TaskSAT: A Domain-Specific Language and Verifier for Scheduling Task Networks

Klaus Havelund and Alessandro Pinto

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
{klaus.havelund, alessandro.pinto}@jpl.nasa.gov

Abstract. Model-based planning and scheduling (P&S), as studied in this paper, is an approach to control of autonomous systems. The approach relies on a model, here referred to as a tasknet, of what tasks are possible and constraints on these. From this model, a P&S system will generate a schedule of tasks satisfying the constraints, such that their execution will achieve a given goal. P&S can be executed repeatedly during a mission, with various time intervals. Programming tasknets turns out to be a complex activity. We propose a domain-specific language for programming tasknets, and a tool, TaskSAT, for exploring them. We check validity, meaning that a schedule exists, safety, meaning that all possible schedules satisfy properties stated in linear temporal logic, and realizability, meaning that a schedule exists for every initial state admitted by the tasknet. Temporal logic can also be used as part of the constraints determining what schedules are generated. Tasknets are translated to SMT problems and solved with the Z3 SMT solver. TaskSAT is developed to emulate MEXEC, a C++ P&S system in operational use at NASA/JPL, and is meant as verification and validation support for programming MEXEC tasknets.

1 Introduction

Autonomous systems act in the real world, making decisions without external supervision. Their basic operations include a few steps that repeat in a cycle [6,35]: observing the environment through sensors, assessing the situation, determining the best course of action to achieve goals, and finally selecting an action to be executed. The first two steps build an estimate of the state of the world, and are typically referred to as situational assessment [10]. The third step is typically called planning [14], and the last step is about executing a plan by dissecting it into lower-level actions.

In all practical applications, the process that compiles high-level goals, such as collecting samples on the surface of a planet, into low-level actions, such as applying current to the actuators, is not implemented as a monolithic function. Such a function would be too complex to be implemented correctly and efficiently, and would not be reusable across applications. The architecture of an autonomous system is, therefore, divided into a hierarchy of such functions [26,1]. Lower-level skills, such as driving along a trajectory, are exposed to higher-levels as tasks

that, when executed, result in changes to the state of the system according to a model that projects away low-level state information. High-level planning is a sequential decision making process that computes a set of tasks, and their execution time, in order to achieve a desired goal (or set of goals). Choosing the tasks is planning; assigning each of them a time is scheduling. We refer to the combination as planning and scheduling (P&S).

Several frameworks have been developed for defining P&S problems. For each framework, several algorithms have been developed to find solutions. For example, a popular framework is the Planning Domain Definition Language (PDDL) [12] in which a planning problem is defined by a set of predicates, a set of actions (i.e., the aforementioned tasks) with their preconditions and effects, an initial state, and a goal. The last international planning competition [20] featured more than 40 different planning systems that all solve variants of such planning problem. Another popular framework is hierarchical task networks [11,13], for which 9 planning systems have competed in the last international planning competition [21].

While implementing a correct planner for any of these frameworks is a challenging problem, authoring models to define a planning domain, namely its predicates and actions, can become a daunting task. A planning domain, in fact, must capture the set of valid plans for each initial and goal state. Validity is expressed in terms of the evolution of the predicates over time, also called behavior. Each plan corresponds to one or many possible behaviors of the system depending on whether the domain is deterministic or not. The art of crafting a good planning domain model consists in making sure that all possible behaviors, for all possible plans, are desirable (i.e., satisfy certain properties in addition to reaching the goal).

The process to develop planning domain models is iterative: a model is developed; a planner is used to compute a plan that is executed in a more accurate model of the environment; the resulting behavior is evaluated to understand whether the planning domain model needs to be changed. This process, however, is time consuming and does not provide sufficient guarantees. Furthermore, when an undesirable behavior is discovered through testing, an analysis needs to be carried out to understand what is wrong in the domain model. A complementary approach consists in verifying the planning domain directly against the properties that we want any system behavior to satisfy.

The problem we address in this paper is the *verification of a task network* (or tasknet for short). In our application domain, a planning domain is a set of tasks T together with their associated behavior models, defined over a state space $\Pi_{l \in L} D_l$, where each $l \in L$ is a state variable with domain D_l. Tasks must be scheduled according to their scheduling constraints, such as desired start time and duration. Given a property φ in Linear Temporal Logic (LTL) that defines the set of desirable behaviors over the state space, the verification method we developed, and implemented in the tool TaskSAT, verifies (i) that a valid schedule exists, (ii) whether all possible valid schedules result in executions that satisfy the temporal properties, and (iii) whether a valid schedule exists for every initial

state that the tasknet admits. The three checks differ in how they quantify over initial states and schedules: (i) is existential in both, (ii) is universal in both, and (iii) is universal in the initial states and existential in the schedules. Temporal logic properties can also be used as constraints influencing what schedules are generated.

The paper is organized as follows. In Section 2 we review related work. In Section 3, we show an example of the kind of task network that we consider in this paper, expressed in a domain-specific language that we have defined. In Section 4 we define the precise semantics of the task network language, and in Section 5 we show how we encode a tasknet as a formula over variables for task times and timeline values. A satisfiability modulo theories (SMT) solver can then be used to find schedules, solve the verification problem for all schedules, and, through a counterexample-guided loop, decide realizability. In Section 6 we describe the implementation and evaluate the approach on a synthetic benchmark of problems with different structural features.

2 Related Work

TaskSAT was developed as a response to the need for analyzing tasknets before their execution in the MEXEC architecture [28], which uses traditional planning and scheduling algorithms. Our approach of using an SMT solver for P&S is not new. From a formal methods perspective, P&S problems were originally considered as deductive proof problems, where the goal was to derive a sequence of actions that leads from an initial state to a goal state. The idea of encoding P&S problems as satisfiability problems was pioneered by Kautz and Selman [22]. They demonstrated that many planning problems could be effectively solved by encoding them into propositional logic and using a SAT solver to find a satisfying assignment that corresponds to a valid plan.

The work most closely related to ours is the SMTPlan+ framework by Cashmore, Fox, Long, and Magazzeni [7] which expanded this paradigm into an SMT problem to handle the continuous dynamics of PDDL+. A contribution of this work was the integration of a computer algebra system (SymPy) to perform symbolic integration of continuous effects during the encoding phase. This allows the SMT solver to reason about nonlinear continuous change using only algebraic expressions. Some features that our approach introduces that are not present in SMTPlan+ include the following. TaskSAT introduces a very readable DSL for defining tasknets. LTL formulas are encoded together with the tasknet, so LTL can be used both to restrict which schedules are valid and to state properties that every valid schedule must satisfy. Additionally, TaskSAT leverages Z3's Optimize module to handle optional tasks and soft constraints. We use a zone abstraction method to partition the time horizon. Another closely related work is the fuzz tool [16], which translates formulas in a very expressive first-order linear temporal logic to a SMT problem, from which Z3 generates command sequences, although this is for testing purposes.

Several efforts have been made in the past to use model checking techniques to analyze and verify plan models. These include [29], which compares the SMV [27], Spin [19], and Murphi [9] model checkers for modeling and verifying plan models expressed in the HSTS planner's DDL language, based on Allen's temporal interval logic, and [17], which translates a subset of JPL's realistic P&S language AML of the Aspen P&S system to the Promela language of the Spin model checker. None of these works, however, deals with time constraints. In contrast, the UPPAAL real-time model checker [31,4] has been used to model P&S problems with time constraints. In [24,23], domain models in the HSTS planner's DDL language are modeled as networks of timed automata, and the resulting models are verified for consistency and correctness.

Extensions of UPPAAL have been developed that are more directly relevant to scheduling [15]. UPPAAL CORA [32] performs cost-optimal reachability for linearly priced timed automata and returns an optimal execution/path through a timed automaton model [5]. UPPAAL TIGA [34] instead addresses timed games with controllable and uncontrollable transitions and returns a winning strategy, i.e. a state-feedback controller [3]. Optimization is primarily with respect to time. UPPAAL Stratego [33] further extends this line to stochastic priced timed games and supports not only synthesis of strategies, but also their quantitative optimization, comparison, and statistical evaluation as first-class objects [8]. Note that, unlike these extensions, which address the synthesis of a schedule or strategy, TaskSAT addresses the verification of the planning domain itself: whether every schedule a planner may return satisfies the properties. UPPAAL can answer such questions given a timed-automata encoding of the domain, as in [24]; TaskSAT instead encodes MEXEC's semantics, with clamped real-valued resources, rates and LTL constraints and properties, directly as SMT problems.

A quite different, but still related, approach is verifying plans/schedules generated by some means (via P&S or otherwise, e.g. manually) against a specification [25].

3 Example of a TaskSAT model

An example of a tasknet modeled in the TaskSAT language is shown in Figure 1. The task network `RoverMission` has a time horizon of 400 time units. This is a constraint on the maximum total length of a schedule. It defines five timelines, the state variables of the model, which the tasks update and query. The `battery` timeline represents the state of charge of the battery, as a percentage of the total capacity. It is a `rate` timeline: its value changes over time at a rate imposed by the tasks impacting it. The second interval, `bounds [0.0, 100.0]`, is the domain the value is clamped to. The first interval, `[10.0, 100.0]`, is the range the value must stay within in any schedule, which can be considered an invariant: a schedule that drains the battery below 10 is not valid. The `temperature` timeline is also a `rate` timeline. The `data` timeline is `cumulative` meaning that it increases or decreases at specific point in time (e.g, by adding a file to a database at the end of a task). The `location` timeline models a `state` that can only take two

possible values (`home`, `target`), and can only be assigned at the boundaries of a task (start or end). Finally, the `arm` timeline is `atomic`: an integer timeline that models a resource held by at most one task at a time; its value must stay in the range $[0, 1]$. A task claims it with a cumulative impact `+= 1` and releases it with `-= 1`; two tasks holding it simultaneously would leave the range, so the timeline enforces mutual exclusion. All timelines are initialized to a value in this case, but we allow for loose specification of the initial state.

