Rule-based constraint propagation: theory and applications
Brand, S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Constraint-Based Modal Satisfiability Checking

7.1 Introduction

Relational structures, such as trees, graphs, transition systems, often provide a natural way to model evolving systems. One may have to deal with such relational structures for a variety of reasons, e.g. to evaluate queries, to check requirements, or to make implicit information explicit. Modal and modal-like logics such as temporal logic and description logic provide a convenient and computationally well-behaved formalism to represent such reasoning [Blackburn et al., 2001, Halpern et al., 2001].

A wide range of initiatives aimed at developing and refining algorithms for solving the satisfiability problem of basic modal logic has taken place in the past decade, driven by an increased computational usage of modal-like logics. These efforts have resulted in a series of implementations. Some implement special-purpose algorithms for modal logic, others exploit existing tools for example for propositional or first-order logic through some encoding. We follow here the second approach. The modal satisfiability problem is modelled and solved as a sequence of constraint satisfaction problems.

Specifically, we stratify a modal satisfiability problem (PSPACE-complete), into layers of constraint satisfaction problems (individually NP-complete). We refine the model substantially by exploiting the restricted syntactic nature of modal problems and the expressive power of constraints — in particular, we use not only the Boolean values. The resulting constraint satisfaction problems can be solved by a moderately expressive constraint solving system. Most of the constraints in our model are simple ones (e.g., at_most_one) for which propagation algorithms are part of many current constraint solving systems. Hence we can solve the modal satisfiability problem essentially by controlling a standard constraint solver. We demonstrate this point by an implementation in the constraint solving platform ECL'PS\textsuperscript{e} system [Wallace et al., 1997]. While it cannot yet fully
Chapter 7. Constraint-Based Modal Satisfiability Checking

compete with today's highly optimised modal provers, our experimental evaluations suggest that the approach is very promising in general. Moreover, it is excellent in some cases.

The main contributions of our work derive from our modelling of modal satisfiability problems: modal formulas are translated into layers of finite constraint problems that have non-Boolean domains, i.e. with further values than 0 or 1, together with appropriate constraints to reason about these values. We show that our modelling has a number of benefits over existing encodings of modal formulas into sets of propositions. For instance, the extended domains together with appropriate constraints give us a better control over the modal search procedure. They allow us to set strategies on the variables to split on in the constraint solver in a compact way. Specifically, by means of appropriate constraints for our model, we can obtain satisfying partial Boolean assignments instead of total assignments.

Background

We have a broad view of what modal logic is. In this view, modal logic encompasses such formalisms as temporal logic, description logic, feature logic, dynamic logic... While originating from philosophy, for the past three decades the main innovations in the area of modal logic have come from computer science and artificial intelligence. The modern, computationally motivated view of modal logic is one that takes modal logics to be expressive, yet computationally well-behaved fragments of first-order or second-order logic. Other computer science influences on modal logic include the introduction of many new formalisms, new algorithms for deciding reasoning tasks, and, overall, a strong focus on the interplay between expressive power and computational complexity. We now give some examples of modern computational uses of modal-like logics.

We start with a brief look at the use of modal-like logics in the area of formal specification and verification; a comprehensive introduction is provided by [Huth and Ryan, 1999]. Requirements such as “the system is always dead-lock free” or “the system eventually waits for a signal” can be compactly expressed in the basic modal logic by augmenting propositional logic with two operators: □ for the guarded universal quantifier over states (commonly read as always, meaning “in all the reachable states”), and ◇ for its existential counterpart (commonly read as eventually, meaning “in some reachable state”). If we formalise the statement “the system is dead-lock free” with the proposition s_free, and “the system waits for a signal” with s_wait, then the two requirements mentioned above correspond to the modal formulas □s_free and ◇s_wait, respectively.

Multi-modal logics are popular in the agent-based community, see e.g. [Rao and Georgeff, 1998]. Each agent is endowed with beliefs and knowledge, and with goals that it needs to meet. The beliefs and knowledge can be expressed by means of multi-modal operators: □_A for “agent A believes” and ◇_B for “agent B disbelieves”; □_B for “agent B knows” and ◇_A for “agent A ignores”.
More complex modal formulas involving until operators or path quantifiers are used to reason about plans of agents, in particular to express and verify specifications on plans, see e.g. [Bacchus and Kabanza, 2000], or extended goals; see [Pistore and Traverso, 2001] for example.

