

Experimental Evaluation of a Planning Language Suitable for Formal Verification^{*}

Rick W. Butler¹, César A. Muñoz², and Radu I. Siminiceanu²

¹ NASA Langley Research Center, Hampton, Virginia, USA.

² National Institute of Aerospace, Hampton, Virginia, USA.

Abstract. The marriage of model checking and planning faces two seemingly diverging alternatives: the need for a planning language expressive enough to capture the complexity of real-life applications, as opposed to a language simple, yet robust enough to be amenable to exhaustive verification and validation techniques. In an attempt to reconcile these differences, we have designed an abstract plan description language, ANMLite, inspired from the Action Notation Modeling Language (ANML) [17]. We present the basic concepts of the ANMLite language as well as an automatic translator from ANMLite to the model checker SAL (Symbolic Analysis Laboratory) [7]. We discuss various aspects of specifying a plan in terms of constraints and explore the implications of choosing a robust logic behind the specification of constraints, rather than simply propose a new planning language. Additionally, we provide an initial assessment of the efficiency of model checking to search for solutions of planning problems. To this end, we design a basic test benchmark and study the scalability of the generated SAL models in terms of plan complexity.

1 Introduction

Historically, the fields of planning and formal verification have had very little interaction. As branches of Artificial Intelligence, planning and scheduling have mainly focused on developing powerful search heuristics for efficiently finding solutions to extremely complex, specialized problems that take into account intricate domain specific information. Traditionally, this field and has been heavily influenced by the goals of one of the major sponsoring agencies (NASA, Ames Center) and its affiliated institutes (RIACS, JPL). The planning software produced is in a perpetual process of expansion to include the latest and fanciest capabilities: re-planning, on-the-fly reconfiguration, resource allocation, etc.

These goals are often contrasting with the main purpose of our field of formal verification. To make the planning software ready for space missions and pass the certification process, the main thrust of our activities are in a completely opposite direction: simplify, reduce complexity, understand the concepts, make software amenable to exhaustive verification.

^{*} Research funding was provided by the National Aeronautics and Space Administration under the cooperative agreement NCC-1-02043.

To this end, we seek to define a simple language that can be used to describe planning problems. Hopefully, by drastically restricting the constructs in the language, two benefits accrue: (1) the language will be easy to understand and write, and (2) the language will lend itself to formal verification.

We have named the language ANMLite [4] because it was developed to support the analysis of planning domains described in the Action Notation Modeling Language (ANML) [17] under development at NASA Ames[10]. ANML succeeds other planning languages, such as PDDL [12] and NDDL, that have been used in the software package EUROPA2 [2].

In ANMLite, a planning problem consists of a finite set of disjoint timelines, a set of valid actions for each timeline, and a set of temporal constraints that govern the correct scheduling of the actions. The constraints can be broadly categorized into two groups. The first group is specified by a *transition relation* and only involves actions on the same timeline. These constraints express the valid succession of actions along the timeline. The transition relation disallows overlapping actions and gaps on a timeline. The second category are general constraints, expressed in some logic of choice, which specify cross-timeline relationships between actions. The temporal logic must be chosen with care. It has to be rich enough to cover all the significant relations that can occur (such as Allen temporal operators [1], a popular logic in planning), but simple enough to avoid inconsistencies and ambiguities. Furthermore, since we seek to develop a framework for formal verification, it must be translatable into a form suitable for model checking or theorem proving. We are currently targeting the SAL model checker [7].

SAL is a framework for combining different tools to calculate properties of concurrent systems. The SAL language is designed for specifying concurrent systems in a compositional way. It is supported by a tool suite that includes state of the art symbolic (BDD-based) and bounded (SAT-based) model checkers, and a unique "infinite" bounded model checker based on SMT solving. Auxiliary tools include a simulator, deadlock checker, and an automated test case generator.

2 Related Work

To the best of our knowledge, formal verification work in planning and scheduling has not been attempted before the initiation of the SAVH (Spacecraft Autonomy for Vehicles and Habitats, now Automation for Operations, A4O) in 2005. There exists previous work on adjacent topics, however.

Model checking has been applied in the context of logics with actions [14] and knowledge representation [11]. The symbolic model checker of choice in this case is NuSMV. Another related area of work is the use model checking for on-line diagnosis of systems [5], applied in this particular case to the the study of the Livingstone framework.

The avenue of using constraint solvers for planning problems has been explored in [16] (based on temporal interval logic and attributes) and [15] (solving a particular class of disjunctive temporal problems via SAT solving techniques).

Test case generation for planning has been attempted in [9]. While testing is not an exhaustive verification technique, it is always seen as complementary and is mostly motivated by the need of low cost and performance. Finally, runtime monitoring, a lightweight version of verification, has been applied to the fault protection engine of the Deep-Impact spacecraft flight software [8].