The task network then declares tasks. A task can be given directly, like `heating` and `collect`, or as an instance of a task definition (`taskdef`), like `charge` and `drive`. A definition captures the behavior of a kind of task, its conditions and impacts, and can be instantiated any number of times; an instance names its definition after the colon and adds the scheduling constraints of that particular occurrence, such as durations and time windows, and may add conditions and impacts of its own. A definition may itself contain subtasks. However, scheduling takes place at the level of the subtasks.

Tasks can be `optional`, meaning that they appear in the schedule only if needed, or `request`, meaning that they appear if possible (not shown here). Each task, such as `drive`, can have several properties. It can have a `start_range` and `end_range` that define the required windows in which the task must start and end, and a `duration_range` for its duration, respectively. A task can specify a pre-condition `pre` and an invariant which are lists of constraints on the timelines[1]. The pre-condition must hold at the start of the task, before the task's own impacts take effect, while the invariant must hold at every time point of its execution thereafter, up to and including the end. For example, the `drive` task can only start if the battery level is between 60 and 100, and the battery level must remain above 15 during the entire execution of the task. Finally, a task defines a set of impacts at different time points of it execution. For example, the `collect` task claims the arm for the duration of its execution (a `maint` impact on the atomic timeline, which adds 1 when the task starts and subtracts it again when the task ends), consumes energy at a rate of 0.5 and decreases temperature at a rate of 0.2 during its execution (also `maint` impacts), and generates data for a total volume of 40 at the end of its execution (a `post` impact). Additionally, a task can be constrained to start `after` another task. In this example, the `drive` task must be scheduled after the `charge` task. A task can also be required to execute `containedin` another task. Both kinds of dependency may carry a time range: `after charge [10, 20]` requires the task to start between 10 and 20 time units after `charge` ends, and `containedin` takes ranges for the offsets from the enclosing task's start and end. Constraints on the time horizon, task durations, start and end times, and `after` and `containedin` dependencies are called *static*: they concern only the task times of a schedule. The remaining constraints, such as pre-conditions and impacts, are called *dynamic*: they concern the timeline values, and hence depend on the execution of the schedule.

A task network can also define a set of future and past time linear temporal logic (LTL) constraints. A schedule is acceptable only if it satisfies these con-

[1] TaskSAT supports equality and membership to an interval.

```
tasknet RoverMission {
  end = 400;
  timelines {
    battery : rate [10.0, 100.0] bounds [0.0, 100.0] = 60.0;
    temperature : rate [5.0, 40.0] bounds [0.0, 100.0] = 10.0;
    data : cumulative [0.0, 50.0] bounds [0.0, 100.0] = 0.0;
    location : state(home, target) = home;
    arm : atomic = 0;
  }
  taskdef charge_def {
    pre {battery in [0.0, 60.0];}
    impacts {
      maint {battery +~ 2.0;}
    }
  }
  taskdef drive_def {
    pre {battery in [60.0, 100.0]; temperature in [10.0, 40.0];}
    inv {battery in [15, 100.0];}
    impacts {
      maint {battery +~ −1.5;}
      post {location = target;}
    }
  }
  optional task charge : charge_def {
    duration_range [60,70];
  }
  task drive : drive_def {
    start_range [100, 120]; end_range [130, 160]; duration_range [30,40];
    after charge;
  }
  optional task heating {
    duration_range [10, 10];
    impacts {
      maint {battery +~ −0.5; temperature +~ 2;}
    }
  }
  task collect {
    duration_range  [20, 30];
    pre {location = target; battery in [60.0, 100.0]; arm = 0;}
    impacts {
      maint {arm += 1; battery +~ −0.5; temperature +~ −0.2;}
      post {data += 40.0;}
    }
  }
  constraints {
    prop not_charge_drive: always not (active(charge) and active(drive));
    prop not_charge_collect: always not (active(charge) and active(collect));
    prop temperature10: always temperature ≥ 10;
  }
  properties {
    prop target_reached: eventually (location = target and data ≥ 30);
    prop drive_charge: always (active(drive) −> eventually active(charge));
    prop temperature: always temperature ≥ 10;
  }
}
```

Fig. 1. A task network example for a prototypical rover.

straints. The set of constraints in Figure 1, for example, say that `charge` and `drive` are mutually exclusive, `charge` and `collect` are also mutually exclusive, and the temperature must be always greater than or equal to 10. Mutual exclusion

is common enough to have an abbreviation: the first constraint can be written `mutex [charge, drive]`, and `mutex` over a longer list states that no two of the tasks overlap. Its operands may also be task definitions, in which case all their instances are excluded pairwise.

The last section of the task network is a list of `properties`, that are also expressed by LTL formulas. For example, the property `target_reached` says that eventually the value of the `location` timeline becomes the `target` location, and a data volume of at least 30 is collected. The formulas in the `constraints` block and the `properties` block are both LTL formulas, but they play opposite roles. Constraints, like the static and dynamic constraints above, are part of the model: they restrict which schedules may be generated. Properties are assertions about the model: they do not influence which schedules are generated, but TaskSAT checks that they hold for all of them. A `final` block, with the same syntax as the timeline initialization, is a property of a special form: it states conditions on the timelines at the end of the last scheduled task, such as the battery being at least half charged, and is checked for all schedules like the other properties.

TaskSAT can be run with this input specification to check several properties. First, TaskSAT makes sure that the set of feasible schedules is not empty, in which case a witness is reported. The tasknet in Figure 1, in fact, does not admit a schedule, and TaskSAT reports such fact. Since `drive` must come after `charge`, the schedule starts with a `charge` task, which charges the battery to 100. The second task to be scheduled is a `drive` task that lasts at least 30 time units. Over this time, the battery is depleted at a rate of -1.5 per time unit, for a total of 45. At the end of the drive, the total battery level is 55 which is not enough to satisfy the precondition of the `collect` task. When the `after charge` constraint is removed from the `drive` task, TaskSAT proves that the set of schedules is not empty and reports the following witness[2]:

```
Schedule for TaskNet `RoverMission`:
  drive         : start =  120, end =  150
  charge        : start =  153, end =  213
  heating       : start =  214, end =  224
  collect       : start =  225, end =  255
```

After checking that the set of schedules is not empty, TaskSAT checks the three LTL properties for all valid schedules, and reports that all three hold.

4 Tasknets and Their Semantics

This section formally defines the following concepts: tasknet, schedule, and schedule execution. Based on these definitions, we define a valid schedule to be one that satisfies the static constraints of the tasks and whose execution satisfies all dynamic constraints of the tasknet, such as pre- and postconditions, and

[2] TaskSAT also produces a detailed schedule execution, showing changes of all timelines over time.

its temporal constraints. A valid schedule is one that a planner might return. Finally, the section defines what it means for a tasknet to be safe, namely that the execution of every valid schedule satisfies its temporal properties. The SMT encoding in Section 5 follows this semantics, with the differences listed in Section 5.8.

4.1 Tasknet Structure

We define the components of a tasknet bottom-up: timelines, timeline predicates, impacts, tasks and temporal formulas, and finally the tasknet itself. Throughout, L is a finite set of timeline identifiers and K a finite set of task identifiers.

Definition 1 (Timeline). *Each timeline identifier $\ell \in L$ is described by a tuple $\mathcal{L}(\ell) = (D_\ell, B_\ell, R_\ell)$ where D_ℓ is the value domain, $B_\ell \subseteq D_\ell$ is the clamping range (bounds), and $R_\ell \subseteq B_\ell$ is the safe range that must not be exceeded during scheduling.*

Timelines come in five kinds, which fix D_ℓ and the impacts admitted (impacts are defined below). A *state* timeline has a finite domain D_ℓ of names or numbers, with $B_\ell = R_\ell = D_\ell$, and admits assignments only. An *atomic* timeline has $D_\ell = B_\ell = \mathbb{Z}$ (it is not clamped) and $R_\ell = [0, 1]$, and admits cumulative impacts only; since two tasks holding it would exceed 1, it models mutual exclusion. The remaining three kinds have $D_\ell = \mathbb{R}$: a *claimable* timeline admits cumulative impacts during a task only (maint, so the amount claimed is returned at the end) and is not clamped ($B_\ell = \mathbb{R}$); a *cumulative* timeline admits assignments and cumulative impacts; a *rate* timeline additionally admits rate impacts and carries a declared initial rate $r^0_\ell \in \mathbb{R}$ (`initial_rate` in the concrete syntax, 0 if not declared). For $B_\ell = [b^-_\ell, b^+_\ell]$, values are clamped by $\mathsf{clamp}_{B_\ell}(x) = \max(b^-_\ell, \min(b^+_\ell, x))$.