Description logics are a family of modal-like logics that are used to represent knowledge in a highly structured form, using (mostly) unary and binary relations on a domain of objects [Baader et al., 2003]. Knowledge is organised in terminological information (capturing definitions and structural aspects of the relations) and assertional information (capturing facts about objects in the domain being modelled). For instance, an object satisfies $\Diamond_R A$ if it is $R$-related to some object satisfying $A$. In the area of description logic, a range of algorithms for a wide variety of reasoning tasks has been developed.

Many more areas exist in which modal-like logics are currently being used, including semi-structured data [Marx, 2004], game theory [Harrenstein et al., 2002], or mobile systems [Cardelli and Gordon, 2000]. What all of these computational applications of modal-like logics have in common is that they use relational structures of one kind or another to model a problem or domain of interest, and that a modal-like logic is used to reason about these structures. For many of the applications mentioned here, modal satisfiability checking — does a given modal formula have a model (an assignment to the variables) — is the appropriate reasoning task.

**Related work**

The past decade has seen a wide range of initiatives aimed at developing, refining, and optimising algorithms for solving the satisfiability problem of basic modal logic. Some of these implement special purpose algorithms for modal logic, such as DLP [Patel-Schneider, 2002], FaCT [Horrocks, 2002], RACER [Haarslev and Möller, 2002], *SAT [Tacchella, 1999], while others exploit existing tools or provers for first-order logic, e.g. MSPASS [MSPASS, 2001], or propositional logic, for instance KSAT [Giunchiglia and Sebastiani, 2000], KBDD [Pan et al., 2002], through some encoding. In this work we follow the latter approach: we propose to model and solve modal satisfiability problems as constraint problems.

The starting-points of our work are [Giunchiglia and Sebastiani, 2000] and [Areces et al., 2000]. In [Giunchiglia and Sebastiani, 2000], modal formulas are modelled as sets of propositions (i.e. Boolean formulas) stratified into layers. The propositions are processed starting from the top layer in a depth-first left-most way, and solved by a propositional solver.

We add a refinement that builds on ideas due to [Areces et al., 2000]. There, an improvement of an existing encoding of modal formulas into first-order formulas was introduced. It enables one to re-use existing first-order theorem provers for deciding modal satisfiability, and, at the same time, to inform the prover about
Chapter 7. Constraint-Based Modal Satisfiability Checking

the restricted syntactic nature of first-order translations of modal formulas. This technique resulted in a significant improvement in performance.

We build on this insight. We improve on the modelling of modal formulas with respect to [Giunchiglia and Sebastiani, 2000] so as to be able to make efficient use of existing constraint solvers to decide modal satisfiability. Specifically, modal formulas are translated into layers of finite constraint satisfaction problems that have domains with non-Boolean values together with appropriate constraints.

While the well-known DPLL algorithm can also return partial Boolean assignments for propositions, there are two key add-ons of our modelling in this respect. First, the use of extended domains and constraints allows more control over the partial assignments returned by the constraint solver than unit propagation allows in DPLL. Second, we can run any constraint solver on top of our modelling to obtain partial assignments. It is by modelling that we obtain partial assignments, and not by modifying existing constraint solvers nor by choosing a specialised solver.

7.2 Propositional Formulas as Constraint Satisfaction Problems

We begin by making a relation between propositional logic formulas and constraints.

7.2.1 Propositions

A propositional formula $\phi$ is a term constructed from propositional (Boolean) variables (i.e., variables with domain $\{0, 1\}$) and the propositional connectives such as $\neg, \land, \lor, \rightarrow$, with the usual interpretation. A positive literal is a propositional variable, a negative literal is a negated variable. When a Boolean-valued assignment $\mu$ satisfies a propositional formula $\phi$, we write $\mu \models \phi$. We denote by $CNF(\phi)$ the result of ordering the propositional variables in $\phi$ and transforming $\phi$ into conjunctive normal form (CNF): i.e., a conjunction of disjunctions of literals without repeated occurrences. A clause of $\phi$ is a conjunct of $CNF(\phi)$.

