduction of task t at step i, but still impose restrictions on some actual subtask of t. This problem is avoided by introducing explicit variables for the subtask relationship between tasks.

3.2 Complexity

In the following, a brief analysis of the complexity of the GCT encoding is provided. Hereby, T, F, and A correspond to the respective amount of tasks, facts, and actions. Let $r = \max \{ |D(t)| \mid t \in T \}$ be the maximal amount of reductions per task and $e = \max \{ |subtasks(d)| \mid d \in D(t) \}$ be the maximal amount of subtasks per task reduction.

The GCT encoding features $O(S \cdot T)$ variables for the starts and ends of tasks, $O(S \cdot T \cdot r)$ variables for the starts and ends of task reductions, and $O(S^2 \cdot T^2)$ variables for modelling the subtask relationship between tasks. In addition, $O(S \cdot F)$ variables for the encoding of facts, $O(S \cdot A)$ variables for the execution of actions and $O(S \cdot A \log A)$ helper variables for the "at most one action" rule (see rule 6) are added. This leads to a total variable complexity of $O(S^2 T^2 + S(Tr + A \log A + F))$ which in practice is clearly dominated by the term $S^2 T^2$.

Similarly, the clause complexity of the GCT is asymptotically dominated by rules 18 and 19 which results in a total of $O(S^2 \cdot T^2 \cdot r \cdot e)$ clauses. The length of each clause is linear either in the amount of tasks, reductions, or steps.

The *Linear Bottom-Up Forward* (LBF) encoding (Mali and Kambhampati, 1998) can be compared to GCT due to its state-based nature. LBF featured a variable complexity of $O(S^2 \cdot TA \cdot r^2 \cdot e)$ and a clause complexity of $O(S^3 \cdot TA \cdot r^2 \cdot e)$. The T^2 factor in GCT's complexity is fundamentally caused by the admission of recursive task definitions, while the r^2 factor in the complexity of LBF comes from the arbitrary constraints that can be specified between reductions. However, the LBF encoding has a clause complexity which is cubic, not quadratic, in the number of encoded steps.

4 SMS ENCODING

The GCT encoding has various shortcomings. Most significantly, it has a quadratic complexity of variables and clauses, rendering it inefficient both for the encoding process and for the solving stage, and a complete re-encoding has to be performed for each additional computational step considered.

To improve this encoding, an initial idea has been to adjust the encoding in order to make it eligible for incremental SAT solving. This way, an abstract encoding is created only once and is then handed to the solver without any given limit on the maximal number of computational steps to consider. The solver can instantiate the clauses as needed and will remember conflicts learned from previous solving steps, therefore reducing run times (Gocht and Balyo, 2017).

However, to achieve an incremental SAT encoding for HTN planning, a number of challenges need to be considered. Specifically aiming at a separated encoding as proposed in (Gocht and Balyo, 2017), it is necessary to reformulate all clauses such that each clause only contains variables from the computational steps i and $i + 1$, for some clause-specific i, and that for each time step, all of the encoded clauses follow a single, general "construction scheme". In the following, such an encoding is presented.

4.1 Encoding Principle

The *Stack Machine Simulation* (SMS) encoding simulates a stack of tasks which is transformed between computational steps, always checking the task on top of the stack and either pushing its subtasks if it is non-primitive, or executing the corresponding action
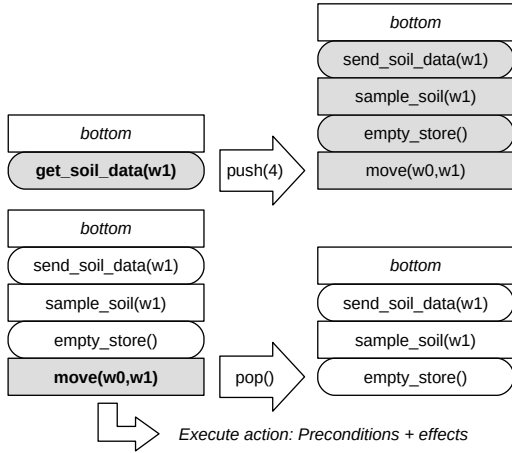
Figure 1: Illustration of the two central transitions between computational steps in the SMS encoding: processing a non-primitive task (top) and a primitive task (bottom).

if it is primitive. This central idea is illustrated in Fig. 1 with a simplified example from the IPC domain "Rover". Non-primitive tasks have rounded corners and primitive tasks are rectangular. First, the non-primitive task *get_soil_data(w1)* is reduced to some valid subtask sequence. In the next step, the primitive task *move(w0,w1)* is processed and removed. Using a stack memory and such transitions, the entire currently considered task hierarchy can be manipulated in a step-by-step manner, eliminating the necessity of linking steps that are further apart.