Definition 2 (Timeline Predicate). *A* timeline predicate *is a formula χ generated by*

$$\chi ::= \ell \in C \mid \chi \wedge \chi$$

where $\ell \in L$ and C is a nonempty finite set whose elements are values $v \in D_\ell$ or intervals $[a, b]$. A valuation $s \in \prod_{\ell \in L} D_\ell$ satisfies $\ell \in C$, written $s \models \ell \in C$, iff $s(\ell) = v$ for some value $v \in C$ or $a \leq s(\ell) \leq b$ for some interval $[a, b] \in C$; and $s \models \chi_1 \wedge \chi_2$ iff $s \models \chi_1$ and $s \models \chi_2$. A set with several elements thus expresses alternatives, and equality is the special case $\ell \in \{v\}$.

Definition 3 (Impact). *An* impact *is a tuple*

$$i = (\ell_i, w_i, \mathsf{op}_i, a_i)$$

where $\ell_i \in L$ is the affected timeline, $w_i \in \{\mathsf{pre}, \mathsf{maint}, \mathsf{post}\}$ indicates when the impact occurs, $\mathsf{op}_i \in \{=, +, \sim, \approx\}$ is the operation (assignment, cumulative, rate increment, or rate assignment), and $a_i \in D_{\ell_i}$ is the value. If $\mathsf{op}_i \in \{+, \sim, \approx\}$, then $D_{\ell_i} = \mathbb{R}$, or ℓ_i is atomic and $\mathsf{op}_i = +$; $\sim$ and $\approx$ require a rate timeline, and $\approx$ is allowed at pre *and* post *only.*

Definition 4 (Task). *Each task identifier $k \in K$ is described by a tuple*

$$\mathcal{K}(k) = (startR_k, endR_k, durR_k, after_k, containedin_k, pre_k, inv_k, post_k, imp_k)$$

where $startR_k$, $endR_k$, and $durR_k$ are closed integer intervals. These specify ranges for the allowed start times, end times, and durations of the task. The constraints $after_k \subseteq K$ and $containedin_k \subseteq K$ are task dependencies. The constraints pre_k, inv_k, and $post_k$ are timeline predicates (Definition 2), indicating pre-conditions, invariants, and post-conditions of the task, respectively. The imp_k component is a finite set of impacts (Definition 3).

A task may in addition carry *soft constraints*, which are not part of the tuple: a priority $p_k \in \mathbb{Z}$ (higher is more important), a preferred start time $s_k^{\mathsf{pref}} \in \mathbb{N}$ and a preferred duration $d_k^{\mathsf{pref}} \in \mathbb{N}$. They do not affect which schedules are satisfying, only which one is preferred (Definition 17). The three ranges are not redundant. For example, $startR_k = [100, 120]$ and $endR_k = [130, 160]$ alone admit durations in $[10, 60]$, but $durR_k = [30, 40]$ is a sensible duration range.

Definition 5 (Temporal Formulas). *Temporal constraints $\varphi_{\mathsf{constr}}$ and properties φ_{prop} are formulas of a linear-time temporal logic over finite traces, generated by*

$$\varphi ::= \mathsf{true} \mid \mathsf{active}(k) \mid \ell \bowtie v \mid \tau \bowtie \tau \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \,\mathsf{S}\, \varphi$$
$$\tau ::= \mathsf{time} \mid \mathsf{start}(k) \mid \mathsf{end}(k) \mid c$$

where $k \in K$, $\ell \in L$, $v \in D_\ell$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Intuitively $\mathsf{active}(k)$ is true when task k is active, and the time terms τ denote the current time and the start and end times of tasks, so that orderings such as $\mathsf{end}(k_1) \leq \mathsf{start}(k_2)$ can be stated directly. The tool provides the abbreviations $\mathsf{sequence}[k_1, \ldots, k_n]$, stating that the tasks occur in this order, and $\mathsf{mutex}[k_1, \ldots, k_n]$, stating that no two of them overlap. The operators U and S are the classical LTL operators 'until' and 'since' (the past dual of 'until'). We use standard abbreviations: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$, $\mathsf{eventually}\ \varphi \equiv \mathsf{true}\,\mathsf{U}\,\varphi$, $\mathsf{always}\ \varphi \equiv \neg(\mathsf{eventually}\ \neg\varphi)$, $\mathsf{once}\ \varphi \equiv \mathsf{true}\,\mathsf{S}\,\varphi$, and $\mathsf{sofar}\ \varphi \equiv \neg(\mathsf{once}\ \neg\varphi)$.

Definition 6 (Tasknet). *A tasknet is a tuple*

$$TN = (T, L, K, K_{\mathsf{req}}, \mathcal{L}, \mathcal{K}, I, \varphi_{\mathsf{constr}}, \varphi_{\mathsf{prop}})$$

where $T \in \mathbb{N}$ is a finite time horizon, L is a finite set of timeline identifiers, K is a finite set of task identifiers with $K_{\mathsf{req}} \subseteq K$ being the required tasks, $\mathcal{L}$ assigns to each timeline identifier its tuple (Definition 1), $\mathcal{K}$ assigns to each task identifier its tuple (Definition 4), I is a timeline predicate (Definition 2) on the initial values, $\varphi_{\mathsf{constr}}$ is a temporal formula (Definition 5) used to constrain schedule generation, and φ_{prop} is a temporal formula for verification (not used in determining what schedules are generated; see Section 3).

Tasks in $K_{\mathsf{opt}} = K \setminus K_{\mathsf{req}}$ need not be scheduled. The concrete syntax distinguishes two flavours, `optional` tasks (scheduled only if needed) and `request` tasks (scheduled if possible); the distinction concerns only which schedule is preferred (Definition 17), not which schedules are valid, so the semantics otherwise treats both alike. We write K_{optional} and K_{request} for the two subsets of K_{opt}. In the following, we shall assume a tasknet TN with its components, which we shall refer to as needed. Task definitions (`taskdef`) are syntactic: before the semantics applies, every instance is merged with its definition (the instance's own fields take precedence, impacts and dependencies are unioned) and the definitions are discarded, so K is the resulting set of task instances.

4.2 Schedules and Execution

Time is discrete. This is a deliberate choice that follows MEXEC, whose plans are expressed in integer seconds: the horizon, the start, end and duration ranges, and the preferred times are all integers. Timeline values, in contrast, range over the reals where appropriate, and are evaluated at the integer time points.

We fix a finite discrete time domain $\mathbb{T} = \{0, 1, \ldots, T\}$ with interval notation $[a, b) = \{u \in \mathbb{T} \mid a \leq u < b\}$ and $(a, b] = \{u \in \mathbb{T} \mid a < u \leq b\}$. A *state* is a valuation $s \in \prod_{\ell \in L} D_\ell$, and a *trace* is a function $\sigma : \mathbb{T} \to \prod_{\ell \in L} D_\ell$.

Definition 7 (Schedule). *A* schedule *is a partial function* $\pi : K \rightharpoonup \mathbb{T} \times \mathbb{T}$ *assigning to task identifiers* k *a start and end time* $\pi(k) = (s_k, e_k)$*. The domain* $\mathrm{dom}(\pi) \subseteq K$ *is the set of scheduled tasks.*

Definition 8 (Fitting Schedule). *A schedule* π *is* fitting *a tasknet (i.e., it fits the static constraints of the tasknet) iff* $K_{\mathsf{req}} \subseteq \mathrm{dom}(\pi)$*, and for all* $k \in \mathrm{dom}(\pi)$ *with* $\pi(k) = (s_k, e_k)$*:* $0 \leq s_k < e_k \leq T$*,* $s_k \in \mathit{startR}_k$*,* $e_k \in \mathit{endR}_k$*,* $e_k - s_k \in \mathit{durR}_k$*,* $\forall k' \in \mathit{after}_k.\ k' \in \mathrm{dom}(\pi) \wedge e_{k'} < s_k$*,* $\forall k' \in \mathit{containedin}_k.\ k' \in \mathrm{dom}(\pi) \wedge s_{k'} \leq s_k \leq e_k \leq e_{k'}$*, and all task boundaries are pairwise distinct: for all distinct* $k_i, k_j \in \mathrm{dom}(\pi)$*,* $s_{k_i} \neq s_{k_j}$*,* $e_{k_i} \neq e_{k_j}$*,* $s_{k_i} \neq e_{k_j}$*, and* $e_{k_i} \neq s_{k_j}$*.*

Distinct task boundaries simplify the SMT encoding (Section 5): each zone boundary then carries the events of exactly one task, so simultaneous effects of different tasks never have to be combined. In practice this is rarely restrictive: if two tasks would share a boundary, moving one of them by one time unit usually gives a valid schedule.