3 ANMLite language concepts

We briefly describe the basic ANMLite concepts. The full syntax of the language is given in the Appendix. For further information, an extensive discussion of the ANMLite language semantics is given in the NASA Technical Memorandum [4].

3.1 Timelines and Actions

Discovering a suitable sequence of actions on a timeline is fundamental to solving a planning problem. The first step in defining the problem is to identify all the actions that can be scheduled on a timeline. In ANMLite, this is declared as in the following example:

```
TIMELINE A ACTIONS
  A0
  A1: [_ , 10]
  A2: [2, _]
```

This specification defines the timeline **A** and its three actions: **A0**, **A1**, and **A2**. Actions **A1** and **A2** have time duration constraints: **A1** takes at most 10 time units and **A2** takes at least 2 time units. Usually, there are also constraints on the sequence of actions, so an intuitive, unambiguous specification of these constraints is highly desirable. There are two different approaches to the specification of these constraints:

- Assume that all action sequences are possible unless specifically forbidden and then specify the sequences that are *not* allowed.
- Assume that no sequences are allowed and then systematically add the allowed sequences.

We have currently opted for the second approach. This is different from many AI planning systems, but it follows the approach frequently used in the formal methods community. We currently believe that this leads to a clearer specification, though we recognize that we may be biased by the historic conventions of our discipline.

3.2 Transitions

The transition relation on a timeline is similar to state-transition systems. Here, the states are the actions and a directed edge represents a valid transition between states. We have used the same construction deployed in the Abstract Plan Preparation Language (APPL) [3]. Hence, the transition relation is a set of pairs of actions, which can be declared by listing for each action the (complete) set of its successors, as in the following example:

TRANSITIONS

```
A0 -> A1 -> A2 -> (A0 | A1 | A3)
A3 -> A2
```

The flexibility of the language is increased by allowing parametrization of actions. For example, the following

```
A1(x,y: animal): [10,_]
```

defines an action **A1** with two parameters of type **animal** that takes at least 10 units of time.

We allow more restrictive forms of transitions to be defined using a simple parameter matching scheme, with implicitly declared variables. For example,

```
A1(cat,u) -> A2(u,_)
```

states that only **A1** actions with a first parameter equal to **cat** are to be followed by an **A2** action and that the first parameter of **A2**, represented by the variable **u**, must be equal to the second parameter of **A1**. Unless explicitly specified on a different constraint, no other transition from **A1** is allowed.

Multiple timeline instances are defined using a **VARIABLE** section:

VARIABLES

```
t1, t2: A
t3: B
```

This specification declares two distinct instances, **t1** and **t2**, defined by **TIMELINE A**, and one instance **t3** defined by **TIMELINE B**.

The variables of the same **TIMELINE** share the transition relation, but might still behave differently, in case specific constraints are declared in the general constraint section. This is beneficial in terms of keeping the model compact, and it is frequently seen in practice.

3.3 Goal Statements and Initialization

In ANMLite, goals can be specified by an action name. Initial states can also be specified using an **INITIAL-STATE** declaration though they are not necessary.

INITIAL-STATE

```
|-> t1.A0
|-> t2.A1
```

This specifies that **A0** is the first scheduled action on timeline **t1** and that **A1** is the first scheduled action on timeline **t2**. A generic form is also allowed

INITIAL-STATE

```
|-> A.A0
```

This means that on *every* timeline of type **A**, **A0** is the first scheduled action.

4 Constraints

The transition statements are adequate to specify the allowed sequences of actions on a timeline, but they cannot be used to specify constraints between actions on different timelines. The constraint section is used to accomplish this. The ANMLite constraints are built upon a simple but powerful foundation: linear inequations between the start and end timepoints of actions. Expressions may contain at most one variable on each side of the relational operator, e.g.

`A1.start + 16 < B2.end`

Restricting the constraint language to these simple linear relationships enables a very natural translation into the SAL model checking language (see Sec. 5).

4.1 Repetitive Actions

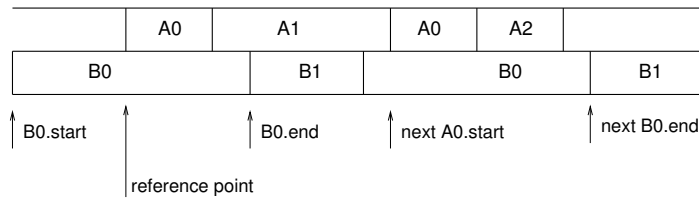
It is often the case that the same action is scheduled several times on a timeline. For example, crew activities on a space station are mostly routine tasks repeated every day, intertwined with other specific activities. Two occurrences of the same action are distinct because they are scheduled at different time intervals. There is a clear need to distinguish between these intervals when writing a set of constraints, which can refer to all or just one of these instances. We consider two approaches: (1) provide a new construct to establish a reference point for a constraint (called the **at** expression) and (2) introduce the qualifier **next** for a second occurrence of an action in the same constraint.

