Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

SMT-Based Reasoning and Planning in TAL

by

Magnus Hallin

LITH-IDA-EX-A--10/032--SE

2010-06-12
Final thesis

SMT-Based Reasoning and Planning in TAL

by

Magnus Hallin

LITH-IDA-EX-A--10/032--SE

Supervisor: Martin Magnusson
Dept. of Computer and Information Science
at Linköping University

Examiner: Patrick Doherty
Dept. of Computer and Information Science
at Linköping University
Abstract

Automated planning as a satisfiability problem is a method developed in the early nineties. It has some known disadvantages, such as its inefficient encoding of numbers. The field of Satisfiability Modulo Theories tries to connect already established solvers for e.g. linear constraints into SAT-solvers in order to make reasoning about numerical values more efficient.

This thesis combines planning as satisfiability and SMT to perform efficient reasoning about actions that occupy realistic time in Temporal Action Logic, a formalism developed at Linköping University for reasoning about action and change.
Acknowledgements

I would like to thank everyone who has helped me complete this thesis – my examiner Patrick Doherty, my supervisor Martin Magnusson, my opponent Mikael Modin, and proofreader Frida Schlaug.

In addition to reading, commenting and being supportive of my work, Martin, together with Jonas Kvarnström, assisted me with the proofs in chapter 4. Without their great help, I would not have been able to complete that chapter.
## Contents

1 Introduction 1
   1.1 Objective ........................................... 2
   1.2 Thesis Outline ........................................ 2

I Theoretical Foundations 3

2 Satisfiability Modulo Theories 5
   2.1 Satisfiability ........................................... 5
   2.2 Theories ................................................ 7
   2.3 Encoding Methods ...................................... 9

3 Temporal Action Logic 11
   3.1 Introduction ........................................... 11
   3.2 Reasoning in TAL ........................................ 13

II SMT In Practice 15

4 From TAL to SMT 17
   4.1 Compiling TAL to Propositional Logic .................. 17
   4.2 SAT-Based Reasoning in TAL ............................ 20
   4.3 The Significant Time Point Concept .................... 20
   4.4 SMT-Based Reasoning in TAL ............................ 25
5 Survey of Solvers 27
  5.1 Criteria ........................................... 27
  5.2 Method ........................................... 28
  5.3 Results .......................................... 30
  5.4 Conclusions .................................... 31

6 A TAL Grounder 37
  6.1 Architecture ..................................... 37
  6.2 Input Language ................................... 38
  6.3 Implementation .................................. 39
  6.4 Summary ......................................... 40

III Result 43

7 Automated Planning with SMT 45
  7.1 Introduction ..................................... 45
  7.2 Optimizations .................................... 46
  7.3 API ............................................. 49
  7.4 Summary ......................................... 56

8 A Game Application 57
  8.1 Introduction ..................................... 57
  8.2 Gameplay ......................................... 58
  8.3 Advantages and Disadvantages .................. 63
  8.4 Performance ...................................... 64
  8.5 Summary ......................................... 65

9 Discussion 67
  9.1 Future Work ...................................... 68

A Scenarios Used in the Survey 69
  A.1 TAL Axioms for the Grounder .................. 69
  A.2 The AIPS-00 Logistics Domain ................. 70
  A.3 The Timed Logistics Domain ................... 72
  A.4 The Russian Hijack Scenario ................... 74

Bibliography 79
Chapter 1

Introduction

The problem of finding a model for a set of propositional formulas – the SAT problem – was formulated over 40 years ago. Thanks to the general nature of logic, this problem has many applications both in industry and in academia. The problem is NP-complete, but much work is invested in making SAT solvers more efficient – the latest solvers can solve surprisingly large problems very fast.

But propositional logic has its weaknesses. In particular, numeric values are not encoded very efficiently into propositional logic. Only fixed-point or integer arithmetic can be used and the problem size is dependent on the size of the numbers involved, which is very limiting. This limitation spawned a new research field: Satisfiability Modulo Theories (SMT) (Barrett et al., 2008).

SMT connects theory-specific solvers, such as a linear constraint solver, into existing SAT-solvers. It can therefore lift the computational complexity away from the size of the numbers and to the number of constraints instead. SMT is not limited to reasoning about numbers – there are many other theories available as well. The field of SMT has gained much from the development of SAT solvers, and many of the existing SMT solvers have built upon the award-winning SAT solver MiniSAT (Eén and Sörensson, 2004) (Bruttomesso et al., 2008; Barrett and Tinelli, 2007).

Another research field which gained from the development on SAT solvers is the field of automated planning. Kautz and Selman (1992) demonstrated how to encode planning problems into propositional logic and managed to beat all traditional planners in the optimal deterministic track in the International Planning Competition 2004.
But as the planning problems become more complex, especially when they contain durative actions which span multiple time points and numeric resources with large quantities, planning as satisfiability falls short. Here again, SMT has proven beneficial. Shin and Davis (2005) describes the TM-LPSAT planner which combines a SAT solver with the linear constraint solver Cassowary. This planner can reason about durative actions with arbitrary length, as well as numeric and even continuous resources.

1.1 Objective

At Linköping University, Temporal Action Logic, a formalism for dealing with action and change was developed (Doherty and Kvarnström, 2008). It is a powerful logic that can be used to describe actions and their effect on the world, as well as domain constraints and dependencies.

With this background, the question arises: can you take advantage of the development of SMT solvers to perform efficient reasoning and planning in TAL? Exploring this possibility is the main objective of this thesis.

The result is a component for applications that need AI. I will exemplify its use in a computer game application towards the end of this thesis, but any application that would need planning in a predefined domain can use the API defined in chapter 7.

1.2 Thesis Outline

The thesis is divided into three parts. In the first part, Theoretical Foundations, I will briefly present the theory behind SMT and TAL.

In the second part, I will demonstrate how you could integrate TAL and SMT in practice, and also present a survey of SMT solvers with their strengths and weaknesses compared.

The last part will demonstrate how to do planning, both in theory and from within an application using an API. Finally, I will conclude the report by discussing the results.
Part I

Theoretical Foundations
Satisfiability Modulo Theories

Satisfiability Modulo Theories is a research field which has its roots in the late seventies (Barrett et al., 2008), but has in recent years gained interest from the software engineering community. SMT can be used, among other things, to prove the correctness of programs and models, or to generate unit tests (Srivastava et al., 2009).

In this chapter, I will provide a background on SMT; what it is and what you can do with it. Because of the wide scope of SMT research, I will only present the theory of linear programming. Barrett et al. (2008) has a more comprehensive survey of theories.

SMT consists of three parts: a satisfiability problem, one or more theories and an interface between them. The following sections will deal with these parts, respectively. To conclude the chapter, I will talk about how a typical solver works.

2.1 Satisfiability

The problem of determining boolean satisfiability, the SAT problem, is the problem of finding an assignment of truth values to variables in order to make a set of propositional formulas true. For example, \( \{A, B, \neg C\} \) is a model for the set \( \{B \rightarrow A, B \lor C, A \rightarrow \neg C\} \). A set of formulas can of course have multiple models; \( \{\neg A, \neg B, C\} \) is another model for the example. The problem formulation has been around for almost 40 years and was the first problem to be proven NP-complete (Cook, 1971).
Annual competitions\(^1\) are held, where SAT-solvers compete on both real-world and randomized problems. Since the competitions started in 2002, there has been a lot of progress. New solvers are very fast for real-world problems.

The SMT community has benefited from this progress by extending various SAT-solvers and thereby gaining good performance on the S in SMT “for free” (Barrett et al., 2008).

**The DPLL algorithm**

Almost all current solvers use the DPLL algorithm (Russell and Norvig, 2003) with different extensions. The DPLL algorithm began as the Davis-Putnam algorithm, constructed in 1960, and became the DPLL algorithm after Davis, Logemann and Loveland extended it 1962 (Russell and Norvig, 2003). Modern implementations of the DPLL algorithm include other extensions such as clause learning.

To better illustrate the principles of DPLL, I present the algorithm on the adjacent page. The algorithm uses three external functions, namely:

- **find-pure-symbol** A pure symbol is a symbol which only appears as either negative or non-negative in all clauses and therefore must be assigned to that value.

- **find-unit-clause** A unit clause is a clause with only one literal. Therefore, that literal must be satisfied.

- **choose-branch-literal** If no pure symbol or unit clause is found, the algorithm must pick a variable and try both its positive and negative assignment. This function picks a variable to branch on, and can vary in complexity. The simplest implementations choose the first unassigned variable, but most solvers use some kind of heuristic.

The DPLL algorithm is very easy to extend, thanks to its simplicity – this will be exemplified in section 2.3. But first, let’s talk about theories.

\(^1\)http://www.satcompetition.org
Algorithm 1 The DPLL algorithm, as presented in Russell and Norvig (2003)

**Require:** clauses ← A set of disjunctions
   symbols ← A list of symbols used in the formula
   model ← The current model, initially []
1: if every clause in clauses are true in model then
2: return true
3: else if some clause in clauses is false in model then
4: return false
5: end if
6: P, value ← find-pure-symbol (symbols, clauses, model )
7: if P is non-null then
8: return DPLL (clauses, symbols − P, extend (P, value, model ))
9: end if
10: P, value ← find-unit-clause (symbols, clauses, model )
11: if P is non-null then
12: return DPLL (clauses, symbols − P, extend (P, value, model ))
13: end if
14: P ← choose-branch-literal (symbols, clauses, model )
15: return DPLL (clauses, symbols − P, extend (P, true, model )) or
   DPLL (clauses, symbols − P, extend (P, false, model ))

2.2 Theories

Plug-in theories make up the core of SMT. The idea is that you have a domain specific theory \( \mathcal{T} \), which you want to reason about in logic. This \( \mathcal{T} \) can be almost anything decidable; there are currently theories of finite and infinite trees, lists, bitvectors, arrays and linear arithmetic (Barrett et al., 2008). SMT takes advantage of the fact that some of them already have efficient solvers in order to reason more efficiently

The only real criterion for a theory to be used in SMT is that it has a function (called a \( \mathcal{T} \)-solver) which takes a set of \( \mathcal{T} \)-literals and determines if it is satisfiable with respect to \( \mathcal{T} \).

However, there are certain properties which are important in practical use (Barrett et al., 2008), among others:

**Model generation** Crucial in real world usage, since you otherwise don’t know which values variables should have.
Conflict set generation  When the \( T \)-solver returns unsat, it should also return the (preferably minimal) set of conflicting literals.

Incrementality and backtrackability  For maximum efficiency, the solver should not have to redo all computation in each deduction step.

Theory propagation  Given a \( T \)-literal \( \mu \), a \( T \)-solver should deduce a new set of literals, \( \Gamma \), which are entailed by \( \mu \), and insert the formula \( \forall \gamma \in \Gamma (\mu \rightarrow \gamma) \). This can be done with a varying degree of completeness. As Dutertre and Moura (2006) notes, neither no propagation nor full propagation is performant; a simple heuristic is the best choice.

Linear programming

One such theory that can be used is linear programming. Linear programming is a part of mathematical optimization theory in which you optimize a linear objective function while satisfying a set of linear constraints (Holmberg, 2003). A linear constraint is an inequality on the form:

\[
a_0x_0 + \ldots + a_nx_n R c
\]

where \( R \in \{=, \leq, \geq\} \), \( \bar{a} \) and \( c \) are constants and \( \bar{x} \) are variables. In a linear programming problem, you also have an objective function \( o_0x_0 + \ldots + o_nx_n \) which you either maximize or minimize, while maintaining the set of (in-)equalities. For example, consider

\[
\begin{align*}
\text{max} \quad & z = 4x_1 + 3x_2 \\
\text{w.r.t.} \quad & 2x_1 + 3x_2 \leq 30 \\
& x_1 \leq 6 \\
& 6x_1 + 4x_2 \leq 50 \\
& x_1 \geq 0 \\
& x_2 \geq 0
\end{align*}
\]

with the optimal solution \( x_1 = 3, x_2 = 8 \) and \( z = 36 \). If the objective function is constant, a linear programming solver will simply determine whether there are feasible solutions to the set of constraints.

Linear programming also shares some properties with propositional logic, such as that they both are convex. If you have an infeasible set of linear constraints, you can
not add more constraints to get a feasible set – likewise, if you have an unsatisfiable set of propositional clauses, you can not add more clauses to get a satisfiable set. This makes it very suitable as a theory for SMT.