Definition 9 (Impact Semantics). *The three impact types are illustrated in Figure 2, taken from MEXEC's user guide [28]. For a schedule* π *with* $k \in \mathrm{dom}(\pi)$ *and* $\pi(k) = (s_k, e_k)$*, an impact* $i \in \mathit{imp}_k$ *on* $\ell = \ell_i$ *contributes to one of four functions: the partial maps* $A_\pi(\ell, t)$ *and* $A^\rho_\pi(\ell, t)$*, the value and the rate assigned to* ℓ *at time* t*, and* $C_i(t)$ *and* $C^\rho_i(t)$*, the change of value and of rate caused by* i *at time* t*:*

- op_i is $=$ *or* op_i is $\approx$*:* $A_\pi(\ell, t)$*, respectively* $A^\rho_\pi(\ell, t)$*, is* a_i *if* $(w_i = \mathsf{pre} \wedge t = s_k) \vee (w_i = \mathsf{post} \wedge t = e_k)$*.*

– op_i is $+$ *or* op_i is $\sim$*:* $C_i(t)$*, respectively* $C_i^\rho(t)$*, is*

$$\begin{cases} a_i & \text{if } (w_i \in \{\mathsf{pre}, \mathsf{maint}\} \wedge t = s_k) \vee (w_i = \mathsf{post} \wedge t = e_k), \\ -a_i & \text{if } w_i = \mathsf{maint} \wedge t = e_k, \\ 0 & \text{otherwise.} \end{cases}$$

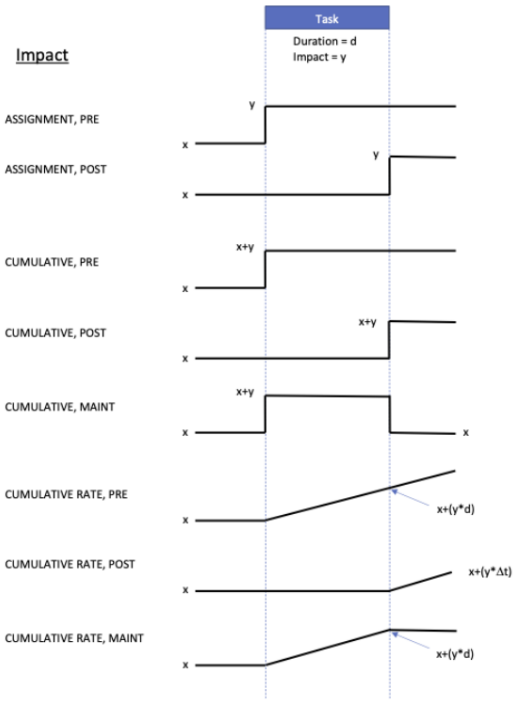


Fig. 2. Impact types (from [28]). Assignment (first two) have assignment impact where applied. Cumulative pre/post (next two) have addition/subtraction impact where applied, whereas in maint-mode (fifth), impact occurs at the beginning and then negated again at the end. Rates (last three) impact differently depending on where they are applied.

In the following, let π be a fitting schedule, σ_0 an initial state satisfying I, and $\mathcal{I}_\ell$ the set of impacts on timeline ℓ of the tasks in $\mathrm{dom}(\pi)$.

Definition 10 (Rate). *For each rate timeline* ℓ*, define the rate function* $\rho_{\pi,\sigma_0,\ell} : \mathbb{T} \to \mathbb{R}$ *by*

$$\rho_{\pi,\sigma_0,\ell}(t) = \begin{cases} A_\pi^\rho(\ell, t) & \text{if } A_\pi^\rho(\ell, t) \text{ is defined,} \\ r_\ell^0 + \sum_{i \in \mathcal{I}_\ell} C_i^\rho(0) & \text{otherwise, if } t = 0, \\ \rho_{\pi,\sigma_0,\ell}(t-1) + \sum_{i \in \mathcal{I}_\ell} C_i^\rho(t) & \text{otherwise,} \end{cases}$$

and let $\rho_{\pi,\sigma_0,\ell} \equiv 0$ *for every other timeline.*

Definition 11 (Execution). *For each timeline* $\ell \in L$*, define the execution value at a time point,* $\mathbb{E}_{\pi,\sigma_0,\ell} : \mathbb{T} \to D_\ell$ *inductively by*

$$\mathbb{E}_{\pi,\sigma_0,\ell}(0) = \mathsf{clamp}_{B_\ell}(\sigma_{0,\ell})$$

and for all $t < T$*,* $\mathbb{E}_{\pi,\sigma_0,\ell}(t+1) = A_\pi(\ell, t)$ *if* $A_\pi(\ell, t)$ *is defined, and otherwise*

$$\mathbb{E}_{\pi,\sigma_0,\ell}(t+1) = \mathsf{clamp}_{B_\ell}\Big(\mathbb{E}_{\pi,\sigma_0,\ell}(t) + \rho_{\pi,\sigma_0,\ell}(t) + \sum_{i \in \mathcal{I}_\ell} C_i(t)\Big).$$

The execution induced by a schedule and an initial state (π, σ_0) *is the trace* $\sigma_{\pi,\sigma_0} : \mathbb{T} \to \prod_{\ell \in L} D_\ell$ *defined pointwise by*

$$\sigma_{\pi,\sigma_0}(t)(\ell) = \mathbb{E}_{\pi,\sigma_0,\ell}(t)$$

for all $t \in \mathbb{T}$*,* $\ell \in L$*.*

Assignment impacts take precedence: if an assignment fires at (ℓ, t), then $\mathbb{E}_{\pi,\sigma_0,\ell}(t+1)$ is determined solely by that assignment. Like cumulative impacts, an assignment firing at t thus takes effect at $t+1$; in particular the precondition of a task, evaluated at s_k, sees the state before the task's own pre impacts. Unlike value impacts, a rate change firing at t is in effect at t itself, so that a maint increment a_i is in effect exactly on $[s_k, e_k)$ and contributes $(e_k - s_k)\, a_i$ to the value over the task's execution.

Definition 12 (Semantics of Temporal Formulas). *Let* π *be a schedule,* $\sigma : \mathbb{T} \to \prod_{\ell \in L} D_\ell$ *a trace, and* $t \in \mathbb{T}$*. Satisfaction of a temporal formula is defined inductively:*

$$\begin{aligned}
(\pi, \sigma, t) &\models \ell \bowtie v \text{ iff } \sigma(t)(\ell) \bowtie v, \quad \bowtie \in \{<, \le, =, \ge, >\},\\
(\pi, \sigma, t) &\models \mathsf{active}(k) \text{ iff } k \in \mathrm{dom}(\pi), \pi(k) = (s_k, e_k), \text{ and } s_k < t \le e_k,\\
(\pi, \sigma, t) &\models \tau_1 \bowtie \tau_2 \text{ iff } [\![\tau_1]\!]_{\pi,t} \bowtie [\![\tau_2]\!]_{\pi,t}, \text{ or a task in } \tau_1, \tau_2 \text{ is not in } \mathrm{dom}(\pi),\\
(\pi, \sigma, t) &\models \neg\varphi \text{ iff } (\pi, \sigma, t) \not\models \varphi,\\
(\pi, \sigma, t) &\models \varphi_1 \wedge \varphi_2 \text{ iff } (\pi, \sigma, t) \models \varphi_1 \text{ and } (\pi, \sigma, t) \models \varphi_2,\\
(\pi, \sigma, t) &\models \varphi_1 \,\mathsf{U}\, \varphi_2 \text{ iff } \exists t' \ge t.\ (\pi, \sigma, t') \models \varphi_2 \text{ and } \forall u \in [t, t').\ (\pi, \sigma, u) \models \varphi_1,\\
(\pi, \sigma, t) &\models \varphi_1 \,\mathsf{S}\, \varphi_2 \text{ iff } \exists t' \le t.\ (\pi, \sigma, t') \models \varphi_2 \text{ and } \forall u \in (t', t].\ (\pi, \sigma, u) \models \varphi_1,
\end{aligned}$$

where $[\![\mathsf{time}]\!]_{\pi,t} = t$*,* $[\![\mathsf{start}(k)]\!]_{\pi,t} = s_k$*,* $[\![\mathsf{end}(k)]\!]_{\pi,t} = e_k$*, and* $[\![c]\!]_{\pi,t} = c$*.*

4.3 Valid, Safe, and Realizable Tasknets

Validity, safety, and realizability are the three questions TaskSAT answers (Section 1).

Definition 13 (Satisfying Schedule). *A schedule π is satisfying a tasknet TN in an initial state σ_0, written* $\mathsf{Sat}(TN, \pi, \sigma_0)$*, iff (1) π is fitting, (2) $\sigma_0 \models I$, (3) $\forall \ell \in L, \forall t \in \mathbb{T}.\ \sigma_{\pi,\sigma_0}(t)(\ell) \in R_\ell$, (4) for all $k \in \mathrm{dom}(\pi)$ with $\pi(k) = (s_k, e_k)$: $\sigma_{\pi,\sigma_0}(s_k) \models pre_k$, $\sigma_{\pi,\sigma_0}(e_k) \models post_k$, and $\sigma_{\pi,\sigma_0}(t) \models inv_k$ for all t with $s_k < t \leq e_k$, and (5) $(\pi, \sigma_{\pi,\sigma_0}, 0) \models \varphi_{\mathsf{constr}}$.*

Definition 14 (Valid Tasknet). *A tasknet is* valid *iff there exist an initial state σ_0 and a schedule π that is satisfying in σ_0 (this is the scheduling problem):*

$$\mathsf{Valid}(TN) \iff \exists \sigma_0, \pi.\ \mathsf{Sat}(TN, \pi, \sigma_0).$$

Definition 15 (Safe Tasknet). *A tasknet is* safe *iff for every initial state σ_0 and every schedule π that is satisfying in σ_0, the temporal properties hold (this is the verification problem):*

$$\mathsf{Safe}(TN) \iff \forall \sigma_0, \pi.\ \big(\mathsf{Sat}(TN, \pi, \sigma_0) \Rightarrow (\pi, \sigma_{\pi,\sigma_0}, 0) \models \varphi_{\mathsf{prop}}\big).$$

Definition 16 (Realizable Tasknet). *A tasknet is* realizable *iff every initial state admitted by I has a satisfying schedule (this is the realizability problem):*

$$\mathsf{Realizable}(TN) \iff \forall \sigma_0 \models I.\ \exists \pi.\ \mathsf{Sat}(TN, \pi, \sigma_0).$$

Realizability only matters when I leaves initial values open, for example a battery level known to lie in an interval at the time the schedule is generated; a tasknet that is valid and safe may still be unschedulable from part of that interval.