Neither of these two constructs were previously considered in planning languages, yet there is an obvious risk of ambiguities in their absence.

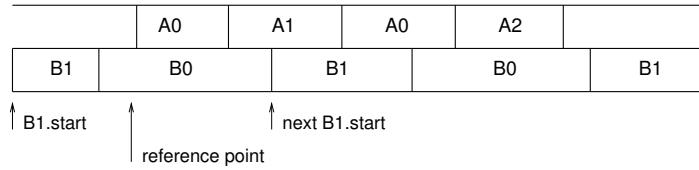
The at Expression. In the following example

`at A0.start: B0.end < next A0.start`

all actions that are active at the timepoint `A0.start` are the current ones. The next instance after the completion of the current one is the **next** one. For example



If the action is not active at the reference point, then the “current” is the last completed one and the next is the first occurrence after the reference point.



It should also be noted that there is an implicit universal quantifier in every constraint. If the reference point involves action **A1** and **A1** can occur multiple times on a timeline, then this constraint applies every time **A1** is scheduled.

4.2 Timeline Instance Specific Constraints

Constraints can be specialized by using a timeline variable in the constraint. Suppose we have

VARIABLES

t1,t2: A

t3,t4: B

CONSTRAINTS

t1.A1.start < t4.B1.end

This constraint only affects timelines **t1** and **t4**. But the constraint

A1.start < B1.end

is equivalent to four constraints:

t1.A1.start < t3.B1.end t1.A1.start < t4.B1.end

t2.A1.start < t3.B1.end t2.A1.start < t4.B1.end

4.3 Vacuous solutions

Consider the Allen logic operator **A1 contains B1**. A constantly debated issue is whether the constraint can be satisfied by the following timeline

A0	A2		
B0		B1	B2

Because the Allen operator has the implicit quantifiers **FORALL A1: EXISTS B1: A1 contains B1**, this constraint can be vacuously met in case **A1** is never scheduled. Whether this is desirable or not is a recurring theme in the plan specification domain. A non-ambiguous semantics should be chosen for all these situations.

4.4 Summary of Constraint Semantics

There are two major issues that need to be resolved when interpreting a constraint in ANMLite:

- Determination of the time point from which the current and next instances of an action can be disambiguated.
- Determination of which actions are universally quantified and which ones are existentially quantified.

These issues are orthogonal and hence the most general solution allows an independent specification of them. The first issue is handled by the **at** expression. The second issue is handled by a syntactic convention, namely, that the last term in the chain of inequalities determines the universally quantified action. This choice is justified by the way the constraint checking has to be performed (efficiently) in the SAL models. The other alternative, of attaching the universal quantifier to the first term, is equally valid from the theoretical point of view.

5 Translating ANMLite to SAL

Although using a model checker might not be the most efficient means of finding a solution to a planning problem, building a translator has provided a sanity check on the meaning of the language constructs.

5.1 Simple Example

We will begin our look at the technique for translating ANMLite to SAL with a very simple two timeline example:

PLAN ex1	
TIMELINE A	TIMELINE B
ACTIONS	ACTIONS
A0: [2, _]	B0: [2, _]
A1	B1: [1, 10]
A2	
TRANSITIONS	TRANSITIONS
A0 -> A1 -> A2	B0 -> B1
END A	END B
INITIAL-STATE	GOALS
-> A.A0	A.A2
-> B.B0	B.B1
END ex1	

Corresponding to the timeline and action declarations, the following types are generated:

```

A_actions: TYPE = {A0, A1, A2, A_null};
B_actions: TYPE = {B0, B1, B_null};

```

In addition to the declared actions, a null state is created for each of the timelines. There are two purposes for these extra states. They provide a means for the completion of an action when the action has no successor and also a convenient mechanism for recording when a goal state has been reached and completed on each timeline.

The generated SAL model consists of three modules: module **A_m**, corresponding to timeline A, module **B_m** to timeline B, and module **Clock**, which advances time.

5.2 Multiple variables

If there are multiple variables of a timeline, say

```
VARIABLES
```

```
  t1,t2: A
```

then a variable identifier type is generated,

```
  A_ids: TYPE = {t1,t2};
```

and the module `A_m` is parametrized with the variable id

```
  A_m[i: A_ids]  : MODULE =
```

Furthermore, since each instance of the timeline is a separate module, all the local and global variables in the parametrized module have to be arrays. For example, a non-parametrized module `A_m` might include a variable for `AO_start`:

```
  GLOBAL
```

```
    AO_start: TM_rng;
```

The parametrized version has to be

```
  GLOBAL
```

```
    AO_start: ARRAY A_ids OF TM_rng;
```

This way, the start of `AO` for instance `t1` is referred to as `AO_start[t1]`.