The Simplex algorithm is the most common way of solving linear programming. It is centered around a tableau which it iteratively pivots until the optimal solution is found or no valid solutions can be found. If the objective function is constant, Simplex will pivot until a valid solution is found or abort if no solutions can be found.

While Simplex has a worst case complexity that is exponential in number of variables, it outperforms almost all polynomial LP-algorithms in most real world scenarios. Also, the algorithm has three of the four properties described earlier. The model and conflict sets are by-products of the algorithm and need only be read from the Simplex tableau. Adding or removing a constraint or variable is simply adding or removing a row or column in the tableau and then possibly performing a few pivots in order to maintain optimality.

Theory propagation can be done incompletely but efficiently when adding a constraint \( \bar{x} \leq c \) by looking for constraints of the form \( \bar{x} \leq c' \) where \( c' \geq c \) and similarly for \( \geq \) and \( = \).

Simplex is therefore very suitable as a \( T \)-solver in SMT for linear constraints. Without an objective function, it will only perform operations on the tableau if the current solution becomes invalid.

### 2.3 Encoding Methods

There are two different approaches that integrate the satisfiability problem and the theories; eager and lazy encoding.

Using eager encoding, all theories are encoded as a SAT problem before solving. Therefore, the only thing needed except the encoder is a simple SAT solver. The eager approach has the drawback of generating intractably large SAT problem instances when used on all but the simplest problems.

On the other hand, lazy encoding generates separate problems for all theories, and relies on hooks in a SAT solver to query the specific theories for satisfiability. This means there are certain literals in the propositional problem, called \( T \)-literals, that correspond to e.g. a linear constraint being active. When the SAT solver assigns such
a variable, it also queries the $\mathcal{T}$-solver which adds the corresponding constraint. If the set of constraints become inconsistent, the $\mathcal{T}$-solver reports that the $\mathcal{T}$-literal can not assume that value, and the SAT-solver backtracks.

Lazy encoding is by far the most common approach, and there exist many solvers for many different theories.
Temporal Action Logic (Doherty and Kvarnström, 2008) is a formalism which has been developed since the early nineties at Linköping University. TAL provides tools to deal with the frame problem\(^1\) and other problems. TAL introduces a high level notation for specifying actions and change.

In the following sections, I will summarize the relevant theory that is used in the rest of this thesis.

### 3.1 Introduction

TAL contains four basic sorts; timepoints, actions and features that assume values (Doherty and Kvarnström, 2008). These four types are used in the definition of the following first order predicates:

- *Holds(timepoint, feature, value)* which denotes that a feature assumes the value at a certain timepoint.
- *Occurs(timepoint, timepoint, action)* which denotes that an action is occurring during the two specified timepoints.
- *Occlude(timepoint, feature)* which permits a feature to change value at a time-point.

---

\(^1\) The **frame problem** is the problem of how to represent dynamic change in logic without specifying everything that is not changed by an action.
Of these, occlusion can be the hardest to understand. The concept is really simple if you think of it as “permission to change”. *If Occlude is false, the fluent will stay the same to the next timepoint.* By minimizing the number of occlusions, as many features as possible will stay the same. This is the purpose of the circumscription policy, described below.

### The High-Level Language

TAL has a high-level language denoted $\mathcal{L}(\text{ND})$ for Language of Narrative Descriptions (Doherty and Kvarnström, 2008). $\mathcal{L}(\text{ND})$ is an abstract macro language that allows you to write narratives more easily, but is translated into $\mathcal{L}(\text{FL})$, which is an ordinary first order logic, where standard reasoning tools can be used (Kvarnström, 2001). The reason it is called a Narrative Language becomes apparent when viewing the syntax:

\[
\begin{align*}
\text{obs} & \ [0] \ location \ (\text{agent}) \ \overset{\text{def}}{=} \ living\text{-}room \quad (3.1) \\
\text{occ} & \ [3, 8] \ move \ (\text{agent}, \ outdoors) \quad (3.2) \\
\text{acs} & \ [t_1, t_2] \ move \ (a, l) \ \leadsto \ R \ ((t_1, t_2) \ location \ (a) \ \overset{\text{def}}{=} \ l) \quad (3.3) \\
\text{per} & \ \forall \ t \ Per \ (t, \ location \ (\text{agent})) \quad (3.4)
\end{align*}
\]

The symbol $\overset{\text{def}}{=}$ denotes *fluent equality*, so (3.1) states that the agent is in the living room at time point 0. The numbers in brackets preceding the formula is the temporal context, so (3.2) specifies that the agent moves outdoors between time points 3 and 8.

Formula (3.3) is an *action specification* which through $\leadsto$ and $R$ states that the effect of move is that the location of the agent will change. The $R$ macro will break the persistence by occluding the fluent at the specified time points in order to allow the location to change.

The last formula (3.4) is a *persistence axiom* which states that the value of location (agent) will persist between time points if not occluded.

$\mathcal{L}(\text{ND})$ includes many more macros than $R$ and $Per$, Doherty and Kvarnström (2008) has a more comprehensive list with explanations of how to translate $\mathcal{L}(\text{ND})$ into first order logic.
3.2 Reasoning in TAL

The letters before each statement denotes the type of the formula. There are six formula types in TAL (Doherty and Kvarnström, 2008):

- **acs**, action specifications,
- **dep**, dependency constraints – dependencies between fluents,
- **dom**, domain constraints – invariant information in the domain,
- **obs**, fluent observations,
- **occ**, action occurrences, and
- **per**, persistence statements.

Their main purpose is to distinguish formulas of different types when performing circumscription, as described below. In the next chapter, they will be used to define the subset of TAL on which SMT-based reasoning is applied to.

**Circumscription Policy**

Predicate circumscription on a set of formulas is a method of minimizing the extension of a certain predicate. The closed world assumption, used in some logic programming environments, is a special case of circumscription (McCarthy, 1986).

Circumscription must be performed on both the *Occurs* predicate and the *Occlude* predicate. *Occurs* must be circumscribed to prevent spurious actions occurring in a narrative. *Occlude* must be circumscribed to minimize the number of potential value changes of features in a narrative.

The circumscription of *Occurs* is done on the set of action occurrence formulas, and *Occlude* on the set of dependency constraints and action specifications.

By enforcing certain restrictions on the \( \mathcal{L}(ND) \) formulas, *Occlude* and *Occurs* only appear positively in the relevant parts of the narrative. Circumscription is then equivalent to predicate completion (Lifschitz, 1991), which is straightforward to compute.

**3.2 Reasoning in TAL**

Reasoning in TAL can be done either by hand, using a proof system of choice, or by automated tools such as VITAL (Kvarnström, 2001). Magnusson (2007) shows some proofs using natural deduction in \( \mathcal{L}(FL) \), as well as an automated Prolog-based TAL reasoner.
In chapter 4, I will demonstrate how to translate narratives into propositional logic and linear constraints, and do both reasoning and planning with a SMT solver.

Planning is an obvious application for a formalism dealing with time. Thus, TALplanner was created. TALplanner is a forward chaining state space search planner which starts at an initial state and applies all valid actions until a goal is reached. To guide the search, the user needs to write “control rules” – rules that are checked on every state expansion and prunes the subtree if invalid.
Part II

SMT In Practice
From TAL to SMT

SMT reasoning requires the input to be in propositional logic with linear constraints. This chapter will first show how to reason in TAL using plain propositional logic (i.e. SAT) and then describe a translation of the timepoints to linear constraints and the application of SMT.

4.1 Compiling TAL to Propositional Logic

Doherty and Kvarnström (2008) describe the relation between a $L$(ND) narrative and the 1:st order logic $L$(FL) theory, depicted in figure 4.1. The goal of this section is to define a translation from $L$(ND) to propositional logic, which is equivalent to the one between $L$(ND) and $L$(FL), given certain restrictions.

Definition 1. The propositionalisation $Ground(N', t_{max})$ of a $L$(ND) theory $N'$, given a natural number $t_{max}$ that places an upper bound on the time point domain, is defined by the following steps:

- Translate $N'$ into $\Gamma$ using the $Trans$ function.
- Apply predicate completion on $Occlude$ in $\Gamma_{acs} \land \Gamma_{dep}$ and $Occurs$ in $\Gamma_{occ}$.
- Add TAL’s unique values axioms to $\Gamma$.
- Ground all formulas in $\Gamma$ with respect to the fluent and value sorts in $N'$. 
As these concepts might be unfamiliar to the reader, I will describe these briefly before proving equivalence.

**The Trans Function**

*Trans* is a purely syntactical translation of an $\mathcal{L}(\text{ND})$ narrative into $\mathcal{L}(\text{FL})$ formulas. It is thoroughly defined in Doherty and Kvarnström (2008), but I will give a small example here. The narrative on page 12 is translated, line by line, as follows:

\[
\begin{align*}
\text{Holds}(0, \text{location(agent), living-room}) \quad (4.1) \\
\text{Occurs}(3, 8, \text{move(agent, outdoors)}) \quad (4.2) \\
\forall t_1, t_2, a, l [\text{Occurs}(t_1, t_2, \text{move}(a, l)) \rightarrow \forall t [t > t_1 \land t \leq t_2 \rightarrow \text{Oclude}(t, \text{location}(a))] \land \\
\text{Holds}(t_2, \text{location}(a), l) \quad (4.3) \\
\forall t [\neg \text{Oclude}(t + 1, \text{location(agent)}) \rightarrow \\
\forall v [\text{Holds}(t + 1, \text{location(agent)}, v) \leftrightarrow \\
\text{Holds}(t, \text{location(agent)}, v)] ] \quad (4.4)
\end{align*}
\]
4.1. Compiling TAL to Propositional Logic

Predicate Completion

Predicate completion, sometimes called Clark completion, is a method of completing the unknown information in a knowledge base by assuming that only what is specified positively is true (Clark, 1978).

To complete a predicate $P$ in a knowledge base, you first gather all reasons for $P$ being true, i.e. all formulas on the form $F \rightarrow P$. Form a disjunction between all these reasons: $F_1 \lor \ldots \lor F_n$. Finally, add the formula $P \rightarrow (F_1 \lor \ldots \lor F_n)$ and the knowledge base is completed.

In TAL, two completions are necessary: $Oclude$ is completed in the action specifications and dependency constraints (the sets $\Gamma_{acs}$ and $\Gamma_{dep}$) in order to minimize potential change, and $Occurs$ is completed in the action occurrences (the set $\Gamma_{occ}$) in order to avoid spurious actions from occurring.

Unique Values Axioms

TAL contains two unique values axioms: One which states that a fluent can only assume at most one value at a time, and one which states that a fluent must assume at least one value at a time. The result of these two is, obviously, that each fluent assumes exactly one value at each timepoint.

Formally, the axioms are the following:

$$\forall t, f, v_1, v_2 [v_1 \neq v_2 \rightarrow \neg(Holds(t, f, v_1) \land Holds(t, f, v_2))]$$ (4.5)

$$\forall t, f, \exists v \mid Holds(t, f, v)$$ (4.6)

Grounding

Grounding is the process of converting a set of first order logic formulas with a finite domain to propositional logic. This is done by eliminating all quantifiers by expanding universal quantifiers with the conjunction of, and existential quantifiers with the disjunction of, all possible values the variables can assume.

Although the concept is simple, the construction of an efficient and powerful grounder is not. The grounder used in this thesis is described in more detail in chapter 6.
4.2 SAT-Based Reasoning in TAL

The core of automated reasoning in TAL is determining whether \( N \models G \), given a \( \mathcal{L}(\text{ND}) \) theory \( N \) and a proof goal \( G \). I accomplish this by running a complete SAT-solver, such as MiniSat (Eén and Sörensson, 2004), on the grounded instance \( \text{Ground}(N \land \neg G, t) \) iteratively with \( t \in \{1, 2, \ldots , t_{\text{max}}\} \), with a user-specified \( t_{\text{max}} \). This can be shown to be complete for TAL, using the following two theorems.

**Theorem 2.** A \( \mathcal{L}(\text{ND}) \) theory \( N \) is unsatisfiable iff \( \text{Ground}(N, t_{\text{max}}) \) is unsatisfiable for some \( t_{\text{max}} \in \mathbb{N} \).