At the initial step, the first k positions of the stack are enforced to contain the k initial tasks, followed by an explicit *bottom* symbol. The goal is defined by the requirement of *bottom* to be on top of the stack, or in other words, of the stack being empty.

With this procedure, the initial tasks and their subtasks will be sequentially processed and broken down into subtasks until all tasks are removed and only *bottom* remains. Thereby, the SAT solver decides which cell of the stack contains which element at which step such that all constraints are satisfied.

The SMS encoding is inherently incremental: To extend an encoding with n computational steps into an encoding with $n+1$ computational steps, new clauses are added and the goal assertions (rules 21, 23) are updated. Apart from these simple operations, the previous formula can be reused without any changes.

4.2 Rules of Encoding

All facts specified in the initial state must hold:

$$\bigwedge_{p \in s_0} holds(p,s_0) \wedge \bigwedge_{p \notin s_0} \neg holds(p,s_0) \quad (20)$$

At the end, all facts from the goal must hold:

$$\bigwedge_{p \in g} holds(p,s_n) \quad (21)$$

The initial stack contains the initial tasks T and a *bottom* symbol afterwards. Let $T = \langle t_0,\ldots,t_j,\ldots,t_k \rangle$:

$$stackAt(j,t_j,s_0) \quad (22)$$
$$stackAt(k+1,bottom,s_0)$$

The stack must be empty in the end:

$$stackAt(0,bottom,s_n) \quad (23)$$

The execution of an action implies its preconditions to hold:

$$execute(a,s_i) \Rightarrow \bigwedge_{p \in pre(a)} holds(p,s_i) \quad (24)$$

The execution of an action implies its effects to hold in the next step.

$$execute(a,s_i) \Rightarrow \bigwedge_{p \in add(a)} holds(p,s_{i+1}) \wedge \bigwedge_{p \in del(a)} \neg holds(p,s_{i+1})$$
$$(25)$$

Facts only change if a supporting action is executed.

$$\neg holds(p,s_i) \wedge holds(p,s_{i+1}) \Rightarrow \bigvee_{p \in add(a)} execute(a,s_i)$$
$$(26)$$

$$holds(p,s_i) \wedge \neg holds(p,s_{i+1}) \Rightarrow \bigvee_{p \in del(a)} execute(a,s_i)$$

At each step, all the *push(k)* and *pop* operations are mutually exclusive. Let a and b be two operations in $\{push(k) \mid 1 \leq k \leq maxPushes\} \cup \{pop\}$. For each distinct operation a and b, we have:

$$\neg do(a,s_i) \vee \neg do(b,s_i) \quad (27)$$

If no *pop* operation is done, then no action is executed, enforced by a virtual action *noAction* which does not have any preconditions or effects.

$$\neg do(pop,s_i) \Rightarrow execute(noAction,s_i) \quad (28)$$

If an action a is on top of the stack, then it is executed (and only this action); additionally, a *pop* operation is done.

$$stackAt(0,a,s_i) \Rightarrow execute(a,s_i) \wedge do(pop,s_i) \quad (29)$$
$$execute(a,s_i) \Rightarrow stackAt(0,a,s_i)$$

If a non-primitive task t is on top of the stack, then one of its possible reductions d must be applied.

$$stackAt(0,t,s_i) \Rightarrow \bigvee_{d \in D(t)} reduc(t,d,s_i) \quad (30)$$

If a non-primitive task t on top of the stack is reduced by some specific reduction $d = \langle t'_1, \ldots, t'_k \rangle$, then a *push* by the amount of subtasks in the reduction is performed, and all of its preconditions must hold.

$$stackAt(0,t,s_i) \wedge reduc(t,d,s_i) \Rightarrow$$
$$\Rightarrow \left(do(push(k),s_i) \wedge \bigwedge_{p \in pre(d)} holds(p,s_i) \right) \quad (31)$$