5.3 Modeling Time

Time is governed by the generic clock module. We have experimented with various implementations of this module. The most straightforward approach is to have the clock module increment the current time by one time unit at each step. This approach is very simple but is not scalable, because the system would traverse a very large number of states that are identical with the exception of the clock value. This state explosion problem is exacerbated by problems with large planning horizons. A possible alleviation of problem is to allow the clock to advance by larger amounts. However, this still does not rule out the traversal of multiple states in an interval of time when nothing interesting happens (from the point of view of action change). The best solution in this case is to use the concept of timeouts [13] that model the *event driven* clocks. In this approach, each timeline maintains a future clock value where an event is scheduled to occur, and time jumps directly to the next interesting event. The timeouts are stored in an array of timepoints and the clock module determines the next (minimum value in the future) timeout.

The modules are composed asynchronously.

```
System: MODULE = A_m [] B_m [] Clock;
```

The SAL model checker will be used to search through all possible sequences of actions on the timelines to find sequences which satisfy all of the constraints specified in the ANMLite model. These constraints fall into two broad categories:

- Timing constraints that impact durations and start/stop times of actions.
- Simple relationships between `start` and `end` variables

The search is started at time 0 and proceeds forward in time until the planning horizon `TM_rng` is reached.

5.4 Model Variables

The **GLOBAL** sections of all of the timeline modules contain variables which record the action that is scheduled during the current time:

```
GLOBAL
    AO_start: TM_rng,
    BO_start: TM_rng,
    B1_start: TM_rng,
    B_state: B_actions,
    A_state: A_actions,
```

The `_state` variables store the current action and the `_start` and `_end` variables record the start and end times of the actions.

5.5 Transitions

The ANMLite **TRANSITIONS** section is the major focus of the translation process. The SAL **TRANSITIONS** section is constructed from this part of the ANMLite model. When a transition occurs, an action is completed and another transition is initiated. No empty time slots are allowed. For example, the following

```
TRANSITIONS
    A0 -> A1 -> A2
```

is translated into three SAL transitions, which are labeled as follows:

```
A0_to_A1:      %% A0 -> A1
A1_to_A2:      %% A1 -> A2
A2_to_A_null:  %% A2 -> A_null
```

The first transition is guarded by the following expression:

```
A_state = A0
AND time >= AO_start + 2
```

The first conjunct ensures that this transition only applies when the current action on the timeline is A0 and the second conjunct insures that the duration of the action is at least 2 time units. This corresponds to the fact that A0 was declared as A0: [2,_].

The **GOALS** statement is translated into the following SAL specification:

```
sched_sys: THEOREM
    System |- AG(NOT(A_state = A_null AND B_state = B_null));
```

Since the “null” states can only be reached from the goal states (i.e., A2 and B1), these efficiently record the fact that the appropriate goal has been reached and completed on each timeline. Note that the ANMLite goal statement has been negated. Therefore when the model checker is instructed to establish the property, any counterexample provided by SAL will serve as a feasible realization of the plan.

5.6 Translating Constraints

There are major conceptual differences between *specifying* constraints and *checking* constraints that need to be reconciled. In principle, the specification is declarative by nature and the modeler usually looks “forward” in time in expressing what needs to happen in order for the plan to complete. The checking of the plan is operational by nature, because **start** and **end** variables are assigned values as they occur, hence testing that a constraint is valid cannot be performed until the last timepoint has occurred. Therefore, in the checking of the constraints the modeler has to look “backwards” in time.

For example, the constraint **A.start < B.end < C.start** cannot be established when **A** starts. Even if **B** has not ended yet, its relationship to the start of **C** cannot be established.

The mechanism of checking constraints with a model checker is based on assigning and updating the values of timeline **state** and each action **start** and **end** variables. This is performed at the timepoints when a timeline transitions from one action to another, according to the **TRANSITIONS** section.

Repetitive actions require special care, as multiple occurrences of the same actions will overwrite the values of the corresponding **start** and **end** variables, so only the most recent one is actually available (and possibly the previous occurrence, given that we allow the **next** qualifier).

For example, if there is a transition **A1 -> A2** on timeline **A**, the following updates are necessary:

```
- A_state' = A2
- A1_end' = time
- A2_start' = time
```

A constraint is, in principle, applicable to all the transitions that affect the variables present in the constraint expression. That is, a **start** variable is relevant to *entering* an action, while the **end** variable is relevant to *exiting* an action. Transition guards are generated for the events that are involved.