*Proof.* The circumscription and quantifier elimination steps are equivalent to predicate completion (Doherty and Lukaszewicz, 1994). The foundational axioms consist of unique name axioms and unique value axioms. The former are automatically satisfied by grounding – unique names imply unique values in propositional logic. TAL domains are finite, with the exception of time. By Herbrand’s (1930) theorem, the 1:st order logic theory is unsatisfiable iff its grounding is unsatisfiable for some \( t_{\text{max}} \). Thus, all the modifications to the translation in Definition 1 preserve unsatisfiability. \( \square \)

**Theorem 3.** Given a complete SAT-solver, a \( \mathcal{L}(\text{ND}) \) narrative \( N \) and proof goal \( G \), \( N \models G \) iff the SAT-solver returns \text{unsat} on \( \text{Ground}(N \land \neg G, t_{\text{max}}) \) for some \( t_{\text{max}} \).

*Proof.* By the deduction theorem, \( N \models G \) iff \( N \land \neg G \) is unsatisfiable. By Theorem 2, \( \text{Ground}(N \land \neg G, t_{\text{max}}) \) for some \( t_{\text{max}} \) will preserve unsatisfiability. By soundness and completeness of the SAT-solver, it will return \text{unsat} iff the problem is unsatisfiable. \( \square \)

4.3 The Significant Time Point Concept

As we will see in the survey in chapter 5, the above works well for small values of \( T_{\text{max}} \), but as soon as the numbers grow larger, the SAT-problem becomes infeasibly large. This chapter will introduce a method of reasoning in TAL using SMT, which has the potential to scale up to large time points. To do this, I will introduce the concept of significant time points and clock time points. This concept was initially created by
Shin and Davis (2005) as a part of their work to create a PDDL+ planner that could reason with continuous time and resources.

Here, I will define this concept in a subset of TAL and prove that it is complete for this subset. I will begin by providing some definitions regarding the subset of TAL used, what a significant time point is and how they are related to clock time points.

**Definition 4.** A TAL model includes a sequence of states that assign values to fluents at each time point, e.g.:

<table>
<thead>
<tr>
<th>time point</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>location(\text{agent}_1)</td>
<td>room_1</td>
</tr>
<tr>
<td>location(\text{agent}_2)</td>
<td>room_2</td>
</tr>
</tbody>
</table>

**Definition 5.** A $\mathcal{L}$(FL) TAL theory is a conjunction $\Gamma_{obs} \land \Gamma_{occ} \land \Gamma_{acs} \land \Gamma_{dep} \land \Gamma_{dom} \land \Gamma_{per}$ (Doherty and Kvarnström, 2008). A $\Gamma$-structure is a structure with the signature of $\Gamma$. A TAL$^{SMT}$ theory is a TAL theory conjunction of sentences on the form:

- $\Gamma_{obs}$ $\text{Hold}_s(t, f, v)$ for $t \in \mathbb{N}$, fluent $f$ and value $v$.
- $\Gamma_{acc}$ $\text{Occ}_s(t_1, t_2, a)$ for $t_1, t_2 \in \mathbb{N}$ and action $a$.
- $\Gamma_{acs}$ $\forall_{t_1, t_2}[\text{Occ}_s(t_1, t_2, a) \rightarrow \Phi(t_1, t_2)]$ for action $a$ with $t_1, t_2$ as the only time point occurrences in $\Phi$.
- $\Gamma_{dom}$ $\forall_t \Phi(t)$ with $t$ as the only time point occurrence in $\Phi$.
- $\Gamma_{per}$ $\forall_{t,f,v}[^{\neg}\text{Oclude}_s(t+1, f) \rightarrow (\text{Hold}_s(t, f, v) \iff \text{Hold}_s(t + 1, f, v))]$

**Definition 6.** The clock time points $c$ of a TAL$^{SMT}$ theory $\Gamma$ are all integer time points $[c_1, \ldots, c_n]$ occurring in $\Gamma$. A significant time point $s$ is the index of clock time $c[s]$ in $c$.

**Definition 7.** Let $\Gamma$ be a TAL$^{SMT}$ theory and $m$ be any $\Gamma$-structure. By the significant time point transformation (STPT) we mean, constructing $\Gamma'$ from $\Gamma$ by replacing any clock time points $c[s]$ by its index $s$, and constructing $m'$ from $m$ by removing all states $t$ in $c[s] < t < c[s + 1]$ for which $\text{Hold}_s(t-1, f, v) \leftrightarrow \text{Hold}_s(t, f, v)$ for all $f, v$. 

To exemplify, the state example in Definition 4 would generate:

<table>
<thead>
<tr>
<th>significant time point</th>
<th>0</th>
<th>1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>location(agent₁)</td>
<td>room₁</td>
<td>room₂</td>
<td>...</td>
</tr>
<tr>
<td>location(agent₂)</td>
<td>room₂</td>
<td>room₁</td>
<td>...</td>
</tr>
</tbody>
</table>

Informally, a significant time point is where actions start or end, or observations take place. The example on page 3.1 contains the following clock time points: \( c = [0, 3, 8] \), and it thus contains the following significant time points: \( s = [0, 1, 2] \). Performing the STPT on the narrative produces the following:

\[
\begin{align*}
[0] \text{location (agent)} & \triangleq \text{living-room} \\
[1, 2] \text{move (agent, outdoors)} \\
[t₁, t₂] \text{move (a, l)} & \leadsto R((t₁, t₂) \ \text{location (a)} \triangleq \text{l}) \\
\forall t \ \text{Per} (t, \text{location (agent)})
\end{align*}
\]

Note that while the macro \( R((t₁, t₂) \ \text{location(a)} \triangleq \text{l}) \) refers to other time points than \( t₁ \) and \( t₂ \), it does not refer to any specific time point other than \( t₁ \) or \( t₂ \) – it rather refers to all time points between them. This is still accepted.

Using these definitions, I will prove that a \( \text{TAL}^{\text{SMT}} \) theory is equivalent to its STPT. First, we will need some lemmas to simplify the theorems.

**Lemma 8.** For any \( \text{TAL}^{\text{SMT}} \) theory \( \Gamma \),

\[
\text{CIRC}[\Gamma_{\text{acs}}; \text{Occlude}] \land \text{CIRC}[\Gamma_{\text{occ}}; \text{Occurs}] \land \Gamma_{\text{obs}} \land \Gamma_{\text{dom}} \models \text{Occlude}(t,f)
\]

iff \( t \) is a clock time point.

**Proof.** \( \text{Occlude} \) occurs only in \( \Gamma_{\text{acs}} \) and \( \Gamma_{\text{per}} \). If no \( \Gamma_{\text{acs}} \models \text{Occlude}(t,f) \), then circumscriptio is free to remove \( t \) from the extension. Otherwise, it must be the case that \( \text{Occurs}(t', t, a) \) for some \( t', a \). Since all \( \text{Occurs} \) time points in a \( \text{TAL}^{\text{SMT}} \) theory are clock time points, so is \( t \).

**Lemma 9.** For any \( \text{TAL}^{\text{SMT}} \) theory \( \Gamma \), \( \Gamma \)-structure \( m \), and their STPT \( \Gamma' \) and \( m' \), if \( m \models \Gamma \) then clock time state \( c[s] \) in \( m \) is mapped to its significant time state \( s \) in \( m' \).
4.3. The Significant Time Point Concept

**Proof.** If \( m \models \Gamma \) then \( m \models \Gamma_{\text{per}} \) and fluents must remain unchanged unless occluded. By Lemma 8, this only happens at clock time points in \( m \). Thus for all states \( t \) in \( c[s] < t < c[s+1] \) we have \( \neg \text{Oclude}(t,f) \) and \( \text{Holds}(t,f,v) \leftrightarrow \text{Holds}(t-1,f,v) \). By Definition 7, all states \( t \) were removed in the construction of \( m' \), leaving exactly the significant time states. \( \square \)

**Lemma 10.** For any TALSMT theory \( \Gamma \), \( \Gamma \)-structure \( m \), and their STPT \( \Gamma' \) and \( m' \), if \( m' \models \Gamma' \) then significant time state \( s \) in \( m' \) was mapped from clock time state \( c[s] \) in \( m \).

**Proof.** Suppose \( m' \) includes some non-clock time state \( t \) in \( c[s] < t < c[s+1] \). By Lemma 8, \( \Gamma' \models \neg \text{Oclude}(t,f) \). If \( m' \models \Gamma' \) then \( m' \models \Gamma_{\text{per}} \) and by Modus Ponens \( \text{Holds}(t-1,f,v) \leftrightarrow \text{Holds}(t,f,v) \). But this contradicts \( \text{Holds}(t-1,f,v) \neq \text{Holds}(t,f,v) \), which must be the case or otherwise \( t \) would have been removed when creating \( m' \) by Definition 7. Thus, any state in \( m' \) corresponds to some clock time state. \( \square \)

**Theorem 11.** For any \( \Gamma \)-structure \( m \), TALSMT theory \( \Gamma \) and their STPT \( m' \) and \( \Gamma' \), \( m \models \Gamma \) iff \( m' \models \Gamma' \).

**Proof.** For the \( \Rightarrow \) direction, suppose \( m \models \Gamma \).

\[
\begin{align*}
\text{Each } \text{Holds}(s,f,v) \in \Gamma'_{\text{obs}} & \text{ corresponds to some } \text{Holds}(c[s],f,v) \in \Gamma_{\text{obs}}. \text{ Since we supposed } m \models \text{Holds}(c[s],f,v) \text{ by Lemma 9 we get } m' \models \text{Holds}(s,f,v). \\
\text{Similarly by Lemma 9.} \\
\text{Each action specification } \text{Occurs}(t_1,t_2,a) \rightarrow \Phi(t_1,t_2) & \text{ depends only on time points } t_1, t_2. \text{ Since we supposed } m \models \text{Occurs}(c[s_1],c[s_2],a) \rightarrow \Phi(c[s_1],c[s_2]), \text{ by Lemma 9 we get } m' \models \text{Occurs}(s_1,s_2,a) \rightarrow \Phi(s_1,s_2). \\
\text{Similarly by Lemma 9.} \\
\text{Persistence can only be falsified by non-occluded fluent value changes. But since the construction of } m' \text{ in Definition 7, by removing states, can not add new changes, and existing changes satisfy persistence by the supposition } m \models \Gamma_{\text{per}}, m' \models \Gamma_{\text{per}} \text{ follows.}
\end{align*}
\]
Thus, \( m' \models \Gamma' \) since \( \Gamma' \equiv \Gamma'_{obs} \land \Gamma'_{occ} \land \Gamma'_{acs} \land \Gamma'_{dom} \land \Gamma'_{per} \).

For the \( \Leftarrow \) direction, suppose \( m' \models \Gamma \).

\[
\begin{align*}
    m' = \Gamma_{obs} & \quad \text{Symmetrically by Lemma 10.} \\
    m' = \Gamma_{occ} & \quad \text{Symmetrically by Lemma 10.} \\
    m' = \Gamma_{acs} & \quad \text{Symmetrically by Lemma 10.} \\
    m' = \Gamma_{dom} & \quad \text{Symmetrically by Lemma 10.} \\
    m' = \Gamma_{per} & \quad \text{Persistence follows for any state } t \text{ in } m \text{ that was not removed in the construction of } m' \text{ since we supposed } m' \models \Gamma_{per}. \text{ Since all other states } t, \text{ that were removed from } m \text{ in the construction of } m', \text{ also satisfy persistence by Definition 7, } m \models \Gamma_{per} \text{ follows.}
\end{align*}
\]

Thus, \( m \models \Gamma \) since \( \Gamma \equiv \Gamma_{obs} \land \Gamma_{occ} \land \Gamma_{acs} \land \Gamma_{dom} \land \Gamma_{per} \). \( \square \)

**Theorem 12.** For any TAL<sup>SMT</sup> theory \( \Gamma \), observation goal \( G \) and their STP transformed \( \Gamma' \) and \( G' \), \( \Gamma \models G \iff \Gamma' \models G' \).

**Proof.** By Theorem 11, \( \Gamma \land \neg G \) has a model iff \( \Gamma' \land \neg G' \) has a model. Stated equivalently, \( \Gamma \land \neg G \) is unsat iff \( \Gamma' \land \neg G' \) is unsat. Using a refutation-complete proof system, \( \Gamma \models G \iff \Gamma' \models G' \). \( \square \)

Now that we have constructed a logically sound method of removing unused time points from a narrative, one can get the original time points back from a model generated by an SMT solver by looking up the clock time point \( c[s] \) from a significant time point \( s \). The STP-transformed example in Definition 7 would become:

<table>
<thead>
<tr>
<th>clock time point</th>
<th>( 0 )</th>
<th>( 2 )</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>significant time point</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>location(agent(_1))</td>
<td>( \text{room}_1 )</td>
<td>( \text{room}_2 )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>location(agent(_2))</td>
<td>( \text{room}_2 )</td>
<td>( \text{room}_1 )</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

What we have created here is a mapping between clock time points and significant time points. The computational complexity of determining satisfiability of a TAL<sup>SMT</sup> theory is a function of the number of significant time points, which means that we can create actions of arbitrary duration yet only receive a performance penalty for the number of action occurrences.
Multiple actions and observations can, of course, take place on the same significant time point. For example, the following narrative only contains six clock time points: [0, 1, 7, 9, 26, 30].