Definition 17 (Preferred Schedule). *The* cost *of a schedule π is the tuple*

$$\mathsf{cost}(\pi) = \Big(|\mathrm{dom}(\pi) \cap K_{\mathsf{optional}}|,\ -|\mathrm{dom}(\pi) \cap K_{\mathsf{request}}|,\ -\sum_{k \in \mathrm{dom}(\pi)} p_k,\ \sum_{k \in \mathrm{dom}(\pi)} |s_k - s_k^{\mathsf{pref}}|,\ \sum_{k \in \mathrm{dom}(\pi)} |(e_k - s_k) - d_k^{\mathsf{pref}}| \Big)$$

where $\pi(k) = (s_k, e_k)$, a missing priority counts as 0, and the last two sums range only over tasks with a preferred start time and a preferred duration, respectively. A satisfying schedule π (with initial state σ_0) is preferred *iff no satisfying schedule has a lexicographically smaller cost.*

In words: as few optional tasks as possible, then as many request tasks as possible, then the highest total priority, then the smallest deviations from the preferred start times and durations. Preferences select among the satisfying schedules and do not change them, so validity, safety and realizability are independent of them. In its default optimize mode, the validity check of the tool reports a preferred schedule (Section 5.7).

5 SMT Encoding

In this section we describe a formal encoding of tasknet scheduling problems into Satisfiability Modulo Theories (SMT). The encoding translates a tasknet (Section 4) into a quantifier-free formula suitable for solving with the Z3 SMT solver [36]. Unless otherwise noted, all formulas presented in this section represent constraints that are generated and asserted to the Z3 solver. The exceptions are computational definitions, which describe the structure of intermediate calculations used in the constraint generation.

5.1 Zone Abstraction

Given a tasknet with horizon $T \in \mathbb{N}$ and tasks $K = \{k_1, \ldots, k_n\}$, we partition time into **zones** defined by task boundaries.

Zone Boundaries. Let Z be the number of zones, where $Z = 2|K| + 2$. We introduce zone boundary variables $z_0, z_1, \ldots, z_{Z-1}$ where:

$$z_0 = 0 \qquad z_{Z-1} = T \qquad z_i < z_{i+1} \text{ for all } i \in \{0, \ldots, Z-2\}.$$

Zone i corresponds to the time interval $(z_i, z_{i+1}]$ for $i \in \{0, \ldots, Z-2\}$.

Task-Zone Correspondence For each task $k \in K$, let s_k and e_k denote its start and end time variables. We enforce a bijection between internal zone boundaries $\{z_1, \ldots, z_{Z-2}\}$ and task boundaries $\{s_k, e_k \mid k \in K\}$:

$$\left[\bigwedge_{k \in K} \left(\bigvee_{j=1}^{Z-2} s_k = z_j \wedge \bigvee_{j=1}^{Z-2} e_k = z_j\right)\right] \wedge \left[\bigwedge_{j=1}^{Z-2} \bigvee_{k \in K} (s_k = z_j \vee e_k = z_j)\right].$$

Distinctness. All task boundaries are pairwise distinct:

$$\bigwedge_{k_i, k_j \in K,\, i \neq j} (s_{k_i} \neq s_{k_j}) \wedge (e_{k_i} \neq e_{k_j}) \wedge (s_{k_i} \neq e_{k_j}) \wedge (e_{k_i} \neq s_{k_j}).$$

These zone constraints are collectively denoted $\Phi_{\texttt{zones}}$.

5.2 Timeline Encoding

Timeline Variables For each timeline ℓ at zone boundary j, we introduce variables based on timeline type: $\sigma^\ell[j] \in \{0, \ldots, |D_\ell| - 1\}$ for state timelines, $\alpha^\ell[j] \in \mathbb{Z}$ for atomic timelines, constrained to $0 \leq \alpha^\ell[j] \leq 1$ at every zone boundary, and $\nu^\ell[j] \in \mathbb{R}$ for the real-valued timelines (claimable, cumulative, and rate).

Initial State For each timeline ℓ with initial value specification v_0^ℓ:

$$\sigma^\ell[0] = \texttt{encode}(v_0^\ell) \text{ (state)}, \qquad \alpha^\ell[0] = v_0^\ell \text{ (atomic)}, \qquad \nu^\ell[0] = v_0^\ell \text{ (real-valued)}$$

where $\texttt{encode} : D_\ell \to \{0, \ldots, |D_\ell| - 1\}$ maps state values (strings) to integer indices for efficient SMT encoding. An atomic timeline without a declared initial value starts at 0.

Range Constraints For real-valued timelines with range $[r_{\min}, r_{\max}]$:

$$\bigwedge_{j=0}^{Z-1} r_{\min} \le \nu^\ell[j] \le r_{\max}.$$

A range that is not declared defaults to $[-10^9, 10^9]$. These constraints, together with $0 \le \alpha^\ell[j] \le 1$ for atomic timelines, form $\Phi_{\texttt{ranges}}$. For cumulative and rate timelines with bounds $[b_{\min}, b_{\max}]$:

$$b_{\min} \le \nu^\ell[0] \le b_{\max}.$$

The initial-state constraints, these bounds at zone 0, and the initial predicate I (Definition 2) evaluated at zone 0 form $\Phi_{\texttt{init}}$. A state or real-valued timeline without a declared initial value is constrained at zone 0 only by I and its range.

5.3 Task Encoding

Static Constraints For each task $k \in K$, we encode $0 \le s_k \le e_k \le T$ and the duration, start, and end ranges (when specified):

$$d_{\min} \le e_k - s_k \le d_{\max} \qquad s_{\min} \le s_k \le s_{\max} \qquad e_{\min} \le e_k \le e_{\max}.$$

Precedence (k after k') and containment (k within k') are encoded as:

$$e_{k'} \le s_k \qquad s_{k'} \le s_k \wedge e_k \le e_{k'}.$$

A precedence may carry a gap $[g_{\min}, g_{\max}]$, encoded as $e_{k'} + g_{\min} \le s_k \le e_{k'} + g_{\max}$, and a containment may carry offsets $[o_{\min}, o_{\max}]$, encoded as $s_{k'} + o_{\min} \le s_k \le s_{k'} + o_{\max}$ and $e_{k'} - o_{\max} \le e_k \le e_{k'} - o_{\min}$. A dependency on a task definition rather than a task instance is encoded as a disjunction over the definition's instances.
These static constraints are denoted $\Phi_{\texttt{sched}}$.

Dynamic Constraints Preconditions hold at task start, invariants during execution, and postconditions at task end: for every zone index $j \in \{0, \ldots, Z-1\}$,

$$(z_j = s_k) \to \texttt{eval}(\texttt{pre}_k, j), \qquad (s_k < z_j \le e_k) \to \texttt{eval}(\texttt{inv}_k, j),$$
$$(z_j = e_k) \to \texttt{eval}(\texttt{post}_k, j).$$

Here $\texttt{eval}(C, j) = \bigwedge_{\ell \in C} (\bigvee_{c \in C[\ell]} \texttt{holds}(\ell, c, j))$ is the formula stating that the conditions C hold at zone j, where $\texttt{holds}(\ell, c, j)$ is the formula stating that ℓ satisfies c at zone j (e.g., $\nu^\ell[j] \in [a, b]$ or $\sigma^\ell[j] = v$).
These dynamic constraints are denoted $\Phi_{\texttt{conditions}}$.

5.4 Impact Semantics

An impact of task k is written as the tuple (k, ℓ, w, op, v), with timeline ℓ, w (`pre`, `maint`, or `post`) saying when it occurs, operation op, and value v (Definition 3). An assignment ($op \in \{=, \approx\}$) *fires at* z_i if $(w = \texttt{pre} \wedge z_i = s_k) \vee (w = \texttt{post} \wedge z_i = e_k)$.

State and Atomic Timelines State timelines admit only assignments, and only at task boundaries (no maint assignments, as in MEXEC). An impact firing at z_i determines the value at index $i+1$; the value at z_i, against which preconditions are evaluated, is unaffected. The transition from zone i to zone $i+1$ is:

$$\sigma^\ell[i+1] = \begin{cases} v & \text{if an assignment } (k, \ell, w, =, v) \in I \text{ fires at } z_i, \\ \sigma^\ell[i] & \text{otherwise.} \end{cases}$$

Atomic timelines admit only cumulative impacts, and change by their accumulated delta $\Delta^\ell[i]$, defined below: $\alpha^\ell[i+1] = \alpha^\ell[i] + \Delta^\ell[i]$. Atomic timelines are not clamped; the constraint $0 \leq \alpha^\ell[j] \leq 1$ makes a second simultaneous claim unsatisfiable, which enforces mutual exclusion.