The general approach of translating constraints into transition guards consists of determining the last timepoint in the chain and substituting that term with the value of the system variable **time**. For example, in the constraint

$$A1.start + 4 < B1.start < C1.end$$

the last timepoint is **C1_end**. The transitions of relevance to this timepoint are from a predecessor of **C1** to **C1**.

6 Experiments

To instrument a scalability study for the model checker, we have explored two options. On the one hand, we have already accumulated a small benchmark of ANML models used for basic checks of the ANML operator semantics against the EUROPA2 [2] implementation. On the other hand, the model checker is not

able to solve even moderately complex problems, with no more than a handful of timelines. Therefore, we took the path of generating random models to fit into the current range of capabilities of SAL.

6.1 Real Models

The small suite of examples includes 73 models designed to investigate the basic Allen temporal logic operators that are at the core of the EUROPA2 [2] package. The main purpose was not the study of performance but to expose semantics issues, inconsistencies in the solutions, and insights into the subtleties of the logic (such as vacuous solutions, repetitive actions, the need for quantifiers, etc). Additionally, a space station crew activities and a dock worker robots models have been developed. Even though not nearly as sophisticated as necessary for practical purposes, they were still too complex to model check with SAL.

6.2 Random Models

The major challenge in using the “real” models is that it is very tedious and time consuming to manually scale up the models (e.g. increase the number of timelines, actions, constraints) in a meaningful way.

Instead, from the statistical point of view, it might be better to just generate random models. They are obviously meaningless from the planning point of view, but they are better from the experimental point of view, since they are completely “unbiased”.

In our experiments, we used 3,900 random models, generated by a small C program which takes in a few parameters:

- the number of timelines, T ;
- the number of actions on a timeline, A ;
- the number of transitions in a timeline, R ;
- the number of constraints/Allen operators in the constraint section, C ;

The transition graph is generated randomly. The program picks a source and target action (without self loops) and adds an edge. For simplicity, the constraints are all of the form $endpoint + constant < endpoint$, where endpoints are randomly selected from the set: $action.\{start/end\}$.

A completely random generator would most likely produce a large number of planning problems with no solution, as is the case of disconnected transition graphs. Therefore, the random generation is “steered” towards more meaningful setups. Instead of completely random graphs (which are likely to contain unreachable goal states), we always add the backbone chain $A_0 \rightarrow A_1 \rightarrow \dots \rightarrow A_{n-1}$ and make A_{n-1} the goal state. This gives the model checker something useful to work with and increases the probability of an existing solution.

The set of sample parameters is the following:

- $T \in \{1, 2, 3, 4, 5\}$;
- $A \in \{3, 4, \dots, 10\}$;

- R takes sample values between the minimal (backbone) graphs with $A - 1$ edges and the full graph with $\frac{A(A-1)}{2}$ edges. The test harness covers values for the fraction of "fullness" $f \in \{0, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, 1\}$, that is

$$R = (A - 1) + f \cdot \left(\frac{A(A-1)}{2} - (A - 1) \right);$$

- C takes sample values from "nothing" to "a lot": $\{0, \frac{A}{4}, \frac{A}{3}, \frac{A}{2}, A, 2A\}$.

6.3 Results

We ran our batch of experiments using SAL version 3.0 on a 64bit, 3.2 GHz machine with 8GB of memory running RH Enterprise Linux version 2.6.9-5ELsmp. We collected the runtime for each model with the `time` command, using a timeout of 30 minutes (after which the SAL instance was aborted).

Outcome. The analysis has to take into account the outcome of a run: a solution is found, no solution is found, or the run is aborted when reaching the timeout cutoff. Since the model checking query was set up as a negation of the statement "no solution exists", in case a counterexample is found, it is then displayed (which is a time-consuming operation for a model checker). Figure 1 shows the outcome breakdown for the runs, function of the four parameters in the experiment.