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>location(agent₁) ← room₁ ∧ location(agent₂) ← room₂</td>
</tr>
<tr>
<td>0</td>
<td>location(box₁) ← room₁ ∧ location(box₂) ← room₂</td>
</tr>
<tr>
<td>1, 7</td>
<td>pick-up(agent₂, box₂)</td>
</tr>
<tr>
<td>1, 9</td>
<td>pick-up(agent₁, box₁)</td>
</tr>
<tr>
<td>7, 26</td>
<td>move(agent₂, room₁)</td>
</tr>
<tr>
<td>9, 26</td>
<td>move(agent₁, room₂)</td>
</tr>
<tr>
<td>26, 30</td>
<td>drop(agent₂, box₂)</td>
</tr>
<tr>
<td>26, 30</td>
<td>drop(agent₁, box₁)</td>
</tr>
</tbody>
</table>

### 4.4 SMT-Based Reasoning in TAL

SAT-based reasoning and SMT-based reasoning are conceptually similar. SAT-based reasoning grounds the input with some user specified \( t_{max} \) and then runs a SAT-solver on the resulting propositional problem. Given a TAL\textsuperscript{SMT} theory \( \Gamma \) and observation goal \( G \), SMT-based reasoning performs the following steps to determine \( \Gamma \vdash G \):

- Perform the STPT on \( \Gamma \) and \( G \), constructing \( \Gamma' \) and \( G' \).
- Construct \( \text{Ground}(\Gamma' \land \neg G', s_{max}) \) where \( s_{max} \) is the largest significant time point (or, equivalently, the number of clock time points in \( \Gamma \)).
- Introduce the following constraints for each significant time point \( s' \):
  \[
  C(s') = c[s'] \\
  C(s') \geq C(s' - 1)
  \]
  \( C \) being an uninterpreted function denoting the clock time point of \( s' \).
- Run a SMT-solver on a conjunction of the grounded instance and the added constraints. If the solver returns \( \text{UNSAT} \), \( \Gamma' \vdash G' \) and thus \( \Gamma \vdash G \) by Theorem 12.
The function $C$ is not strictly necessary, but it will prove useful when performing SMT-based planning, which will be done in Chapter 7.
Chapter 5

Survey of Solvers

Much like in the world of SAT, there are many SMT solvers available, each with different focus and performance. Competitions\(^1\) are held annually as well.

SMT-LIB\(^2\) is a library with benchmarks, of which a subset is used in SMT-COMP. SMT-LIB also defines a Lisp-like input language and a set of theories and logics that solvers can support. Logics are subsets of theories that can be useful for solvers to group together.

The aim of this survey is to determine the efficiency in SMT planning compared to SAT planning and which SMT solvers that are suited for planning. Therefore, I won’t run any SMT-LIB benchmarks in this report, but rather run planning problems. Remember though that SMT is a general-purpose reasoning framework and not constructed for the special purpose of planning. Planning is just one of many interesting reasoning problems to which it can be applied.

5.1 Criteria

In this survey, we are interested in the solvers that support any SMT-LIB logic that includes linear arithmetic, namely:

**LRA** Closed logic formulas with linear real arithmetic,

\(^1\)SMT-COMP, http://www.smtcomp.org/
\(^2\)http://combination.cs.uiowa.edu/smtlib/
QF_LRA  Unquantified logic formulas with linear real arithmetic, and

QF_UFLRA  Unquantified logic formulas with linear real arithmetic with uninterpreted sort, function, and predicate symbols. Solvers that support this also supports QF_LRA by definition.

Of the twelve solvers that participated in SMT-COMP 2009, four of them support at least one of the above. These solvers have been benchmarked and measured according to a number of criteria:

**Speed**  Speed is of course important, but can be measured in many ways. The most important aspect in planning is incrementality – the ability to add clauses to a partially solved problem. As we will see in chapter 7, it is preferable if the solver supports some sort of state preservation between solves.

**API**  Availability and documentation. Encoding a problem as text file to feed the solver isn’t really viable for problems that must be solved many times per second.

**License**  An open and permissive license such as MIT, BSD or Apache improves the possibilities of applying the solver.

**Encoding**  A lazy encoding approach is preferable for performance reasons, see section 2.3 on page 9 for a definition of different encoding schemes.

### 5.2 Method

Considering the aim of this thesis, to investigate the possibility to use SMT and SAT as a way of doing general purpose reasoning and planning in TAL, the solvers will be compared against the existing solutions for reasoning and planning in TAL – VITAL and TALplanner, respectively.

Each benchmark will be run until the 95% confidence interval\(^3\) becomes lower than 10% of the average time, but at least five times in the cases that the timings converge fast. The timeout for all solvers is set to 30 minutes.

\(^3\)The interval within which a sample lies with a certain confidence. It is an indication of how statistically reliable a value is. A 95% confidence interval means that a sample will lie in this interval with a probability of 95%.
For reasoning benchmarks, I will run many of the larger non-experimental examples in VITAL in the SAT and SMT-based systems and compare performance.

There are, regarding planning, some considerations that must be made when comparing different systems. Different systems have often different goals, and this is no exception. TALplanner is a forward chaining state space search planner which is guided by control rules to prune the search space. It participated, and won, the hand-tailored planner track in the AIPS-2000 competition. As a hand-tailored planner, it can scale to much larger problems given that efficient control rules are written.

The SAT and SMT-based approaches, however, are general purpose reasoning systems and thus subject to the same scalability problems as fully automated planners. It is possible to write simple control rules for SAT-planners as well, taking advantage of the fact that some subproblems in SAT are linear instead of exponential. I will not write any control rules for the SAT or SMT-planners in this thesis, and will therefore include TALplanner without control rules (but with a maximum search depth given) in the benchmark as well.

The domain that will be used is the Logistics domain from AIPS-2000, which has very well written and efficient control rules for TALplanner. The original domain uses unit time in all actions – all actions take one time point. I will construct a timed benchmark from the unit time version by introducing a distance between all locations and a speed of all vehicles. The unit time problems will be rewritten into SAT planning instances and the timed version to SMT instances.

I will also make a comparison between planning as satisfiability and planning as SMT to highlight the differences between the two approaches. I will take a planning problem, start with unit time actions and increase their duration until any effects can be observed. I will do this for a number of planning problem of increasing complexity – creating two axis of comparison: clock time points, which varies with the duration of the actions, and significant time points, which varies with the complexity of the goal. The same planning domain will be used in the entire benchmark to minimize hidden variable bias.
5.3 Results

I will present the results below, with instructions on how to interpret the graphs on the following pages. Then, I will discuss these results and draw conclusions in the next section.

The solvers

Table 5.1 on page 32 shows which SMT solvers were available and supported at least one of the desired logics. For the SAT planning problems, I’ve included, thanks to its performance to size ratio, MiniSat (Eén and Sörensson, 2004), and clasp\(^4\) since it scored first place in two categories in the most recent SAT competition.

The API field is a subjective measure, assessed by looking at header files and documentation. CVC3 has a gigantic API because it supports almost all logics defined by SMT-LIB. OpenSMT, on the other hand, has a very concise API with the possibility of building formulas and expressions in a semi object-oriented way. MathSAT beginning with version 4 and Yices 1 have also got a semi object oriented API, with interfaces in C.

All available APIs support stack based assumptions – that is, you can add certain formulas as assumptions, check the satisfiability and then retract them if the problem turned out to be unsatisfiable.

Reasoner Benchmarks

Reasoning in the domains included in VITAL are unmeasurably fast – both VITAL and all SAT/SMT-solvers are finished within 40 milliseconds, and there are too much statistical instability in this time frame to make an accurate measurement. For reference, the grounder input for the Russian Hijack Scenario, formalized in TAL by Doherty and Kvarnström (2008), is included in Appendix A.4.

Planner Benchmarks

Figure 5.1 shows the two SAT solvers running on the 15 first problems in the AIPS-00 logistics domain. While they don’t scale as well as TALplanner, they are faster for smaller problems (for makespans smaller than 15 time points in these problems).

\(^4\) http://potassco.sourceforge.net/
Figure 5.2 shows all SMT solvers, together with TALplanner both with and without control rules, on the first 15 problems in the timed version of the AIPS-00 logistics domain. From the graph, we can immediately see that adding linear constraints to the planning problem has a constant performance penalty. It is also obvious that SMT planning can't compete with TALplanner's hand tailored control rules, but also that SMT planning scales better than TALplanner without control rules (though neither one managed to solve all 15 problems).

### Comparing SAT to SMT

Figure 5.3 shows a comparison between SAT and SMT on the same planning domain with different plan and action lengths. The dark opaque surface is the clasp SAT solver and the light transparent surface is the Yices2 SMT solver. The Y axis is timings in milliseconds – note that this axis is linear in comparison to the logarithmic axis in the other benchmarks.

The graph shows that planning as satisfiability has a lower overhead and scales better than planning as SMT in the number of significant time points. It does not, however, scale in the direction of clock time points, where planning as SMT is constant.

### 5.4 Conclusions

As expected, neither SAT nor SMT planning can compete with a hand tailored planner. Control rules guide the search and allow TALplanner to scale very well to larger problems. However, control rules are also a tradeoff between performance and flexibility in the domain: if it is unknown at design time of the system which kinds of problems it will face, the use of control rules can render some conclusions impossible to reach.

However, a problem independent planner is probably easier to use when formalizing new domains. When the domain is working, control rules can be written to improve the performance of the planner.

The comparison between SAT and SMT planning shows that SMT planning really can't compete with SAT planning in STRIPS-like domains. But if plans with realistic durations are needed, SAT planning breaks down very quickly. The makespans of the plans in the timed version of the Logistics domain are in the order of tens of thou-
sands of clock time points, something a SAT planner would be completely unable to handle.

As we will see in chapter 8, SAT based planning can be used in environments with limited resources, such as in an embedded system. The SAT and SMT solvers are memory usage bounded by the size of the problem, making maximum memory consumption very predictable. For large scale planning problems with high performance demands, a hand tailored planner such as TALplanner is a much better choice.

<table>
<thead>
<tr>
<th>Solver</th>
<th>API</th>
<th>Language</th>
<th>Encoding</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVC3(^a)</td>
<td>Very big</td>
<td>C++</td>
<td>Lazy</td>
<td>BSD-like</td>
</tr>
<tr>
<td>MathSAT(^b)</td>
<td>Good</td>
<td>C</td>
<td>Lazy</td>
<td>Non-commercial</td>
</tr>
<tr>
<td>OpenSMT(^c)</td>
<td>Good</td>
<td>C++</td>
<td>Lazy</td>
<td>GPL v3</td>
</tr>
<tr>
<td>Yices 1.0.27(^d)</td>
<td>Good</td>
<td>C</td>
<td>Lazy</td>
<td>LGPL</td>
</tr>
<tr>
<td>Yices2 proto(^e)</td>
<td>None</td>
<td>Unknown</td>
<td>Lazy</td>
<td>Non-commercial</td>
</tr>
</tbody>
</table>

\(^b\) Bruttomesso et al. (2008), http://mathsat4.disi.unitn.it/
\(^c\) http://verify.inf.unisi.ch/openam
\(^d\) http://yices.csl.sri.com
\(^e\) Dutertre and de Moura (2006), http://yices.csl.sri.com/

Table 5.1: Comparison of SMT solvers.
Figure 5.1: Planning as satisfiability in the logistics domain, time in milliseconds.
Figure 5.2: Planning as SMT in the timed logistics domain, time in milliseconds.
Figure 5.3: Comparison between planning as satisfiability and planning as SMT.
As a part of this thesis, I have implemented a grounder for TAL. This grounder compiles formulas in a subset of an order-sorted first order logic down to propositional logic. The only restriction imposed by the grounder is that the domains must be finite, each sort can only contain a finite number of elements, and there is a recursion limit restriction on function symbols.