Delta Accumulation We use I_Δ for the set of all delta impacts. For an atomic or real-valued timeline ℓ over zone i, spanning $(z_i, z_{i+1}]$, each delta impact $(k, \ell, w, +, v) \in I_\Delta$ contributes:

$$\text{cumul_contrib}^\ell_{k,w}[i] = \begin{cases} v & \text{if } w = \texttt{pre} \wedge z_i = s_k \\ v & \text{if } w = \texttt{maint} \wedge z_i = s_k \\ -v & \text{if } w = \texttt{maint} \wedge z_i = e_k \\ v & \text{if } w = \texttt{post} \wedge z_i = e_k \\ 0 & \text{otherwise} \end{cases}$$

The accumulated delta is:

$$\Delta^\ell[i] = \sum_{(k,\ell,w,+,v) \in I_\Delta} \text{cumul_contrib}^\ell_{k,w}[i]$$

Rate Integration We use I_R for the set of all rate impacts. For each rate timeline ℓ we introduce, next to the value variables $\nu^\ell[j]$, a rate variable $\rho^\ell[j] \in \mathbb{R}$ per zone boundary, holding the rate in effect during zone j. Its initial value is the declared initial rate of the timeline, 0 by default. A rate impact firing at z_i already applies in zone i, which starts at z_i, so a task's rate is in effect from the first zone in which it executes. With $\rho^\ell[0] = r^0_\ell$, a rate assignment $(k, \ell, w, \approx, r)$ firing at z_{i+1} sets $\rho^\ell[i+1] = r$; otherwise $\rho^\ell[i+1] = \rho^\ell[i] + \Delta^\ell_\rho[i+1]$, where Δ^ℓ_ρ, the accumulated rate change, is defined like Δ^ℓ above with rate impacts in place of delta impacts. The rate contribution over zone i, of duration $\Delta t_i = z_{i+1} - z_i$, is the integral of the rate in effect during the zone:

$$R^\ell[i] = \rho^\ell[i] \cdot \Delta t_i.$$

This product of two variables is the only nonlinear term in the encoding.

Zone Transition for Real-Valued Timelines The value at zone boundary $i+1$ is computed as:

$$\nu^{\ell}_{\texttt{raw}}[i+1] = \nu^{\ell}[i] + \Delta^{\ell}[i] + R^{\ell}[i].$$

If bounds are specified, which is possible for cumulative and rate timelines, the raw value is clamped with clamp_{B_ℓ} (Section 4); claimable and atomic timelines are not clamped. Finally, assignments override the result:

$$\nu^{\ell}[i+1] = \begin{cases} v & \text{if an assignment } (k,\ell,w,=,v) \in I \text{ fires at } z_i, \\ \mathsf{clamp}_{B_\ell}(\nu^{\ell}_{\texttt{raw}}[i+1]) & \text{otherwise.} \end{cases}$$

These timeline transition constraints are collectively denoted $\Phi_{\texttt{impacts}}$.

5.5 Temporal Logic Encoding

Temporal formulas are evaluated at zone positions. Let $\varphi[j]$ denote the encoding of formula φ at zone index j. We encode the core operators as follows (derived operators like $\vee$, $\rightarrow$, always, eventually, sofar, once are defined in Section 4):

$$\begin{aligned}
(\ell \diamond v)[j] &\equiv \nu^{\ell}[j] \diamond v \text{ (real-valued)}, \quad \alpha^{\ell}[j] \diamond v \text{ (atomic)}, \\
(\ell = v)[j] &\equiv \sigma^{\ell}[j] = \texttt{encode}(v) \text{ (state)}, \\
\mathsf{active}(k)[j] &\equiv \alpha^{\texttt{__}k\texttt{__active}}[j] = 1, \\
(\tau_1 \diamond \tau_2)[j] &\equiv \textstyle\bigvee_{k \in \text{tasks}(\tau_1,\tau_2) \cap K_{\text{opt}}} \neg\iota_k \ \vee \ \tau_1[j] \diamond \tau_2[j], \\
(\neg\varphi)[j] &\equiv \neg\varphi[j], \\
(\varphi_1 \wedge \varphi_2)[j] &\equiv \varphi_1[j] \wedge \varphi_2[j], \\
(\varphi_1 \,\mathsf{U}\, \varphi_2)[j] &\equiv \bigvee_{k=j}^{Z-1} \Big(\varphi_2[k] \wedge \bigwedge_{m=j}^{k-1} \varphi_1[m] \Big), \\
(\varphi_1 \,\mathsf{S}\, \varphi_2)[j] &\equiv \bigvee_{k=0}^{j} \Big(\varphi_2[k] \wedge \bigwedge_{m=k+1}^{j} \varphi_1[m] \Big),
\end{aligned}$$

where $\diamond \in \{<, \leq, =, \geq, >\}$; for a state timeline whose state names are numbers, $\ell \diamond v$ holds in exactly the states whose name, read as a number, satisfies it; $\texttt{__}k\texttt{__active}$ is an atomic timeline, initially 0, on which task k has the maint impact $+1$, so that it is 1 exactly in the zones in which k executes; $\text{tasks}(\tau_1, \tau_2)$ is the set of tasks mentioned in τ_1, τ_2, and ι_k is the Boolean variable stating that task k is scheduled (Section 5.7); and the time terms τ are encoded as $\mathsf{time}[j] = z_j$, $\mathsf{start}(k)[j] = s_k$, and $\mathsf{end}(k)[j] = e_k$.

5.6 Valid, Safe, and Realizable Tasknets

Satisfying Schedule. A schedule π and initial state σ_0 satisfy a tasknet if all constraints hold. The SMT encoding captures $\mathsf{Sat}(TN, \pi, \sigma_0)$ as:

$$\Phi_{\texttt{sat}} \equiv \Phi_{\texttt{zones}} \wedge \Phi_{\texttt{init}} \wedge \Phi_{\texttt{ranges}} \wedge \Phi_{\texttt{sched}} \wedge \Phi_{\texttt{conditions}} \wedge \Phi_{\texttt{impacts}} \wedge \varphi_{\texttt{constr}}[0].$$

Valid Tasknet. To determine if a tasknet is valid ($\mathsf{Valid}(TN)$), we check whether there exists a satisfying schedule:

$$\mathtt{SAT}(\Phi_{\mathtt{sat}}).$$

If satisfiable, the model provides zone boundaries $z_0, \ldots, z_{Z-1}$, task times s_k, e_k for each $k \in K$, and the values $\sigma^\ell[j]$, $\alpha^\ell[j]$, $\nu^\ell[j]$ of the state, atomic and real-valued timelines at each zone j.

Safe Tasknet. To determine if a tasknet is safe ($\mathsf{Safe}(TN)$), we check whether all satisfying schedules satisfy the property φ_{prop}. We search for a counterexample:

$$\mathtt{SAT}(\Phi_{\mathtt{sat}} \wedge \neg\varphi_{\mathsf{prop}}[0]).$$

If unsatisfiable, the tasknet is safe (all satisfying schedules satisfy φ_{prop}). The conditions of a `final` block are checked in the same way, at the end of the last scheduled task.

Realizable Tasknet. Let x denote the zone-0 timeline variables (the initial state) and y all remaining variables (zone boundaries, task times, inclusion variables, and timeline values at zones $j \geq 1$), so that $\Phi_{\mathtt{sat}}$ reads $\Phi(x, y)$, and let $\mathit{Init}(x)$ be the conjunction of the declared initial values, I, and the zone-0 domain constraints. Realizability ($\mathsf{Realizable}(TN)$) is the quantifier-alternating query

$$\forall x.\ \mathit{Init}(x) \rightarrow \exists y.\ \Phi(x, y).$$

Posed directly, this query combines quantifier alternation with the nonlinear products $r \cdot (z_{i+1} - z_i)$ of rate impacts, a combination that remains challenging for SMT solvers. We instead use a counterexample-guided loop in the style of counterexample-guided inductive synthesis (CEGIS) [30]. A solver S over x, whose models are the admitted initial states for which no schedule is yet found, is initialized with $\mathit{Init}(x)$ and the loop repeats: (1) if S is unsatisfiable, every initial state is covered and the tasknet is realizable; (2) otherwise take a model x^* of S and solve the quantifier-free planning query $\Phi(x^*, y)$; if it is unsatisfiable, x^* is an initial state with no satisfying schedule, reported together with its unsat core. If it is satisfiable with model (x^*, y^*), let σ be the *schedule skeleton* of y^*, the concrete zone boundaries, task times, and inclusion values. Substituting σ into Φ fixes every z_i, so the residual formula $\Phi_\sigma(x, y_{aux})$, where y_{aux} are the timeline values (and rates) at zones $j \geq 1$, is linear, and the blocking clause $\neg\exists y_{aux}.\ \Phi_\sigma(x, y_{aux})$, a quantified *linear* formula that Z3 handles reliably, is added to S. The clause removes exactly the set of initial states for which the fixed schedule σ is valid; this set contains x^*, which guarantees progress. Each verdict therefore rests on a concrete schedule or a concrete initial state. Over real-valued initial regions the loop is a semi-decision procedure, so it runs under an iteration and time budget and otherwise reports *unknown*; in practice one skeleton often covers the whole region.