The stack content moves according to the performed operation at a given step. Let t be any task or *bottom*, and $j > 0$:

$$do(push(k),s_i) \wedge stackAt(j,t,s_i) \Rightarrow$$
$$\Rightarrow stackAt(j+k-1,t,s_{i+1}) \quad (32)$$
$$do(pop,s_i) \wedge stackAt(j,t,s_i) \Rightarrow$$
$$\Rightarrow stackAt(j-1,t,s_{i+1}) \quad (33)$$

A non-primitive task and a reduction together define the corresponding subtasks as the new stack content at the positions which are freed by the occurring *push* operation. Assuming that a reduction of t is $d = \langle t'_0, \ldots, t'_{k-1} \rangle$:

$$stackAt(0,t,s_i) \wedge reduc(t,d,s_i) \Rightarrow$$
$$\Rightarrow \bigwedge_{j=0}^{k-1} stackAt(j,t'_j,s_{i+1}) \quad (34)$$

4.3 Variants

In addition to this encoding, which we will refer to as SMS-ut (unary tasks) from now on, two additional encodings of SMS have been considered.

The variant SMS-ur (unary reductions) does not encode tasks, but instead reductions as the content of the stack. This leads to transitional clauses of a different kind, and the set of variables deciding the chosen reduction are no more necessary.

In the variant SMS-bt (binary tasks), the stack content is encoded not with one variable for each possible task at each position, but instead with one variable for each binary digit of a number representing the task at that position. This reduces the total variable requirement for the stack content at each cell from $O(T)$ to $O(\log T)$ per computational step, but complicates some transitional clauses.

4.4 Complexity

In the following, the asymptotic complexity of clauses and variables of the encoding is discussed. The encoding variant SMS-ut is chosen for this purpose. Assume that after n computational steps, a plan π of length $|\pi| \leq n$ is found.

The amount of variables is dominated by the encoding of the stack itself: if a stack of size σ has been encoded, then $O(n \cdot \sigma \cdot T)$ variables are used to encode the stack. Additionally, the action executions generate $O(n \cdot A)$ variables and the state encoding generates $O(n \cdot F)$ variables. Moreover, $O(n \cdot r)$ variables representing the chosen reduction of the current task are needed for $r = \max \left\{ |D(t)| \,\middle|\, t \in T \right\}$. This leads to a total of $O\big(n \cdot (\sigma T + A + F + r)\big)$ variables. With some $\sigma \in O(n)$, this measure implies a quadratic term $n^2 T$.

Regarding the amount of clauses: the classic planning clauses include $O(n \cdot A)$ clauses for preconditions and effects of actions, and $O(n \cdot F)$ clauses for frame axioms (rule 26). To uniquely specify the content of the stack at each computational step, $O(n \cdot \sigma \cdot T \cdot e)$ transitional clauses (for each *push(k)*, $1 \leq k \leq e$, and for *pop*) are needed, where $e = \max \left\{ |subtasks(d)| \,\middle|\, d \in D(t) \right\}$. $O(n \cdot A)$ clauses link the stack operations with the execution of actions. As e is usually a small constant, a pairwise "at most one action" rule is added by introducing $O(n \cdot e^2)$ clauses (rule 27). For the actual transitions, each task which can be on top of the stack causes $O(r \cdot e)$ clauses; consequently, $O(n \cdot T \cdot re)$ clauses are needed. In total, $O\big(n \cdot (A + F + \sigma Te + e^2 + Tre)\big)$ clauses are added. Again assuming $\sigma \in O(n)$, the dominating factor becomes $n^2 Te$.

5 EXPERIMENTAL EVALUATION

We evaluate the proposed encodings on a set of planning benchmarks. Due to the significant differences regarding recursion in the HTN models (recursion is not supported in (Mali and Kambhampati, 1998)), we cannot directly compare the performances of our different encodings. However, the encodings proposed by (Mali and Kambhampati, 1998) have been a starting point for the ideas implemented in the GCT encoding. GCT is then used as baseline in our experiments for the SMS encodings that are build on a different approach.

A total of 120 instances from six domains have been considered. The instances have been generated by using problem generators from the International Planning Competition (IPC). All experiments have been conducted on a server with 24 cores of Intel Xeon CPU E5-2630 clocked at 2.30 GHz and with 264 GB of RAM, running Ubuntu 14.04. Each run has been cut off after five minutes or when its RAM usage exceeded 12 GB. Glucose (Audemard and Simon, 2009) has been used as the primary SAT solver and as a standalone application to efficiently solve non-incremental formulae produced by GCT. The applica-

Table 1: Run time scores of encodings.