The following chapter will describe the overall architecture, as well as some internals, of this grounder.

## 6.1 Architecture

Figure 6.1 shows how the grounder is used. The grounder supports many output formats and can be used to produce both DIMACS output for use in a SAT-solver, SMT-LIB format for most SMT-solvers as well as some solver-specific formats for Yices1 and CVC3.

Since the grounder is designed to support general first order logic, it does not automate the Trans-function from chapter 3. However, some meta-programming functionality, such as textual macro expansion or file inclusion, can be done by running a preprocessor on the input, for example the epp C preprocessor included in GCC. Using this, the base constructs and some macros of TAL can be kept in a file for inclusion in all other domain files.
6.2 Input Language

Each statement in the input can be either a type declaration or a formula. Let’s begin by describing the different kind of type declarations that exist. I will give some examples at the end of this chapter.

: **type** *typename* Creates a new top-level type with the given name. Top-level types are used to create a common base type for certain domains.

: **type** *typename*[ *typename*, …] Creates a new parameterized top-level type. Top-level types can be parameterized on other types to indicate that they represent another type. This is similar to generics in Java.

: **integertype** *typename* [ *min..max* ] Creates a new top-level integer type with the specified range. The range is inclusive.

: **instances** *supertype* *subtype* { *identifier*, … } Creates a new subtype of a given supertype and creates a set of constants belonging to that subtype.

: **function** *valuetype* *name* ( *param*, … ) Creates a new function belonging to the given valuetype. Each parameter must be a typename.

: **numfunction** *name* ( *param*, … ) Creates a new numeric function. Numeric functions are functions that only exist in the linear arithmetic part of the problem.

: **predicate** *name* ( *param*, … ) Creates a new predicate with the specified name and arguments.
6.3 Implementation

:bucket name :complete predicate Creates a new formula bucket and performs predication completion on all formulas in this bucket on the given predicate.

The formula syntax resembles traditional first order logic with infix notation, but the connectives are exchanged for ASCII-compatible symbols: Conjunctions are denoted by &; disjunctions by |; implication by ->; equivalence by <->; and finally negation by ~. Quantifiers are written forall and exists followed by a comma-separated list of identifiers. Both round and square parentheses can be used. Formulas are terminated by a semicolon.

A formula can be placed in a bucket by writing :b bucketname before the formula. This indicates that, when all formulas have been grounded, the predicate specified in :bucket will be completed on the conjunction of all formulas in its bucket.

6.3 Implementation

The internal structure of the grounder (depicted in figure 6.2) resembles a traditional multi-pass compiler. In this section, I will follow a typical program from input to output.

When the input in figure 6.3 is given to the grounder, the parser will traverse the document and create types from the declarations as well as parse trees from the formulas. The tokenizer and parser are generated by Flex and Bison⁴.

The next phase of the compiler traverses each parse tree and creates a CNF² representation of it. Some basic optimisations are done in this pass, such as removing clauses containing both ~P and P. Numerical constraints are normalized and checked for linearity in this step as well.

When the formulas have been converted to CNF, they are type checked and grounded. The type check is a by-product of the type inferencer which deduces the most strict type bound on each variable appearing in the formula. This phase will generate ground clauses and ground linear constraints.

¹http://dinosaur.compilertools.net/
²A formula is in Conjunctive Normal Form if it is a conjunction of clauses, where a clause is a disjunction of literals, i.e. an atomic formula or its negation.
The propositional clauses are then post-processed before being written to file. This postprocessing performs some simple optimizations and sorts the clauses according to a time point order.

### 6.4 Summary

While not part of the theory behind SMT-based reasoning in TAL, the grounder is central to making it practical. Much like you seldom write computer programs in assembly language, it is far too much work to manually ground and CNF-convert first order logic formulas by hand to be viable.

The grounder has in practice been proved very stable and efficient in my solver benchmarks. I have not found any related publications or any other grounders, so I can unfortunately not perform any benchmarks.
However, the speed of the grounder is not the performance bottleneck when planning. Grounding is done once on the problem domain while the actual planning is done multiple time with different initial states and goals.
6. A TAL Grounder

:define Per(t, f) (~Oclude(t+1, f) ->
    forall __v [Holds(t+1, f, __v) <-> Holds(t, f, __v)])
#define Dur(t, f, v) (~Oclude(t, f) -> Holds(t, f, v));

#define X(t, f) Oclude(t, f)
#define H(t, f, v) Holds(t, f, v)
#define O(t1, t2, a) Occurs(t1, t2, a)
#define R(t, f, v) (X(t, f) & H(t, f, v))
#define I(t, f, v) (X(t, f) & H(t, f, v))

#define Ct(t, f, v) (Holds(t, f, v) & ~Holds((t)-1, f, v))
#define Cf(t, f, v) (~Holds(t, f, v) & Holds((t)-1, f, v))
#define C(t, f, v) (Ct(t, f, v) | Cf(t, f, v))

Figure 6.3: The TAL predicates, types, foundational axioms and textual macros
Part III

Result
Automated Planning with SMT

Planning is the problem of determining a sequence of actions that satisfies a goal. A planner is an application which takes a domain, an initial state and a goal, and returns such a sequence.

There are many algorithms that can be used when planning. For example, TAL-planner uses forward chaining search (Kvarnström and Doherty, 2001), while FF and its derivatives use hill climbing (Emil Keyder, 2009).

Kautz and Selman (1992) developed a method for planning as satisfiability. This chapter will demonstrate how this method is applicable when planning as SMT as well.

7.1 Introduction

When planning as satisfiability, Kautz and Selman (1992) state: “a planning problem is not a theorem to be proved; rather, it is simply a set of axioms with the property that any model of the axioms corresponds to a valid plan.”

The problem – including initial state and goals – is formulated as, or converted to, a problem in propositional logic. Running a SAT-solver on this problem will result in a model or UNSAT. In the satisfied case, the plan is constructed by simply picking the actions assigned to true in the model. SATPLAN is depicted in algorithm 2.

I propose a SMTPLAN algorithm, very closely related to the SATPLAN algorithm, described in algorithm 3.
Algorithm 2 SAT\textsc{plan}, as presented in Russell and Norvig (2003).

\textbf{Require: }\textit{problem} $\leftarrow$ a planning problem
\hspace{1em} $T_{\text{max}}$ $\leftarrow$ an upper limit for plan length
1: \hspace{1em} \textbf{for } $T = 0$ to $T_{\text{max}}$ \textbf{do}
2: \hspace{2em} $cnf, mapping \leftarrow \text{translate-to-SAT}(\textit{problem}, T)$
3: \hspace{2em} assignment $\leftarrow \text{SAT-solver}(cnf)$
4: \hspace{2em} \textbf{if} assignment is not null \textbf{then}
5: \hspace{3em} \textbf{return} extract-solution(assignment, mapping)
6: \hspace{2em} \textbf{end if}
7: \hspace{1em} \textbf{end for}
8: \hspace{1em} \textbf{return} failure

Algorithm 3 SMT\textsc{plan} conceptual algorithm.

\textbf{Require: }\textit{problem} $\leftarrow$ a planning problem
\hspace{1em} $T_{\text{max}}$ $\leftarrow$ an upper limit for plan length
1: \hspace{1em} \textbf{for } $T = 0$ to $T_{\text{max}}$ \textbf{do}
2: \hspace{2em} $cnf, constraints, mapping \leftarrow \text{translate-to-SMT}(\textit{problem}, T)$
3: \hspace{2em} assignment $\leftarrow \text{SMT-solver}(cnf, constraints)$
4: \hspace{2em} \textbf{if} assignment is not null \textbf{then}
5: \hspace{3em} \textbf{return} extract-solution(assignment, mapping)
6: \hspace{2em} \textbf{end if}
7: \hspace{1em} \textbf{end for}
8: \hspace{1em} \textbf{return} failure

The similarities between SMT\textsc{plan} and SAT\textsc{plan} are unsurprising. The only difference is that the SAT algorithm has been replaced by the SMT algorithm, and that the corresponding functions now return or take a set of linear constraints too.

### 7.2 Optimizations

The conceptual simplicity, which is the heart of planning as satisfiability, is shared by SMT-planning. But there is much room for improvement.

Optimizations can be done both in the problem specification itself and in the actual planner implementation. High level problem specification optimizations can give huge performance boost in certain domains, while the low level optimizations ensure that no unnecessary operations are done.
### 7.2. Optimizations

#### Implementation-level Optimizations

The algorithm is centered around the `translate-to-SMT` and `SMT-solver` functions. Now, there are two problems with that: `translate-to-SMT` converts a formula into CNF, which may result in an exponential blowup in the number of clauses, and `SMT-solver` employs SAT and Simplex, both of which are exponential in time.

What we want to do is to minimize both the number of calls and the input size to these functions.

#### Binary Search

The worst case for planning according to the original algorithms is $T_{\text{max}}$ solves; this happens either when the plan is $T_{\text{max}}$ steps or if no plan is found. By starting at $T_{\text{max}}/2$ and doing a binary search, the worst case becomes $\log_2(T_{\text{max}})$.

There is a slight drawback with doing a binary search. If you want optimal plans, the best case in the linear case is 1 solve, while the binary search still requires $\log_2(T_{\text{max}})$ solves. Since the solving time increases exponentially with $T$, this might really slow down planning for some cases.

However, if you expect many goals to be unattainable, using binary search is faster. Also, the average case for linear search is $T_{\text{max}}/2$ but $\log_2(T_{\text{max}})$ for binary search. This makes it easy to estimate when to use binary or linear search.

#### Incremental Solving

Calling `translate-to-SMT` each iteration of the loop in the algorithm is wasteful. Instead, let `translate-to-SMT` return $T_{\text{max}}$ sets of clauses, each set $cnf_i$ corresponding to clauses with maximum time point $T$. `SMT-solver` is then called with the union of all sets $cnf_i$ with $i \leq T$.

If the SMT solver supports a stack-like API, the algorithm can be reformulated as in algorithm 4.

The new algorithm has many advantages above the original one. First, the perhaps costly call to `translate-to-SMT` on the entire domain has been replaced by a much lighter call to a similar function `translate-goal-to-SMT`. 
Algorithm 4 SMTPLAN with a stack-like SMT solver.

Require: domain ← a planning problem
        goal ← the goal
        $T_{\text{max}}$ ← an upper limit for the plan length
1: $\langle \text{cnf}_0, \ldots, \text{cnf}_{T_{\text{max}}} \rangle$, $\langle \text{constraints}_0, \ldots, \text{constraints}_{T_{\text{max}}} \rangle$, mapping
   ← translate-to-SMT(problem, $T_{\text{max}}$)
2: solver ← make-SMT-solver()
3: for $T = 0$ to $T_{\text{max}}$ do
4:   $\text{cnf}_0^{\text{goal}}$, constraints$_0^{\text{goal}}$ ← translate-goal-to-SMT(goal, $T$, mapping)
5:   add-to-solver(solver, $\text{cnf}_T$, constraints$_T$)
6:   push-solver-state(solver)
7:   add-to-solver(solver, $\text{cnf}_0^{\text{goal}}$, constraints$_0^{\text{goal}}$)
8:   assignment ← SMT-solve(solver)
9:   if assignment is not null then
10:      return extract-solution(assignment, mapping)
11:   end if
12:   pop-solver-state(solver)
13: end for
14: return failure

Second, the solving is done incrementally. The sets cnf$_0$ to cnf$_{T_{\text{max}}}$ are of roughly equal size, so each iteration of the loop only adds a small bit of the problem to the solver. When solving for a certain time point $T$, the solver already has a model for the problem in $T - 1$, so it only needs to propagate that model into the new time point.

Note that the goal is added separately from the domain clauses, within a push/pop pair. This means that if no plan is found, the goal is retracted from the solver and the solving can continue with the solver in a satisfied state.

High Level Optimizations

High level optimizations have an advantage over low level optimizations; they can reduce the problem size even before the problem reaches the planner.

By trading brevity in the domain formulas for fewer predicates and smaller formulas, the SAT instance size can greatly be reduced. One such optimization is control rules
– rules which “guide” the planner towards the goal. Small Horn-like\(^1\) rules can guide the solver towards the goal faster. There is, however, a tradeoff here – adding too many control rules might choke the solver because of the increase in problem size.

### 7.3 API

When the planner is used standalone – called from the command line for example – the use case is as in figure 7.1. The domain formulas and initial state is written as \(\mathcal{L}(FL)\), grounded by the grounder described in the previous chapter, and given to the planner. The planner also takes a list of goals separate from the domain and initial state.