5.7 Optional Tasks and Optimization

Optional tasks and the soft constraints of Definition 17 are supported using Z3's Optimize module for task validation (instead of the Solver module used for tasknet safety verification), which finds schedules satisfying all constraints while minimizing the cost of Definition 17. The Optimize module is used in the tool's default optimize mode, and only if the tasknet has optional or request tasks or soft constraints; otherwise validity is checked with the Solver module as well. For each task $k \in K_{\texttt{opt}}$, whether optional or request, we use Boolean variable ι_k to encode inclusion. The task's ranges, dependencies, conditions, and impacts are guarded by ι_k, and a dependency of a scheduled task on an optional or request task k' requires $\iota_{k'}$:

$$\iota_k \rightarrow (\text{task } k \text{ constraints}).$$

The objectives are the components of the cost of Definition 17, minimized lexicographically, where a task k is scheduled iff $\iota_k \vee k \in K_{\texttt{req}}$.

5.8 Relationship to the Semantics

The encoding follows the semantics of Section 4, but we have not proved it sound or complete with respect to it, and it differs in the following known respects: (1) every task boundary must equal an internal zone boundary, so $1 \leq s_k$ and $e_k \leq T - 1$, and a schedule with a task starting at 0 or ending at T is not found; (2) zone alignment is not guarded by ι_k, so an unscheduled optional or request task still occupies two zone boundaries, which can make a tasknet with a short horizon unsatisfiable; (3) clamping is applied once per zone rather than at every time step, and the results differ when an impact pushes a value beyond a bound and a rate pulls it back within the same zone; (4) safe ranges, conditions, and temporal formulas are evaluated at zone boundaries only, so for timelines with nonzero rates, and for formulas mentioning time, a violation or witness strictly inside a zone is missed; (5) a declared initial value outside the bounds makes the encoding unsatisfiable, whereas the semantics clamps it. Differences (1) and (2) concern the zone construction; (3) and (4) are inherent in the zone abstraction for timelines with rates.

6 Implementation and Evaluation

In this section, we briefly describe how TaskSAT is implemented, and we present a light evaluation of the tool. The tool is available on GitHub [18]. The tool was developed with extensive use of LLMs, which in our experience worked remarkably well, also for formal artifacts such as a Lean specification of the semantics. The code has not been reviewed line by line, but is validated by a test suite of over 300 tests.

6.1 Implementation

TaskSAT is implemented in Python and consists of several modules. A parser, implemented in the PLY parser framework [2], parses a tasknet into an abstract syntax tree (AST). A sequence of rewrite passes then reduces derived constructs to the core language of Section 4: parameters are substituted, task definitions are merged into their instances, and abbreviations such as `sequence`, `mutex` and the `active` predicate are expanded into boundary constraints and timeline operations. A wellformedness checker validates the AST to ensure that it is correct, including type checking. A static lint pass then reports specifications that are unschedulable for structural reasons, such as a precondition on a timeline no task ever writes, before any solving takes place. The AST is then transformed into SMT constraints, as outlined in Section 5 using the Z3 SMT solver (version 4.15.4.0) [36], specifically using the Python API provided by Z3. Finally, the SMT constraints are solved.

TaskSAT solves three problems: (1) a validity check that a schedule exists, (2) a safety check that all satisfying schedules satisfy the provided (if any) temporal properties, and (3) an opt-in realizability check that every initial state admitted by the tasknet has a schedule. TaskSAT can be run in one of two solver modes: in the default *optimize* mode, the validity check uses Z3's Optimize module to find a preferred schedule (Definition 17). In *satisfy* mode, the validity check runs two Z3 solvers in parallel, on two cores: a plain one, and one in which every constraint is labeled, so that when no schedule exists, Z3 returns an unsat core naming the conflicting constraints, which TaskSAT reports as an explanation. Neither solver is faster in general. Both run, and the first answer is used. The labeled solver can be disabled with a command-line option. Optimize mode uses a single solver, since Z3's Optimize module does not support unsat cores. Only the validity check in satisfy mode runs two solvers in parallel. The safety check uses a single solver without labels, and the realizability check a single labeled one, so that a counterexample comes with an unsat core. Temporal properties are verified one by one to allow for generation of counter example schedule executions, in case properties are violated. The transformation from a tasknet AST to SMT constraints follows the formal encoding described in Section 5. Listing 1.1 shows a code snippet of how pre-conditions, invariants, and post-conditions are encoded. For each task and for each zone boundary, implications encode when conditions must hold.

6.2 Evaluation

The example in Section 3 is shown to be unsatisfiable in 2.3 seconds; with the `after charge` constraint removed, a schedule is found in 1.7 seconds and the three properties are verified in 3.0 seconds, 4.7 seconds in total. However, the example is small and does not represent the complexity of more realistic tasknets. This section describes a light evaluation of TaskSAT on a collection of 16 more complex, but artificial tasknets. The evaluation focuses on execution times to find schedules (validity) and verify properties for all possible schedules (safety).

```
for t in self.all_scheduled_tasks:
    s = self.start_vars[t.id]
    e = self.end_vars[t.id]
    for j in range(Z): # For each zone boundary
        pre_formula = self._conds_holds_zone(t.pre, j) if t.pre else
            True
        inv_formula = self._conds_holds_zone(t.inv, j) if t.inv else
            True
        post_formula = self._conds_holds_zone(t.post, j) if t.post
            else True
        zj = z[j]
        self.solver.add(Or(zj ≠ s, pre_formula))
        self.solver.add(Or(zj ≠ e, post_formula))
        self.solver.add(Or(zj ≤ s, zj > e, inv_formula))
```

Listing 1.1. Generation of pre, invariant, and post condition constraints.

All tasknets were run in *satisfy* mode, with a limit of 300 seconds on the validity check and 10 seconds per property; the validity check was run with both solvers, taking the first answer, and with the plain solver alone. The evaluation was carried out on an Apple MacBook Pro with an M4 Max chip (14 cores) and 36 GB of memory, running macOS 26.6.2, Python 3.14.0 and Z3 4.15.4.

The tasknets are divided into four categories representing different patterns and verification objectives, as described below. Each category contains four tasknets with varying numbers of tasks, timelines, and properties. The tasknets were automatically generated using a parameterized Python script, available on the project's GitHub repository.

Category 1: Sequential Chain (seq). The sequential chain pattern tests performance on strict sequential dependencies. Each task T_i lasts between 1 and 6 time units, charges the battery, which is initially empty, at rate $+1.0$ while it executes, and has the precondition that the battery level is at least $5.0 \times (i-1)$. Since a task adds at most 6, this forces the tasks into an essentially sequential order $T_1 \to T_2 \to \cdots \to T_n$. One temporal property is verified for each tasknet, which checks that the final task T_n executes. The solver must explore exponentially many choices of durations as the chain length increases. Each tasknet is named `seq-N` where N is the number of tasks (5, 10, 15, or 20).

Category 2: Sequential Force (seqforce). This pattern represents a modification of the sequential chain pattern above, by adding explicit `start_range` and `end_range` temporal constraints, which place the tasks in the required sequential order, but with flexible time windows. This pattern represents a scheduling problem where time windows are more precisely defined, but still flexible. Each tasknet is named `seqforce-N` where N is the number of tasks (5, 10, 15, or 20).

Category 3: Parallel Resource (par). This pattern models controlled parallel task executions, with an upper bound on number of tasks that can execute concurrently, controlled by a number of timelines. Each timeline represents an

Table 1. Performance on the benchmark suite (16 benchmarks) in satisfy mode, with a 300-second limit on the validity check and 10 seconds per property. Validity is given for two configurations: *both*, the default, which runs a solver with labels (yielding unsat cores) and one without in parallel, and *plain*, which runs the solver without labels alone. Total is for the default configuration and includes parsing and encoding; the safety check always runs a single solver without labels, so it does not depend on the configuration. Abbreviations: TO = time out, i.e. the time limit was reached (300 seconds for validity, 10 seconds per property); (m/n TO) in the safety column = m of the n properties timed out; not run = the safety check was not run, since validity timed out; N/A = no properties. *With a longer limit, this property is found violated: T_{15} is optional in seqforce-15, so a valid schedule may omit it.

Benchmark	**Tasks**	**Validity (both)**	**Validity (plain)**	**Safety**	**Total**
Sequential Chain					
seq-5	5	0.70s	27s	0.03s	0.76s
seq-10	10	102s	TO	0.11s	102s
seq-15	15	TO	TO	not run	TO
seq-20	20	TO	TO	not run	TO
Sequential Force					
seqforce-5	5	0.27s	46s	0.03s	0.33s
seqforce-10	10	2.5s	20s	0.12s	2.7s
seqforce-15	15	17s	157s	TO*	28s
seqforce-20	20	110s	TO	0.62s	111s
Parallel Resource					
par-5-2	5	0.02s	0.02s	N/A	0.05s
par-10-3	10	0.25s	0.52s	N/A	0.33s
par-20-5	20	133s	79s	N/A	133s
par-30-10	30	TO	TO	N/A	TO
Property Verification					
prop-5-10	5	0.06s	0.03s	40s (4/10 TO)	41s
prop-10-20	10	0.38s	0.13s	86s (8/20 TO)	86s
prop-10-50	10	0.36s	0.15s	220s (20/50 TO)	220s
prop-15-100	15	53s	33s	581s (47/100 TO)	634s

independent resource on which up to k tasks can operate in parallel ($k = 3$ in all benchmarks), and is implemented as a counter that increments on task start and decrements on task end, and enforced through pre-conditions ($counter \leq k - 1$). There are no defined temporal properties. Each tasknet is named `par-N-L` where N is the number of tasks (5, 10, 20, or 30) and L is the number of timelines (2, 3, 5, or 10).