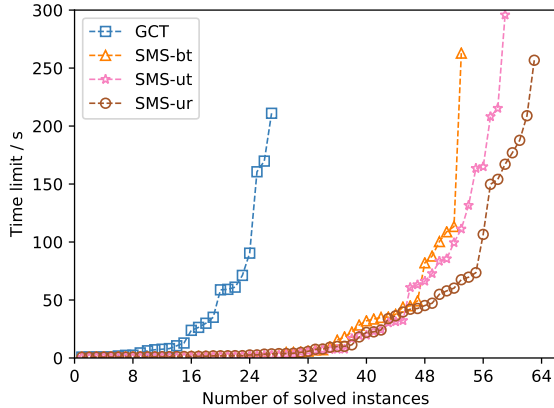| Domain	GCT	SMS-bt	SMS-ut	SMS-ur
Barman	0.09	1.90	1.96	**4.68**
Blocksworld	0.08	9.22	**10.94**	6.74
Childsnack	0.98	3.90	**9.95**	4.50
Elevator	4.21	**14.86**	13.32	10.29
Rover	0.44	**6.17**	5.40	5.58
Satellite	0.96	7.08	7.17	**16.08**
Total	6.75	43.13	**48.74**	47.88

Figure 2: Amount of instances solved by each encoding approach relative to the time limit.

tion Incplan (Gocht and Balyo, 2017) has been used as a unified approach to solve the incremental planning problems produced by SMS.

Fig. 2 illustrates the number of instances solved by each competing approach relative to the set time limit. With the GCT encoding, 27 instances have be solved within the maximum time limit. SMS significantly outperformed this baseline: the variant `bt` solved 53 instances, `ut` solved 59 instances, and `ur` solved 63 instances.

A domain-specific comparison of run time performances is provided in Table 1. For each tested instance, a score of 1 is attributed to the competitor with the lowest run time t^*, and a score of t^*/t is attributed to each further competitor with a run time t. Unfinished computations lead to a score of 0. For each domain, the scores of all instances are then summed

Table 2: Plan length scores of encodings.

Domain	GCT	SMS-bt	SMS-ut	SMS-ur
Barman	0.85	2.72	2.00	**5.00**
Blocksworld	2.00	10.00	**13.00**	11.00
Childsnack	3.00	6.00	**10.00**	8.00
Elevator	13.00	**16.00**	15.00	15.00
Rover	3.86	**6.62**	6.55	**6.62**
Satellite	4.00	9.61	11.79	**16.77**
Total	26.70	50.96	58.33	**62.40**

up. Overall, the variant `ur` performed best on the Barman and Satellite domains whereas the task-based variants `ut` and `bt` generally performed better on the Blocksworld and Elevator domains. The binary encoding variant `bt` only lead to small improvements on some domains while significantly worsening the run times on other domains. This result is consistent with the general belief that SAT constraints encoded in a binary manner can actually hinder the propagation of variable assignments and conflicts and, as a result, increase the execution time of SAT planning (Ghallab et al., 2004). The length of the plans found with the different encodings are compared in Table 2, with the scores being computed analogously to the run time scores. By design, the GCT encoding generally leads to the shortest possible plan, as the amount of possible primitive actions is increased one by one until a solution is found. The only exception for this is if *nop* actions are involved, which do not account for the plan length, but still cause additional solving iterations. Yet, the SMS encoding variants lead to plans which are nearly as short as the plans found with GCT.

6 RELATED WORKS

In his pioneering work, Sacerdoti (Sacerdoti, 1975) proposed a planner called NOAH (Nets of Action Hierarchies). The system was built up on a data structure called *procedure net*. This data structure introduced for the first time the concept of tasks network and reduction. These two concepts are today part of all the modern HTN planners. Mali (Mali and Kambhampati, 1998; Mali, 2000) proposed to use classical SAT solvers by encoding HTN planning problems into satisfiability problems. Recent and modern HTN planners such as SHOP (Simple Hierarchical Ordered Planner) (Nau et al., 1999; Ramoul et al., 2017) maintain states during the search process and explore a space of states. Each task network contains a representation of the state in addition to the tasks. A reduction is applicable if and only if some constraints expressed as precondition of the reduction hold in the state. A comprehensive comparison of these different works is given in (Georgievski and Aiello, 2015) and a complexity analysis of HTN planning in (Erol et al., 1994; Alford et al., 2015).