![Diagram of API](image)

**Figure 7.1:** A SMT planner used as a standalone application.

This is, however, not the intended use for this planner. Instead, I want it to be used from inside another application that requires planning services, such as the computer game described in chapter 8. My implementation of the planner has an API that should be used as depicted in figure 7.2.

\(^1\)A horn clause is a clause with at most one positive literal. Since most SAT solvers use an unit propagation algorithm, they are linear in time for Horn clauses. However, a control rule does not strictly need to be Horn for the performance boost to show.
Figure 7.2: A SMT planner used as an API from inside another program.

The problem files are read once and reused until the planner is freed. In a game, this can be done when a level is loaded. Each time an agent wants to reach a goal, the initial state and goal are created and given to the planner.

I intend to describe how to do this in detail in the next chapter, but first, I am going to introduce the API with which applications plan. The code listings below are C++ classes and structs, with their private members and functions removed for brevity.

**Representing Terms and Formulas**

In order to represent initial state and goals, applications need to be able to construct simple formulas. The API only considers formulas in CNF, which is often enough for what you want.

**Ground Atomic Formulas**

A formula is ground in first order logic if it contains no variables (neither closed nor free). A formula is ground and atomic if it only consists of a constant or a predicate with ground arguments.
The variable-action mapping our grounder gives us is a list of ground atomic formulas. Each GAF corresponds to a boolean variable in the SAT instance. For example, in the logistics domain, \( \text{Occurs}(0, 1, \text{fly}(\text{plane}_1, \text{city}_1-1, \text{city}_2-1)) \) is a GAF.

Numerical variables are ground terms with a special meaning in the problem, for example, \( \text{distance}(\text{city}_1-1, \text{city}_2-1) \) in the timed logistics domain is a ground numeric term. The grounder writes a list of ground numerical terms similarly to the list of GAFs.

Ground atomic formulas and ground terms share syntax, and therefore they share superclass in this API: `GroundAtomicExpression`. Ground terms are constructed by the `GroundTerm` constructor below, while the ground numeric terms and ground atomic formulas are constructed via the storage APIs defined on the next page.

```cpp
class GroundAtomicExpression {
public:
    const std::string &name() const;
    const std::list<GroundTerm> &args() const;
};

class GroundTerm : public GroundAtomicExpression {
public:
    GroundTerm(const std::string &name);
    GroundTerm(const std::string &name, const GroundTerm &arg1);
    GroundTerm(const std::string &name, const GroundTerm &arg1,
               const GroundTerm &arg2);
    GroundTerm(const std::string &name, const GroundTerm &arg1,
               const GroundTerm &arg2,
               const GroundTerm &arg3);
    GroundTerm(const std::string &name, const std::list<GroundTerm> &args);
};

class GroundNumericTerm : public GroundAtomicExpression {
public:
    unsigned int index() const;
};

class GroundAtomicFormula : public GroundAtomicExpression {
public:
```
GAFLiteral operator~() const;
unsigned int index() const;
};

GAFs and GNTs are the primary interface for feeding the planner goals and initial state. It must therefore be easy to construct them. The planner keeps track of all GAFs and GNTs used in the domain and exposes them using two interfaces: GAFStorageInterface and GNTStorageInterface:

class GAFStorageInterface {
public:
    virtual const GroundAtomicFormula *gaf(const std::string &name);
    virtual const GroundAtomicFormula *gaf(const std::string &name,
                                            const GroundTerm &arg1);
    virtual const GroundAtomicFormula *gaf(const std::string &name,
                                            const GroundTerm &arg1,
                                            const GroundTerm &arg2);
    virtual const GroundAtomicFormula *gaf(const std::string &name,
                                            const GroundTerm &arg1,
                                            const GroundTerm &arg2,
                                            const GroundTerm &arg3);
    virtual const GroundAtomicFormula *gaf(const std::string &name,
                                            const std::list<GroundTerm> &args);
};

class GNTStorageInterface {
public:
    virtual const GroundNumericTerm *gnt(const std::string &name);
    virtual const GroundNumericTerm *gnt(const std::string &name,
                                            const GroundTerm &arg1);
    virtual const GroundNumericTerm *gnt(const std::string &name,
                                            const GroundTerm &arg1,
                                            const GroundTerm &arg2);
    virtual const GroundNumericTerm *gnt(const std::string &name,
                                            const GroundTerm &arg1,
                                            const GroundTerm &arg2,
                                            const GroundTerm &arg3);
    virtual const GroundNumericTerm *gnt(const std::string &name,
                                            const std::list<GroundTerm> &args);
};
Using the methods defined above, the GAF and GNT mentioned above are constructed programmatically by:

```cpp
gaf("Occurs", GroundTerm("0"),
    GroundTerm("1"),
    GroundTerm("fly", GroundTerm("plane1"),
    GroundTerm("city1·1"),
    GroundTerm("city2·1")));

gnt("distance", GroundTerm("city1·1"), GroundTerm("city2·1"));
```

### Clauses

Goals can sometimes be disjunctions. The class `Clause` represents a disjunction of GAF literals, and the class `Formula` represents a set of clauses:

```cpp
class GAFLiteral {
public:
    GAFLiteral(const GroundAtomicFormula *gaf, bool _sign = true);

    // Returns the negation of this literal.
    GAFLiteral operator~() const;

    const GroundAtomicFormula *gaf() const;
    bool sign() const;
};

class Clause {
public:
    Clause(const GAFLiteral &l);

    const_iterator begin() const;
    const_iterator end() const;

    void add_literal(const GAFLiteral &l);
};
class Formula {
public:
```
Formula(const Clause &c);
Formula(const GAFLiteral &l);

const_iterator begin() const;
const_iterator end() const;

void add_clause(const Clause &c);
void add_unit_clause(const GAFLiteral &l);
};

The Planner

The planner interface is defined as follows:

class Planner : public GAFStorageInterface,
 public GNTStorageInterface {
 public:
 Planner(const std::string &gafs_file,
 const std::string &gnts_file);

 void add_initial_state(const GroundTerm &fluent,
 const GroundTerm &value);
 void add_initial_state(const GAFLiteral &l);

 void complete_predicate(const std::string &predicate_name);

 bool plan(const GAFLiteral &goal, Plan &out_plan);
 bool plan(const Formula &goal, Plan &out_plan);
};

Initial state is inserted to the planner by two overloaded instances of the method add_initial_state. The first one takes two ground terms, a fluent and a value respectively, and inserts the GAF Holds($0, f, v$). The other overload takes a GAF literal and inserts it directly into the planner. When using this overload, a closed world assumption is often required. The function complete_predicate takes care of this – it inserts negative literals for all GAFs with the predicate name specified not occurring in the initial state.
To exemplify this, consider a scenario where you have doors connecting rooms. If you insert `Door(room_1, room_3)` and `Door(room_3, room_1)` as initial state and do not close the world, the SMT solver can infer plans using, for example, `Door(room_1, room_2)` since its negation is not specified. The `complete_predicate` method will insert `¬Door(room_1, room_2)`, and possibly many others, since it does not appear in the initial state.

Assuming complete information about initial state, `Holds` does not need to be completed since the unique value axioms state that each fluent has exactly one value at each time point.

**Plans**

Plans consist of three parts: a clock time point mapping from significant time points to clock time points, a list of actions to begin at each time point, and the complete model for the problem.

```cpp
struct Plan {
  struct Action {
    int start;
    int end;
    GroundTerm action;
  }

  typedef std::vector<Action> actions_t;

  std::vector<float> clock_time_points;
  std::vector<actions_t> actions;
  std::vector<GAFLiteral> model;
};
```

The arrays `clock_time_points` and `actions` are guaranteed to be of equal size, and their length represent the number of significant time points in the plan. Each entry `i` in the `actions` array is an array with actions whose `start = i`. 
7.4 Summary

Any application that would need planning in a predefined domain, such as a game or a reasoning system in a UAV, can use the API above. This is, to my knowledge, the first planning API that can be used efficiently both in a desktop application and in an embedded system.
Chapter 8

A Game Application

Given the empirical study, and the limitations it showed, there are still interesting applications where these techniques are applicable. One such application is computer games, to which we now turn.

During 2009, I implemented an iPhone game together with Martin Magnusson (PhD student at IDA, LiU). This game took advantage of planning as satisfiability when guiding the opponents to their goals. The experiences I gained from implementing that game influenced the API described previously – the API grew out of the game into a separate library.

This chapter will describe this game, Trick Trap\(^1\), and how SAT planning is used.

8.1 Introduction

Planet Platform has been invaded by visitors from the 3rd dimension. You and your buddies must trap them in “depth chambers”, where they’ll be beamed back to wherever they came from. (This explains why they appear inside locked rooms. They arrived from dimension Z and so did not need to pass through the walls.)

\(^1\)http://www.tricktrap.com
Trick Trap is a platform-based puzzle game, where you guide your character and his friends through a level, avoiding bad-guys which try to thwart you. A level consists of rooms and open areas, connected by doors.

The iPhone was chosen as platform form its potential to showcase AI to a wider audience, and to prove that planning can be very efficient with the limited resources available in an embedded system too.

### 8.2 Gameplay

The player guides the character by tapping where they should go. On some levels, there are more than one character; the active character will then be highlighted, and you switch active character by tapping on them.

Characters can pick up and drop keys, as well as open and close doors. This is done by tapping on the object with which an action should be performed, and a menu appears (figure 8.1).

![Figure 8.1: Interaction with Trick Trap](image)

(a) The player wants to pick up a key  
(b) A menu appears when the key is tapped  
(c) The character picks up the key

The opponents, disguised in dino suits, can perform the same actions a player can perform. The player must *trick* the opponents by taking keys, opening and closing doors in order to *trap* them in depth chambers to complete the level. Hence the name Trick Trap.
8.2. Gameplay

Figure 8.2: A level in Trick Trap. There is a blue key to right of the player.

**World Discretization**

A level consists of rooms and doors between them. There may be open areas, but they are still considered rooms. All tiles an agent can walk to must be part of a room. In the logic, these are the datatypes:

- **node** A room in a level.
- **mobile** A supertype for all objects in the world that have a location.
- **door** A door between two rooms (nodes).
- **agent** An agent, subtype of mobile.
- **key** A key, subtype of mobile.

To indicate that a door and a node is connected, the predicate $Door(door, node)$ is set to true for the appropriate values. As the connectivity of rooms is constant over time, we can use a simple predicate instead of a fluent.
The following TAL features are defined in the domain:

\(at(mobile) \equiv node\) An object is at a specific node.

\(is\text{-}open\text{(door)}\) A door is open.

\(holding\text{(agent)} \equiv key\) An agent is holding a specific key.

All action occurrences and domain constraints are given in figure 8.3. There are five actions: \(walk, take, drop, open, close\), defined by the first five formulas. The following six formulas are constraints which prevent agents from performing certain actions simultaneously. For example, an agent can not walk through a door and close it (or another door for that matter) at the same time.

The last two formulas are constraints added because of race conditions\(^2\) in the game. For example, if one agent walks through a door while, simultaneously, another agent locks the door, the result becomes dependent on which agent reaches the door first. Race conditions are always hard to predict in advance, especially since we want multi-agent plans in the game.

**Agent Architecture**

Each agent is a state machine. They have three primary states: \(Idle\), where the agent stands still, possibly waiting for the planner to return a plan. \(Following\text{ a path}\), since the planner and path-finder are two separate systems. And finally, \(executing\text{ a plan}\).

While each agent has its own desires\(^3\) stack, it is the level that acts as a central hub for managing desires. When a player performs an action or when an opponent finishes executing a plan, desires are generated for the opponents. Currently, opponents have two desires: If there is an open door to the node where a player is standing, it should be closed (formula 8.1), and the opponent should avoid players by not standing in the same room as them (formula 8.2 where \(npc\) is the agent to plan for and \(pc\) is the player controlled agent). The antecedent of the formulas below are computed by inspecting the level state, in order to minimize the number of calls to the planner.

\(^2\)A race condition is when the result of a process becomes dependent on the specific timing of events.