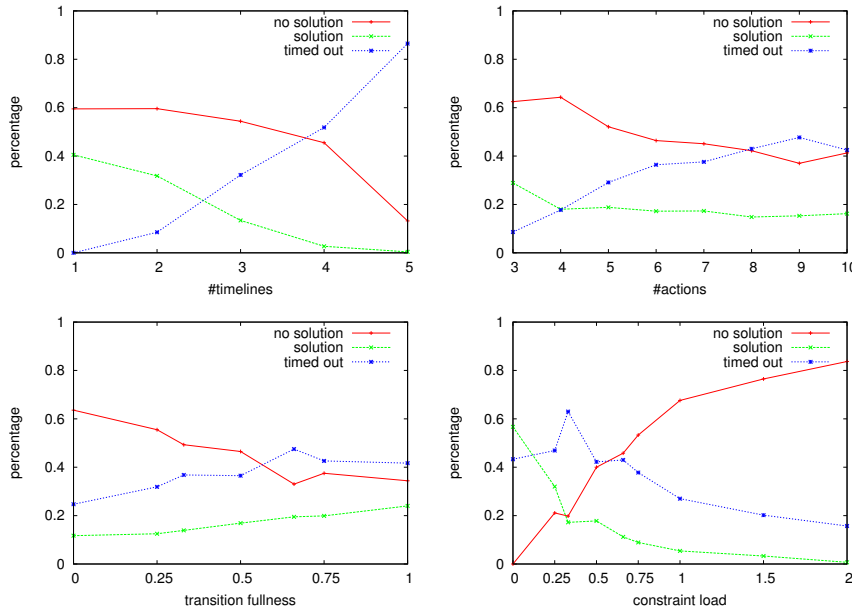


Table 1. Outcome breakdown

We observe a few natural trends. The number of timeouts increases dramatically with the number of timelines, which is the largest contributor to the complexity of a SAL model. The number of timeouts also increases with the number of actions, but more interestingly, for the number of constraints, it first peaks for an intermediate value, before dipping. We attribute this to the fact that increasing the number of constraints is likely to reduce the chances of an existing solution.

In terms of finding a solution, there is a mix of results. As seen above, more timelines and more constraints decrease the possibility of a solution. The number of actions has a small effect, while the number of transitions seems to favor the existence of a solution. This can be attributed to the fact that more edges in the transition graph would allow the reach of a goal state by “bypassing” actions that are tied into unfeasible constraints.

In general, the generated models seem to be more likely to lack a solution rather than have one. This is probably due to the “random” (that is often meaningless) nature of the models.

Runtimes. Figure 2 shows the average runtime for six combinations of parameters. The trends are also mixed. While it is obvious that the runtime will grow with the increase in the number of timelines and actions, the number of transitions seems to have a negligible effect on the runtime. Also, the number of constraints produces a peak in the middle and decreases for larger values.

The dependency on the number of timelines can be illustrated by the total runtime of the script for the approximately 800 models for each value of T . It took less than a day (23 hours) to finish the models with $T = 1$, more than two days (51 hours) for $T = 2$, six and a half days (156 hours) for $T = 3$, ten days (240 hours) for $T = 4$, and the case $T = 5$ is still running.

We also computed the averages in subcategories, corresponding to the existence of a solution or not. Both due to lack of space and also to the fact that the comparison is unfair to the case when a solution exists (given that the model checker spends more time constructing the counterexample), we left those graphs out of this paper. The profile of the graphs is largely similar to the overall averages, but is roughly scaled (down for no solution, up for an existing solution) by a constant factor.

7 Conclusions

We are just making baby steps in this area. Traditional symbolic model checking technology is not mature enough to handle complex applications. Yet, with the help of more advanced techniques (timeout automata and other deductive approaches, such as SMT solvers [6]), more progress can be made.

In general, we believe that there is a clear role for formal methods in designing planning languages. While the powerful heuristics of the AI software are more suited to efficiently find a solution, exhaustive techniques, such as model checking, are obviously the only alternative to prove the lack of a solution. Moreover, in safety-critical applications, eliminating ambiguities in the specification