Category 4: Property Verification (prop). This pattern focuses on verification of temporal properties. It keeps tasks simple with fixed durations and abundant resources. The complexity arises from verifying a growing number of temporal logic properties. Each tasknet is named `prop-N-P` where N is the number of tasks (5, 10, or 15) and P is the number of properties (10, 20, 50, or 100).

Table 1 presents the results. In the default configuration, 13 of the 16 benchmarks complete. Neither configuration dominates: running both solvers completes one to two orders of magnitude sooner on the sequential benchmarks (seqforce-5: 0.27 versus 46 seconds), where the labeled solver is the fast one, but takes longer on par-20-5 (133 versus 79 seconds), where the plain solver alone is fastest. On the sequential chain, the time grows exponentially with the number of tasks; narrowing the start and end ranges (seqforce) makes the problems much easier. In the property benchmarks, properties over state timelines and task activity are mostly decided in well under a second; the main reason for the timeouts are properties that bound the value of a rate timeline, as in $\mathsf{always}\,(\mathit{battery} \geq 10)$, since they involve the nonlinear product of rate and duration. Since each property is checked by a separate solver call, the safety time grows linearly with the number of properties. Overall, TaskSAT handles 10–15 tasks in a reasonable time.

MEXEC tasknets. We also applied TaskSAT to tasknets translated from MEXEC models, with up to 21 timelines, 19 task definitions and 215 task instances. Direct verification did not scale to these networks. More useful was to let an LLM analyze such a tasknet, identify potential problems, and generate a small TaskSAT model demonstrating each one. This is not unlike the use of model checkers such as Spin [19] on abstractions extracted from code. This exposed two actual defects: a task requiring a state transition that no impact performs, and a pattern in which a container task requires a resource to be held from its start, although only the contained task, which starts later, claims it.

7 Conclusion

We presented a domain-specific language for modeling tasknets that reflects the concepts of the MEXEC P&S system. The language supports timelines over finite and continuous domains and allows impacts such as assignments, delta updates, and linear rate updates. A system like TaskSAT can be used for several purposes, including fast prototyping of tasknets, as an actual input language to a system like MEXEC, or as a target language for a verifier for MEXEC tasknets. We defined its semantics and explained the translation to SMT. Using the Z3 SMT solver, we generate schedules, verify all possible schedules against temporal logic properties, and check that every admitted initial state has a schedule. Temporal logic properties can also be used to constrain the generation of schedules. Beyond what is described here, the tool provides parameters, automatic instantiation of task definitions referenced by dependencies, visualization of tasknets, schedules and counterexamples, and a web interface. Future work includes a soundness and completeness argument for the encoding, a study of the interaction between soft constraints (priorities and preferred start times and durations) and verification, namely which of the preferred schedules are reported as witnesses and counterexamples, a Python API, handling of environment actions that can invalidate a schedule, and scalability: beyond solver-level techniques such as symmetry breaking and incremental solving, we are exploring a compositional check that verifies one repeated mission block and concludes for any number

of repetitions. Due to the translation to SMT we can furthermore make the TaskSAT language more expressive, including the temporal logic.

Acknowledgments. The research was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. We thank Gregg R. Rabideau for having patiently answered our questions about the MEXEC P&S system of which he is a main developer. © 2026. California Institute of Technology. Government sponsorship acknowledged.

Disclaimer. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

1. Albus, J.S., Huang, H.M., Messina, E.R., Murphy, K., Juberts, M., Lacaze, A., Balakirsky, S.B., Shneier, M.O., Hong, T.H., Scott, H.A., Proctor, F.M., Shackleford, W.P., Michaloski, J.L., Wavering, A.J., Kramer, T., Dagalakis, N., Rippey, W.G., Stouffer, K.A., Legowik, S.: 4D/RCS Version 2.0: A Reference Model Architecture for Unmanned Vehicle Systems. NIST Interagency/Internal Report (NISTIR) 6910, NIST (2002)
2. Beazley, D.M.: PLY (Python Lex-Yacc). https://www.dabeaz.com/ply/, accessed: 2026-10-19
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for playing games! In: Computer Aided Verification (CAV 2007). Lecture Notes in Computer Science, vol. 4590, pp. 121–125. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_14
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
5. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Priced timed automata: Algorithms and applications. In: Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004). Lecture Notes in Computer Science, vol. 3657, pp. 162–182. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11561163_8
6. Boyd, J.R.: The Essence of Winning and Losing. Self-published (1995), part of the collection "A Discourse on Winning and Losing"
7. Cashmore, M., Fox, M., Long, D., Magazzeni, D.: A compilation of the full PDDL+ language into SMT. In: Coles, A.J., Coles, A., Edelkamp, S., Magazzeni, D., Sanner, S. (eds.) Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS). pp. 79–87. AAAI Press (2016)
8. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: UPPAAL Stratego. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16

9. Dill, D.: Murphi. https://github.com/tyler-utah/Murphi2019 (2026), accessed 2026-10-19
10. Endsley, M.R.: Toward a theory of situation awareness in dynamic systems. Human Factors **37**(1), 32–64 (1995)
11. Erol, K., Hendler, J., Nau, D.S.: Semantics for hierarchical task-network planning (1994)
12. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research **20**, 61–124 (2003). https://doi.org/10.1613/jair.1129
13. Georgievski, I., Aiello, M.: An Overview of Hierarchical Task Network Planning (2014). https://doi.org/10.48550/arXiv.1403.7426
14. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Elsevier (2004)
15. Grobelna, I., Gajewski, K., Karatkevich, A.: A systematic review on the applications of UPPAAL. Sensors **25**(11) (2025). https://doi.org/10.3390/s25113484
16. Havelund, K., Clark, T., Reddy, V.: Fuzz testing with temporal constraints. In: Proceedings of the 22nd International Colloquium on Theoretical Aspects of Computing (ICTAC 2025). Marrakesh, Morocco (Nov 2025), 24–28 November 2025
17. Havelund, K., Groce, A., Holzmann, G., Joshi, R., Smith, M.: Automated testing of planning models. In: Proceedings of the Fifth International Workshop on Model Checking and Artificial Intelligence (MoChArt'08). Lecture Notes in Artificial Intelligence, vol. 5348, pp. 90–105. Springer, Patras, Greece (2009). https://doi.org/10.1007/978-3-642-00431-5_6
18. Havelund, K., Pinto, A.: TaskSAT: SMT-based task network scheduling and verification. https://github.com/nasa-jpl/tasksat (2025), accessed: 2026-10-19
19. Holzmann, G.: Spin. https://spinroot.com/spin/whatispin.html (2026), accessed 2026-10-19
20. International Planning Competition 2023 Classical Tracks. https://ipc2023-classical.github.io/, accessed: 2026-10-19
21. International Planning Competition 2023 HTN Tracks. https://ipc2023-htn.github.io/, accessed: 2026-10-19
22. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence (ECAI). pp. 359–363. John Wiley & Sons, Vienna, Austria (August 1992)
23. Khatib, L., Muscettola, N., Havelund, K.: Mapping temporal planning constraints into timed automata. In: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01). IEEE Press, Cividale Del Friuli, Italy (2001)
24. Khatib, L., Muscettola, N., Havelund, K.: Verification of plan models using UPPAAL. In: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems (FAABS'00). Lecture Notes in Computer Science, vol. 1871, pp. 114–122. Springer, Maryland (2001). https://doi.org/10.1007/3-540-45484-5_9
25. Kurklu, E., Havelund, K.: A flight rule checker for the LADEE Lunar spacecraft. In: 17th International Colloquium on Theoretical Aspects of Computing (ICTAC'20). LNCS, vol. 12545 (2020)
26. Matni, N., Ames, A.D., Doyle, J.C.: Towards a Theory of Control Architecture: A quantitative framework for layered multi-rate control (2024). https://doi.org/10.48550/arXiv.2401.15185
27. McMillan, K.L.: SMV. https://mcmil.net/smv.html (2026), accessed 2026-10-19
28. MEXEC user's guide, version 1.5.0. Jpl internal document, Jet Propulsion Laboratory, California Institute of Technology (May 2024)

29. Penix, J., Pecheur, C., Havelund, K.: Using model checking to validate AI planner domain models. In: Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard (1998)
30. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, University of California, Berkeley (Dec 2008)
31. UPPAAL Team: UPPAAL. https://uppaal.org/ (2026), accessed 2026-10-19
32. UPPAAL Team: UPPAAL Cora. https://uppaal.org/features/#cora (2026), accessed 2026-10-19
33. UPPAAL Team: UPPAAL Stratego. https://uppaal.org/features/#uppaal-stratego (2026), accessed 2026-10-19
34. UPPAAL Team: UPPAAL Tiga. https://uppaal.org/features/#tiga (2026), accessed 2026-10-19
35. Wooldridge, M.: Reasoning about Rational Agents. MIT Press (2000)
36. The Z3 SMT solver. https://github.com/z3prover/z3