\(^3\)A desire is a goal that the agent has not attended yet. One plan is needed to fulfill each desire.
\[ [t_1, t_2] \text{walk}(a, d) \leadsto [t_1] \text{at}(a) \equiv n_1 \land \text{Door}(d, n_2) \land n_1 \neq n_2 \rightarrow R([t_2] \text{at}(a) \equiv n_2) \]
\[ [t_1, t_2] \text{take}(a, k) \leadsto [t_1] \text{at}(a) \equiv n \]
\[ 
\rightarrow \neg [t_1] \text{holding}(a_2) \equiv k \land [t_1] \text{at}(k) \equiv n \land R([t_2] \text{holding}(a) \equiv k) \]
\[ [t_1, t_2] \text{drop}(a, k) \leadsto [t_1] \text{holding}(a) \equiv k \land [t_2] R([t_2] \neg \text{holding}(a) \equiv k) \]
\[ [t_1, t_2] \text{open}(a, d) \leadsto [t_1] \text{at}(a) \equiv n \land \text{Door}(d, n) \]
\[ 
\rightarrow [t_1] \neg \text{is-open}(d) \land R([t_2] \text{is-open}(d)) \]
\[ [t_1, t_2] \text{close}(a, d) \leadsto [t_1] \text{at}(a) \equiv n \land \text{Door}(d, n) \]
\[ 
\rightarrow [t_1] \text{is-open}(d) \land R([t_2] \text{is-open}(d)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \text{walk}(a, d_1) \lor [t, t'] \neg \text{open}(a, d_2)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \text{walk}(a, d) \lor [t, t'] \neg \text{take}(a, k)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \text{walk}(a, d) \lor [t, t'] \neg \text{drop}(a, k)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \text{walk}(a, d_1) \lor [t, t'] \neg \text{close}(a, d_2)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \neg \text{drop}(a, k) \lor [t, t'] \neg \text{open}(a, d)) \]
\[ \forall t, t', a, d_1, d_2 ([t, t'] \neg \text{close}(a, d) \lor [t, t'] \neg \text{open}(a, d)) \]
\[ \forall t, t', a_1, a_2, d_1, d_2 ([t, t'] \text{walk}(a_1, d_1) \land [t, t'] \text{walk}(a_2, d_2) \rightarrow a_1 = a_2) \]
\[ \forall t, t', a_1, a_2, d ([t, t'] \text{walk}(a_1, d) \land [t, t'] \text{close}(a_2, d) \rightarrow a_1 = a_2) \]

Figure 8.3: Action specifications and domain constraints in Trick Trap’s domain.

Only the consequent of the formulas are added as goals.

\[ \forall_{pc, n, d} ([0] \text{at}(pc) \equiv n \land \text{Door}(d, n) \land [0] \text{is-open}(d) \rightarrow [T_{\max}] \neg \text{is-open}(d)) \]
(8.1)

\[ \forall_{pc, n} ([0] \text{at}(pc) \equiv n \land \text{at}(npc) \equiv n \rightarrow [T_{\max}] \neg \text{at}(npc) \equiv n) \]
(8.2)

Agents are completely asynchronous, they update their state in a method `updateGameLogic()`, called by the level on each frame. Events are sent from the level to agents through messages such as `addDesire()` and `followPath()`. All planning is done sequentially by a central planner object in a separate thread and each agent gets a `FuturePlan` object when asking this object to plan, representing a plan which may or may not exist yet.
Multi-Agent Plans

Multi-agent plans are worth discussing a bit more. When there are two or more opponents, there is always only one agent that performs the planning. This agent is called a team leader, while the others are team members. To avoid duplication of desires, the only agent which has desires is the team leader. When the team leader wants to reach a goal, it plans for all agents in its team and synchronizes their actions. Concurrent plans are a side effect of SAT planning; by not explicitly denying actions for other agents, concurrent plans are automatically constructed if necessary.

There are, however, many pitfalls when doing multi-agent plans. For example, SAT planning has no concept of “shortest plans”; single-agent plans become optimal in the number of actions because the plan length is equal to the number of actions. But when doing multi-agent plans, there can be any number of actions per time point (limited by the number of team members). Informally, the SAT solver can add unnecessary concurrent actions as long as the goal is satisfied. Figure 8.4 illustrates this phenomenon.

\[
\begin{align*}
\text{Occurs}(0,1,\text{take(agent2,key1)}) & \quad \text{Occurs}(0,1,\text{take(agent2,key1)}) \\
\text{Occurs}(0,1,\text{open(agent2,door1)}) & \quad \text{Occurs}(1,2,\text{open(agent2,door1)}) \\
\end{align*}
\]

Figure 8.4: Two plans with equal makespan and equal final state look equally optimal to a SAT planner.

There are at least two solutions to the above problem. You can either globally deny concurrent actions and postprocess the plan afterwards to merge time points with non-conflicting actions, or you can eliminate those problematic concurrent actions that appear in testing by adding control rules. While the second option is not as general as the first one, it was what we chose to do in Trick Trap. The last two formulas in figure 8.3 remove the unwanted behaviors that we observed during testing.
8.3 Advantages and Disadvantages

Using automated planning as a high level game AI has both its drawbacks and benefits. We gathered a lot of experience in applying automated planning during the development of Trick Trap; I intend to share the conclusions here.

Atomicity

With our planner, there are no secondary goals or partially fulfilled goals. If an agent can not reach its goal, no action can be taken. This is what i call atomicity, a goal can not be separated and not be incremental.

Consider the abstract goal “all doors facing an agent must be closed”. If an agent executes such a plan, the player is locked up and the opponents have won. On the other hand, if the opponent can not find such a plan, no action at all can be taken. We had to reformulate the goal as “a door facing an agent must be closed” and iteratively plan as long as it was not fulfilled.

This led to another problem, however. If an opponent were standing on a node $n$ where a player also stood, we added the goal “the agent must not stand on $n$”. This could lead to the opponent moving back and forth between two nodes, both with agents standing on them. This goal had to be merged into the slightly larger “an agent can not stand on a node where there is another agent”.

Underspecified Domain

As we quickly noticed, a SAT planner will do everything it is not explicitly told not to do in order to attain its goal. We began by not stating that an agent can not walk through a door and close one (the same or another) at the same time. After we watched some plans generate, it quickly grew into the six control rules in figure 8.3.

The predicate completion facility, mentioned in chapter 7, also came to use. It is a part of good object oriented design to let each object in the level manage and insert its own state—such as where an agent is, or which nodes a door is connected to— into the planner. But it is very unnatural to let them state their negations as well. If we let a door object tell the planner which nodes it is connected to, we must have a simple way of telling the planner which node it is not connected to as well.
Changing Initial State

To make as much use of idle time as possible, an agent should begin to plan for its next goal while it is executing a plan for its current goal. But while an agent is executing a plan, the initial state may change, so the plan it created might be impossible to execute in the new initial state.

We are currently experimenting with using the predicted final state in the currently executing plan as initial state for the new plan. But at the time of writing, it is too early to tell whether this is a workable solution.

Powerful Abstraction

This list of difficulties in applying planning might give the reader the impression that there are only drawbacks with using planning. This is certainly not the case. Planning in Trick Trap has proven to be a very powerful abstraction. A level designer can focus on designing a level while knowing that the characters in the level will make smart decisions without having to think about how.

8.4 Performance

$T_{max}$ in Trick Trap is currently set to 15, which gives us the possibility to have very long plans. Most plans are not that long, and many plans are aborted before completion since circumstances change quickly in a dynamic environment such as a computer game. Still, performance is very good, even on the iPhone. There is, sometimes, a noticeable delay where the opponents are standing still before starting to move, but this only occurs when a combination of events involving the player leads to a number of consecutive failed plans.

The player character also does some planning, for example to pick up a reachable key in order to open a door. This planning must be very fast to ensure a responsive user interface. We prioritize and limit the maximum plan length for the player to improve the response time. Since the planner works by dequeueing items off a list of work orders, we can insert the player’s planning requests at the beginning of the list instead of at the end. The plan length limit must be there, otherwise the player could use the automated planner to win and not have to do any thinking for himself; rendering the whole puzzle-part of the game useless.
8.5 Summary

While Trick Trap is yet to be released (mostly due to graphics and level design), I still consider it a successful demonstrator of planning both on an embedded system as well as in a game. The autonomy of an agent has both its benefits and drawbacks, as stated in the previous section; but the planning technology mixed very well with the game mechanics in Trick Trap and simplifies the development of new levels.

It will be interesting to see users impressions when it is released on the App Store.
While we’ve reached the end of this report, we’ve only begun the journey towards applications of logic-based reasoning technology. I would like to revisit some of the key milestones that were part of this thesis.

**Planning as Satisfiability**

A SAT or SMT reasoner applied to planning might not take advantage of the high-level structure of planning as well as special-purpose planners can. Nevertheless, as Kautz and Selman (1992) has shown, SAT planning can be highly competitive.

Moreover, planning as SAT or SMT has advantages over traditional planning. One of the key points, which I haven’t mentioned yet, is failed plans. Failed plans are very common in the real world; especially if the generated plan is long. By using the fact that a SAT planner really is a general purpose reasoner, and thus capable of adhering to arbitrary constraints in the domain, an agent can plan “around” failed actions. For example, if an agent failed to execute an action in a plan, it can replan from the original initial state and insert the plan executed so far, including the failed action negated. The SAT planner will then generate a new plan without the action that failed and take advantage of the already executed actions.

I believe that the ability to insert arbitrary domain constraints is the key strength of planning as satisfiability. But there are also other advantages, such as the fact that there are no restrictions in how you can specify preconditions and effects, unlike STRIPS and PDDL+. 
The Significant Time Point Concept

The second key milestone is the introduction of SMT. Through the concept of significant and clock time points, the computational complexity of planning is no longer a function of the plan’s length, but the plan’s complexity – the number of features that change during the plan. This is a very powerful abstraction which lets actions take arbitrary amount of time without imposing greater computational complexity.

9.1 Future Work

Finally, the possibilities for future work abounds, and we mention only one here. Shin and Davis (2005) describe a very comprehensive set of extensions to the PDDL+ language. These extensions allow their planner to use continuous time, much like in this thesis, but many other continuous metrics as well. For example, it can reason about resources with either instantaneous or continuous change in time. Reasoning about resources is already possible in TAL, but automated reasoners such as VITAL only reason about integer or fixed-point resources.
Appendix A

Scenarios Used in the Survey

A.1 TAL Axioms for the Grounder

:type value
:type fluent[value]
:type action

:predicate Holds(timepoint, fluent[T], T)
:predicate Occlude(timepoint, fluent[value])
:predicate Occurs(timepoint, timepoint, action)

:instances value bool {true , false}
:integer type timepoint [0 .. MAX_T]

:bucket acs :complete Occlude

// If the constant PLANNING is not defined, an action occurrence
// bucket will be defined and Occurs will be completed.
#ifndef PLANNING
   :bucket occ :complete Occurs
#endif

// Unique value axioms.
forall t, f, v1, v2 [Holds(t, f, v1) & Holds(t, f, v2) => v1==v2];
forall t, f [ exists v Holds(t, f, v) ];
// If CUSTOM_PERSISTENCE is not defined, persistence is assumed
// on all fluents.
#ifndef CUSTOM_PERSISTENCE
  forall t, f, v [\neg\text{Oclude}(t+1, f) ->
    [\text{Holds}(t, f, v) <-> \text{Holds}(t+1, f, v)]];
#endif

#define Per(t, f) (~\text{Oclude}(t+1, f) ->
  forall v__ [ \text{Holds}(t, f, v__) <->
    \text{Holds}(t+1, f, v__) ])
#define Dur(t, f, v) (~\text{Oclude}(t, f) -> \text{Holds}(t, f, v))

#define X(t, f) \text{Oclude}(t, f)
#define R(t, f, v) (X(t, f) & \text{Holds}(t, f, v))
#define I(t, f, v) (X(t, f) & \text{Holds}(t, f, v))

#define Ct(t, f, v) (\text{Holds}(t, f, v) & \neg\text{Holds}((t)-1, f, v))
#define Cf(t, f, v) (~\text{Holds}(t, f, v) & \text{Holds}((t)-1, f, v))
#define C(t, f, v) (Ct(t, f, v) | Cf(t, f, v))

A.2 The AIPS-00 Logistics Domain

#define PLANNING

#include "TAL·base.txt"

:instances value thing []
:instances thing package []
:instances thing vehicle []
:instances vehicle truck []
:instances vehicle airplane []
:instances value location []
:instances location airport []
:instances value city []

:function fluent[bool] at(thing, location)
:function fluent[bool] in(package, vehicle)
:predicate InCity(location, city)
:bucket citydef :complete InCity