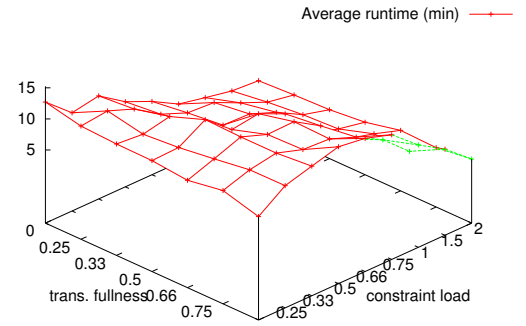
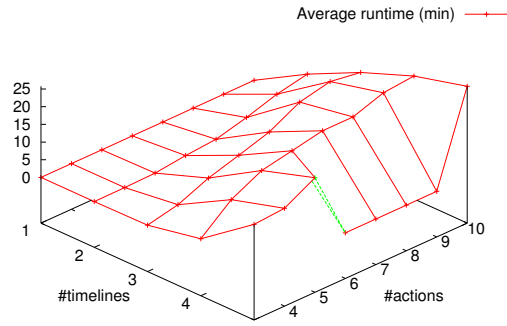
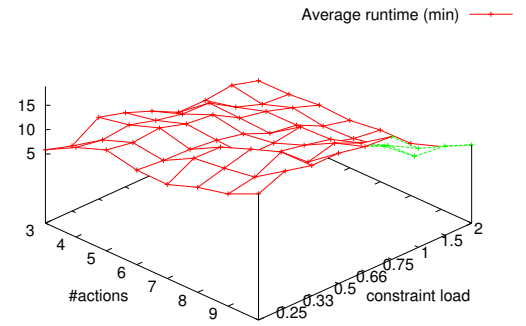
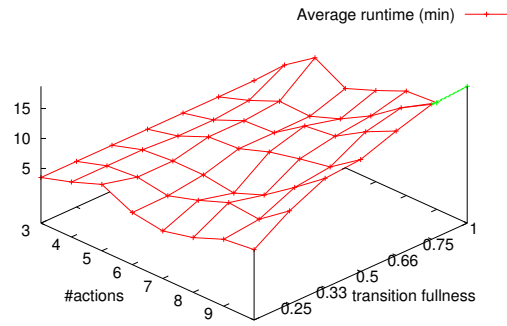
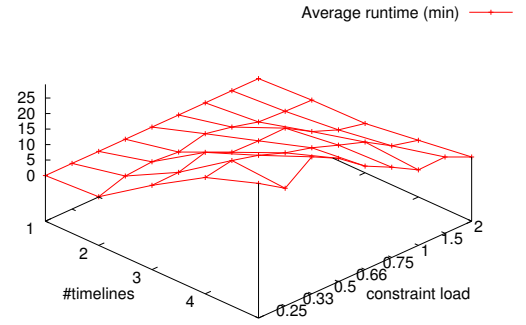
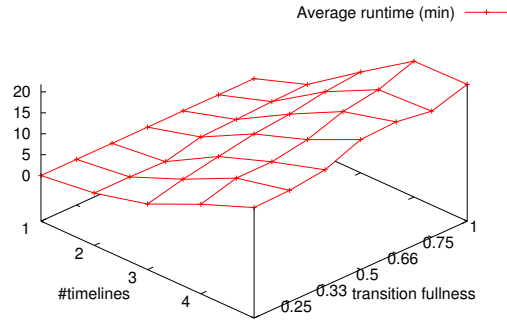


Table 2. Average runtimes

language is a strong requirement. Our comparative study with EUROPA2 has provided valuable insight and feedback to the designers to help them make the planning language more robust and safe.

References

- [1] James F. Allen and George Ferguson. Actions and Events in Interval Temporal Logic. Technical Report TR521, University of Rochester, 1994.
- [2] Tania Bedrax-Weiss, Conor McGann, Andrew Bachmann, Will Edington, and Michael Iatauro. EUROPA2: User and Contributor Guide. Technical report, NASA Ames Research Center, Moffett Field, CA, Feb 2005.
- [3] Rick W. Butler and César A. Muñoz. An Abstract Plan Preparation Language. Report NASA/TM-2006-214518, NASA Langley, NASA LaRC, Hampton VA 23681-2199, USA, 2006.
- [4] Rick W. Butler, Radu I. Siminiceanu, and César A. Muñoz. The ANMLite language and logic for specifying planning problems. Report TM-2007-215088, NASA Langley, Hampton VA 23681-2199, USA, November 2007.
- [5] Alessandro Cimatti, Charles Pecheur, and Roberto Cavada. Formal Verification of Diagnosability via Symbolic Model Checking. In *IJCAI*, pages 363–369, 2003.
- [6] Leonardo de Moura and Bruno Dutertre. Yices 1.0: An Efficient SMT Solver. Technical report, SRI International, 2006. SMCOMP’06, <http://yices.csl.sri.com>.
- [7] Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL Language Manual. Technical Report SRI-CSL-01-02, CSL Technical Report, 2003. <http://sal.csl.sri.com/documentation.shtml>.
- [8] Doron Drusinsky and Garth Watney. Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine. In *28th IEEE/NASA Software Engineering Workshop*, page 127, 2003.
- [9] Martin S. Feather and Ben Smith. Automatic Generation of Test Oracles – From Pilot Studies to Application. *Automated Software Engineering*, 8(1):31–61, January 2001.
- [10] Jeremy Frank and Ari Jonsson. Constraint-based Attribute and Interval Planning. *Journal of Constraints*, 8:339–364, 2003.
- [11] Alessio Lomuscio, Charles Pecheur, and Franco Raimondi. Automatic Verification of Knowledge and Time with NuSMV. In *IJCAI*, pages 1384–1389, 2007.
- [12] Drew McDermott and AIPS’98 IPC Committee. PDDL – the Planning Domain Definition Language. Technical report, Yale University, 1998. Available at www.cs.yale.edu/homes/dvm, 1998.
- [13] Sam Owre and Natarajan Shankar. Formal Analysis Methods for Spacecraft Autonomy, Final Report. Technical Report SRI-17625, SRI International, 2007.
- [14] Charles Pecheur and Franco Raimondi. Symbolic Model Checking of Logics with Actions. In *MoChArt 2006*, pages 113–128, 2006.
- [15] Hossein M. Sheini, Bart Peintner, Karem A. Sakallah, and Martha E. Pollack. On Solving Soft Temporal Constraints Using SAT Techniques. In *Principles and Practice of Constraint Programming 11th International Conference, Sitges, Spain*, pages 607–621, October 2005.
- [16] David E. Smith, Jeremy Frank, and Ari K. Jonsson. Bridging the Gap between Planning and Scheduling. *The Knowledge Engineering Review*, 15(1):113–128, 2000.
- [17] David E. Smith, Jeremy Frank, and Conor McGann. The ANML Language. Technical report, NASA Ames, unpublished report, 2006.