SAT solving has been successfully applied in the classical STRIPS planning context (Fikes and Nilsson, 1971) since the initial proposal (Kautz and Selman, 1992; Kautz et al., 1996). As a significant improvement, more efficient action encodings based on the execution of multiple actions in parallel (as long as some valid ordering on the actions exists) have

been proposed (Rintanen et al., 2004; Rintanen et al., 2006). Recently, incremental SAT solving has been shown to significantly speed up the planning procedure of Madagascar (Gocht and Balyo, 2017), based on the observation that reusing conflicts from previous solving attempts can lead to a much faster solving process (Nabeshima et al., 2006).

7 CONCLUSION

In the work at hand, we have presented new efficient approaches of totally ordered HTN planning by making use of SAT solvers. Previous SAT encodings for HTN planning problems had many shortcomings restricting their practical usage. We proposed two new encodings, GCT and SMS, which mend these shortcomings and thus can exploit efficient existing HTN grounding routines. SMS is specifically designed for incremental SAT solving and works reliably on all kinds of special cases which may occur in the considered planning domains. We experimentally evaluated both encodings and showed their practical applicability by running our planning framework on problem domains from the International Planning Competition (IPC). With the SMS encoding significantly outperforming GCT regarding overall run times while finding plans of comparable length, we have defined a new baseline of SAT planning on totally ordered HTN domains.

In future work, we will investigate alternative SAT encodings based on the general idea of SMS in order to further improve the overall performance of the approach. While the SMS encoding works reliably, its performance is limited by the amount of primitive actions in the shortest possible plan, and the stack size must be provided as parameter. Enhancements of SMS which function without any external parameters and which require less incremental iterations in order to find a solution may significantly speed up the planning process.

REFERENCES

Alford, R., Bercher, P., and Aha, D. (2015). Tight bounds for HTN planning with task insertion. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1502–1508.

Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404.

Biere, A. (2013). Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT competition*, 51.

Eén, N. and Sörensson, N. (2003a). An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer.

Eén, N. and Sörensson, N. (2003b). Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560.

Erol, K., Hendler, J., and Nau, D. (1994). UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Artificial Intelligence Planning Systems*, volume 94, pages 249–254.

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208.

Georgievski, I. and Aiello, M. (2015). HTN planning: Overview, comparison, and beyond. *Artifical Intelligence*, 222:124–156.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier.

Gocht, S. and Balyo, T. (2017). Accelerating SAT based planning with incremental SAT solving. *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 135–139.

Kautz, H., McAllester, D., and Selman, B. (1996). Encoding Plans in Propositional Logic. In *Proceedings of the International Conference on Knowledge Representation and Reasoning*, pages 374–384.

Kautz, H. and Selman, B. (1992). Planning as Satisfiability. In *Proceedings of the European Conference on Artificial Intelligence*, pages 359–363.

Mali, A. (2000). Enhancing HTN planning as satisfiability. In *Artificial Intelligence and Soft Computing*, pages 325–333.

Mali, A. and Kambhampati, S. (1998). Encoding HTN planning in propositional logic. In *Proceedings International Conference on Artificial Intelligence Planning and Scheduling*, pages 190–198.

Nabeshima, H., Soh, T., Inoue, K., and Iwanuma, K. (2006). Lemma reusing for SAT based planning and scheduling. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 103–113.

Nau, D., Cao, Y., Lotem, A., and Munoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In *Proceedings of the international joint conference on Artificial intelligence*, pages 968–973.

Ramoul, A., Pellier, D., Fiorino, H., and Pesty, S. (2017). Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools*, 26(5):1–24.

Rintanen, J. (2012). Planning as satisfiability: Heuristics. *Artificial Intelligence Journal*, 193:45–86.

Rintanen, J., Heljanko, K., and Niemelä, I. (2004). Parallel encodings of classical planning as satisfiability. In *European Workshop on Logics in Artificial Intelligence*, pages 307–319.

Rintanen, J., Heljanko, K., and Niemelä, I. (2006). Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080.

Sacerdoti, E. (1975). A structure for plans and behavior. Technical report, DTIC Document.