:function action load(package, vehicle, location)
:function action unload(package, vehicle, location)
:function action drive(truck, location, location, city)
:function action fly(airplane, airport, airport)

// #acs load(p, v, l)
:b acs forall t, p, v, l [ Occurs(t, t+1, load(p, v, l)) ->
    Holds(t, at(p, l), true) &
    Holds(t, at(v, l), true) &
    ~Holds(t+1, at(p, l), true) &
    Holds(t+1, at(v, l), true) &
    Occlude(t+1, at(p, l)) &
    Occlude(t+1, at(v, l)) ];

// #acs unload(p, v, l)
:b acs forall t, p, v, l [ Occurs(t, t+1, unload(p, v, l)) ->
    Holds(t, in(p, v), true) &
    Holds(t, at(v, l), true) &
    Holds(t+1, at(p, l), true) &
    Occlude(t+1, at(p, l)) &
    Occlude(t+1, in(p, v)) ];

// #acs drive(tr, from, to, c)
:b acs forall t, tr, from, to, c [ Occurs(t, t+1, drive(tr, from, to, c)) ->
    Holds(t, at(tr, from), true) &
    InCity(from, c) &
    InCity(to, c) &
    Occlude(t+1, at(tr, from)) &
    Occlude(t+1, at(tr, to)) ];

// #acs fly(a, from, to)
:b acs forall t, a, from, to [ Occurs(t, t+1, fly(a, from, to)) ->
    Holds(t, at(a, from), true) &
    Occlude(t+1, at(a, from)) &
    Occlude(t+1, at(a, to)) ];

// at is a partial function, uniqueness required
forall t, o, 11, l2 [
    Holds(t, at(o, 11), true) &
    Holds(t, at(o, l2), true) -> 11 == l2 ];
// in is a partial function, uniqueness required
forall t, p, v1, v2 [
Holds(t, in(p, v1), true) & Holds(t, in(p, v2), true) -> v1 = v2 ];

// We can not load and drive at the same time
forall t, tr, from, to, c, p [ Occurs(t, t+1, drive(tr, from, to, c)) ->
~Occurs(t, t+1, load(p, tr, from)) &
~Occurs(t, t+1, unload(p, tr, from)) ];

// and we can not load and fly at the same time, either
forall t, a, from, to, p [ Occurs(t, t+1, fly(a, from, to)) ->
~Occurs(t, t+1, load(p, a, from)) &
~Occurs(t, t+1, unload(p, a, from)) ];

A.3 The Timed Logistics Domain

#define PLANNING

#include "TAL-base.txt"

:instances value thing []
:instances thing package []
:instances thing vehicle []
:instances vehicle truck []
:instances vehicle airplane []
:instances value location []
:instances location airport []
:instances value city []

:function fluent[bool] at(thing, location)
:function fluent[bool] in(package, vehicle)
:function fluent[bool] loading(vehicle)

:numfunction distance(location, location)
:numfunction load·time()
:numfunction unload·time()

:numfunction C(timepoint)

:predicate InCity(location, city)
bucket citydef :complete InCity

function action load(package, vehicle, location)
function action unload(package, vehicle, location)
function action drive(truck, location, location, city)
function action fly(airplane, airport, airport)

C(0) == 0;
forall t C(t) <= C(t+1);

// #acs load(p, v, l)
:b acs forall tl, t2, p, v, l [ Occurs(tl, t2, load(p, v, l)) -> t2 > tl &
C(t2) == C(tl) + load-time() &
Holds(tl, at(p, l), true) & Holds(tl, at(v, l), true) &
Holds(t2, at(p, l), true) &
~Holds(tl+1, at(p, l), true) & Occlude(tl+1, at(p, l)) &
(t2 > tl+1 -> Holds(tl+1, loading(v), true) & Occlude(tl+1, loading(v)) &
~Holds(t2, loading(v), true) & Occlude(t2, loading(v))) &
Holds(t2, in(p, v), true) & Occlude(t2, in(p, v)) ];

// #acs unload(p, v, l)
:b acs forall tl, t2, p, v, l [ Occurs(tl, t2, unload(p, v, l)) -> t2 > tl &
C(t2) == C(tl) + unload-time() &
Holds(tl, in(p, v), true) & Holds(tl, at(v, l), true) &
Holds(t2, at(v, l), true) &
Holds(tl+1, at(p, l), true) & Occlude(tl+1, at(p, l)) &
(t2 > tl+1 -> Holds(tl+1, loading(v), true) & Occlude(tl+1, loading(v)) &
~Holds(t2, loading(v), true) & Occlude(t2, loading(v))) &
~Holds(t2, in(p, v), true) & Occlude(t2, in(p, v)) ];

// #acs drive(tr, from, to, c)
:b acs forall tl, t2, tr, from, to, c [ Occurs(tl, t2, drive(tr, from, to, c)) -> t2 > tl &
C(t2) == C(tl) + TRUCK_SPEED * distance(from, to) &
~Holds(tl, loading(tr), true) & Holds(tl, at(tr, from), true) &
InCity(from, c) & InCity(to, c) &
~Holds(tl+1, at(tr, from), true) & Occlude(tl+1, at(tr, from)) &
Holds(t2, at(tr, to), true) & Occlude(t2, at(tr, to)) ];
// #acs fly(a, from, to)
: b acs forall t1, t2, a, from, to [ Occurs(t1, t2, fly(a, from, to)) -> t2 > t1 &
  C(t2) == C(t1) + AIRPLANE_SPEED * distance(from, to) &
  Holds(t1, at(a, from), true) & ~Holds(t1, loading(a), true) &
  ~Holds(t1+1, at(a, from), true) & Occlude(t1+1, at(a, from)) &
  Holds(t2, at(a, to), true) & Occlude(t2, at(a, to)) ];

// at is a partial function, uniqueness required
forall t, o, l1, l2 [  
  Holds(t, at(o, l1), true) & Holds(t, at(o, l2), true) -> l1==l2 ];
// in is a partial function, uniqueness required
forall t, p, v1, v2 [  
  Holds(t, in(p, v1), true) & Holds(t, in(p, v2), true) -> v1==v2 ];

A.4 The Russian Hijack Scenario

#define MAX_T 16
#define CUSTOM_PERSISTENCE
#include "/TAL-base.txt"

:instances value thing ()
:instances thing airplane [sas609]
:instances thing person [boris, dimiter, erik]
:instances thing pthing [gun, comb1, comb2, comb3]

:instances value location {home1, home2, home3, office, air, airport}
:instances location runway {run609, run609b}

:instances value pocket {pocket1, pocket2, pocket3}

:function fluent[location] loc(thing)
:function fluent[bool] inpocket(person, pthing)
:function fluent[bool] poss-board(person, airplane)
:function fluent[bool] drunk(person)
:function fluent[bool] onplane(person, airplane)

:function action put(person, pthing, pocket)
:function action travel(person, location, location)
:function action fly_airplane, runway, runway)
:function action board(person, airplane)

// Observations
Holds(0, poss_board(erik, sas609), true);
Holds(0, loc(boris), home1);
Holds(0, loc(gun), office);
Holds(0, loc(comb1), home1);
Holds(0, drunk(boris), false);
Holds(0, loc(erik), home2);
Holds(0, loc(comb2), home2);
Holds(0, drunk(erik), false);
Holds(0, loc(dimit), home3);
Holds(0, loc(comb3), home3);
Holds(0, drunk(dimit), true);
Holds(0, loc(sas609), run609);

// Action occurrences
:b occ Occurs(1, 2, put(boris, comb1, pocket1));
:b occ Occurs(1, 2, put(erik, comb2, pocket2));
:b occ Occurs(2, 4, travel(dimit, home3, office));
:b occ Occurs(3, 5, travel(boris, home1, office));
:b occ Occurs(4, 6, travel(erik, home2, office));
:b occ Occurs(6, 7, put(boris, gun, pocket1));
:b occ Occurs(5, 7, travel(dimit, office, airport));
:b occ Occurs(7, 9, travel(erik, office, airport));
:b occ Occurs(8, 10, travel(boris, office, airport));
:b occ Occurs(9, 10, board(dimit, sas609));
:b occ Occurs(10, 11, board(boris, sas609));
:b occ Occurs(11, 12, board(erik, sas609));
:b occ Occurs(13, 16, fly(sas609, run609, run609b));

// #acs put(person, pthing, pocket)
:b asc forall t1, t2, pe, pt, po [ Occurs(t1, t2, put(pe, pt, po)) & t2 > t1 ->
  (forall l [ Holds(t1, loc(pe), 1) <-> Holds(t1, loc(pt), 1) ]) &
  (forall t3 [ t3 > t1 & t3 <= t2 -> Occclude(t3, inpocket(pe, pt)) ]) &
  Holds(t2, inpocket(pe, pt), true)];
// #acs travel(person, location, location)
:b acs forall t1, t2, pe, l1, l2 [ Occurs(t1, t2, travel(pe, l1, l2)) & t2 > t1 ->
  Holds(t1, loc(pe), l1) &
  Holds(t2, loc(pe), l2) & Occlude(t2, loc(pe)) ];

// #acs board(person, airplane)
:b acs forall t1, t2, pe, a [ Occurs(t1, t2, board(pe, a)) & t2 > t1 ->
  (Holds(t1, poss-board(pe, a), true) & Holds(t1, loc(pe), airport) ->
   Occlude(t2, loc(pe)) &
   (forall 1 [ Holds(t2, loc(a), l) -> Holds(t2, loc(pe), l) ] &
    Holds(t2, onplane(pe, a), true)) ];

// #acs fly(airplane, runway, runway)
:b acs forall t1, t2, a, r1, r2 [ Occurs(t1, t2, fly(a, r1, r2)) & t2 > t1 ->
  Holds(t1, loc(a), r1) &
  (forall t3 [ t3 > t1 & t3 < t2 ->
    Occlude(t3, loc(a)) & Holds(t3, loc(a), air) ] &
    Occlude(t2, loc(a)) & Holds(t2, loc(a), r2) ];

/// Dependency constraints
// If a person has a gun, it is not possible for
// the person to board an airplane.
:b acs forall t, pe [ Holds(t, inpocket(pe, gun), true) ->
  forall a [ Occlude(t, poss-board(pe, a)) &
    Holds(t, poss-board(pe, a), false) ] ];

// If a person is drunk, it may or may not be
// possible for him to board an airplane.
:b acs forall t, pe [ Holds(t, drunk(pe), true) ->
  forall a Occlude(t, poss-board(pe, a)) ];

// If a person moves, all things in his pocket move.
:b acs forall t, pe, pt, l [
  Holds(t, inpocket(pe, pt), true) & Ct(t, loc(pe), l) ->
  Occlude(t, loc(pt)) & Holds(t, loc(pt), l) ];

// If an airplane moves, all person on board the airplane move.
:b acs forall t, a, p, l [  
    Holds(t, onplane(p, a), true) & Ct(t, loc(a), l) ->  
    Occlude(t, loc(p)) & Holds(t, loc(p), l) ];

/// Domain constraints  
// One person can not be in two different pockets  
forall t, pt1, pel, pe2 [  
    pel != pe2 & Holds(t, inpocket(pel, pt1), true) ->  
    Holds(t, inpocket(pe2, pt1), false) ];

// One person can not be on board two different airplanes  
forall t, pel, a1, a2 [  
    al != a2 & Holds(t, onplane(pel, a1), true) ->  
    Holds(t, onplane(pel, a2), false) ];

/// Persistence etc.  
forall t, th Per(t, loc(th));  
forall t, pe, pt Per(t, inpocket(pe, pt));  
forall t, p Per(t, drunk(p));  
forall t, a, p Per(t, onplane(a, p));  
// poss-board is durational.  
forall t, a, p Dur(t, poss-board(p, a), true);


P. Doherty and W. Lukaszewicz. Circumscribing features and fluents: A fluent logic for reasoning about action and change. In Proceedings of the 8th International Sym-


Automated planning as a satisfiability problem is a method developed in the early nineties. It has some known disadvantages, such as its inefficient encoding of numbers. The field of Satisfiability Modulo Theories tries to connect already established solvers for e.g. linear constraints into SAT-solvers in order to make reasoning about numerical values more efficient.

This thesis combines planning as satisfiability and SMT to perform efficient reasoning about actions that occupy realistic time in Temporal Action Logic, a formalism developed at Linköping University for reasoning about action and change.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.op.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.op.liu.se/

© Magnus Hallin
Linköping, November 27, 2011