A The ANMLite Syntax

A.1 Timeline declarations

```
<anml_def>      ::= PLAN <identifier>
                  ( <type_decl> | <timeline_decl> | <constraints_decl> |
                    <vars_decl> )*
                  [ <inits_decl> ]
                  [ <goals_decl> ]
                  END <identifier>

<type_decl>      ::= TYPE ( <simple_type_decl> | <compound_type_decl> )

<simple_type_decl> ::= <identifier> = <type>

<compound_type_decl> ::= <identifier> <parameters> [ = <type> ]

<type>           ::= <basic_type> | <enumeration> | <interval> | <defined_type>

<basic_type>     ::= INT | FLOAT | BOOL | STRING |

<enumeration>    ::= [ <identifiers> ]

<identifiers>    ::= <identifier> ( , <identifier> )*

<identifier>     ::= <ID>

<interval>       ::= [ <add_expression_or_nil> , <add_expression_or_nil> ]

<add_expression_or_nil> ::= <additive_expression> | <nil>

<defined_type>   ::= <identifier> <arguments>

<arguments>      ::= [ <strict_arguments> ]

<strict_arguments> ::= "(" <expression_or_nil> ( , <expression_or_nil> )* ")"

<expression_or_nil> ::= <expression> | <nil>

<nil>            ::= "_"

<parameters>     ::= [ strict_parameters() ]

<strict_parameters> ::= "(" <parameter> ( ; <parameter> )* ")"

<parameter>      ::= <identifiers> ":" <type>

<timeline_decls> ::= TIMELINE <identifier> <parameters>
                    <actions_decl>
                    <transition_decl>
                    END <identifier>

<actions_decl>   ::= ACTIONS ( action_decl() )+

<action_decl>    ::= <identifier> <parameters> <duration_decl> [ action_body() ]

<duration_decl>  ::= [ ":" <interval> ]

<transitions_decl> ::= [ TRANSITIONS ( <transition_decl> )+ ]

<transition_decl> ::= <action>
                    ( ( -> <action> )+ ["-|"] |
                      -> * "\" <action>
                    )

<action>         ::= <simple_action> |
```



```

( "(" <simple_action> ( "|" <simple_action> )+ ")" )

<simple_action> ::= <qualified_id> <arguments>

<start_end_var> ::= [ NEXT ] <identifier> <start_end>

<start_end> ::= ".start" | ".end"

<qualified_id> ::= <ID> ( .<ID> )

<expression> ::= .....

```

A.2 Constraints

```

<constraints_decl> ::= CONSTRAINTS ( <constraint_decl> )+

<constraint_decl> ::= ( <at_formula> | <bool_formula> )

<at_formula> ::= [ <at_expression> ] <timepoint> <rel_op>
               <timepoint> [ <plusinteger> ]

<timepoint> ::= ( <start_end_term> | <integer> )

<start_end_term> ::= ( next ) <ID> <start_end> ( <add_op> <integer> )

<add_op> ::= + | -

<bool_formula> ::= ( <simp_bool_formula> ("&&" | "||" | "->")+
                   <simp_bool_formula>
                   | "!" <simp_bool_formula>
                   | <simp_bool_formula>
                   )

<simp_bool_formula> ::= <state_var> "==" <state> |
                     <state_var> "!=" <state>

<bin_logic_op> ::= "&&" | "||" | "->"

<at_expression> ::= at <qualified_id> ( <start_end> )
(<strict_arguments>) :
<state_var> ::= <ID>
<state> ::= <ID>

<inits_decl> ::= INITIAL_STATE ( init_decl )+
<goals_decl> ::= GOALS ( goal_decl )+
<goal_decl> ::= <action>
<vars_decl> ::= VARIABLES ( var_decl )+
<var_decl> ::= <identifiers> <COLON> <type> ( = <integer> )
<init_decl> ::= |-> <action>

```

A.3 Condition and Effect Statements

```

<action_body> ::= "{" [ condition() ] [ effect() ] "}"

<condition> ::= "condition:" <expression>
<effect> ::= "effect:" <identifier> "!=" <expression>

```