Propositions as CSPs

It is straightforward to transform a propositional formula into a CSP that is satisfiable exactly if the formula is. First the formula is transformed to CNF. Then each resulting clause is viewed as a constraint. For example, the CNF formula

$$\phi = (\neg x \lor y \lor z) \land (x \lor \neg y)$$

(7.1)
corresponds to the CSP

\[ \mathcal{P}_\phi = \{C_1(x, y, z), C_2(x, y) \mid x, y, z \in \{0, 1\}\} \]

in which the constraint \(C_1\) forbids the assignment \(\{x \mapsto 1, y \mapsto 0, z \mapsto 0\}\); and the constraint \(C_2\) disallows the assignment \(\{x \mapsto 0, y \mapsto 1\}\). The relation of propositional formulas and CSPs is studied extensively in [Walsh, 2000].

### 7.2.2 Partial Assignments

A constraint solver presented with a formula encoded as a CSP will return a total assignment to the propositional variables. In contrast, we are here interested in \textit{partial} propositional assignments to the variables. For example, the assignment

\[ \{x \mapsto 1, z \mapsto 1\} \]

satisfies the formula \(\phi\) in (7.1) but is silent about the variable \(y\).

One way to get such partial but satisfying propositional assignments, without modifying the underlying constraint solver, is to encode the propositional formula into a CSP with an extra value beside the Boolean 0 and 1. We use the additional value "u" to mark those propositional variables that are not required in an assignment satisfying the encoded propositional formula. So we encode proposition \(\phi\) from (7.1) into a CSP \(\mathcal{P}_\phi\) in such a way that

\[ \{x \mapsto 1, y \mapsto u, z \mapsto 1\} \]

is a solution to \(\mathcal{P}_\phi\), from which in turn we obtain the desired partial satisfying assignment above.

Let us give a precise definition of the new encoding. We assume from now on an implicit total order on the propositional variables; it lets us ignore the order of occurrence of variables in a clause, (e.g., we do not distinguish \(x \lor y\) and \(y \lor x\)).

#### 7.2.1. Definition.

Given a propositional formula \(\psi\), we denote by \(\text{CSP}(\psi)\) the CSP \(\mathcal{C}; X \in \mathcal{D}\) associated with it. It is defined as follows:

1. \(X\) is the ordered sequence of propositional variables occurring in \(\psi\).
2. A domain \(D_i = \{0, 1, u\}\) is associated with each \(x_i\) in \(X\).
3. For each clause \(\theta\) in \(\text{CNF}(\psi)\), a constraint \(C_\theta\) exists that is on the variables \(Y = y_1, \ldots, y_m\) occurring in \(\theta\). A tuple \(d = d_1, \ldots, d_m\) from the product of the domains of \(Y\) satisfies \(C_\theta\) if some variable \(y_k\) exists such that
   - \(d[y_k] = 1\) if \(y_k\) occurs positively in \(\theta\),
   - \(d[y_k] = 0\) if \(y_k\) occurs negatively in \(\theta\).
We give no further requirements in this definition on how constraints are represented and implemented; on purpose, as such detail is not necessary for the theoretical results concerning the modal satisfiability solver. Nevertheless, some modelling choices and implementation details are discussed in Section 7.4 below.

The modelling of propositional formulas as in Definition 7.2.1 allows us to make any complete solver for finite CSPs return a partial Boolean assignment that satisfies a propositional formula \( \psi \) if \( \psi \) is satisfiable.

We denote by \( \mu|_{\text{Bool}} \) the Boolean sub-assignment of \( \mu \), that is, the set \( \{ x \mapsto b \mid (x \mapsto b) \in \mu \text{ and } b \in \{0, 1\} \} \).

**7.2.2. Theorem.** Consider a propositional formula \( \psi \) and let \( X \) be its ordered sequence of variables.

1. a total assignment \( \mu \) for \( \text{CSP}(\psi) \) satisfies \( \text{CSP}(\psi) \) if and only if \( \mu|_{\text{Bool}} \) satisfies \( \psi \);

2. \( \psi \) is satisfiable if and only if a complete constraint solver returns a total assignment \( \mu \) for \( \text{CSP}(\psi) \) such that \( \mu|_{\text{Bool}} \) satisfies \( \psi \).

**Proof.** First notice that a proposition and its CNF are equivalent: a Boolean assignment satisfies one exactly if it satisfies the other. Item 1 follows from this fact, Definition 7.2.1, and the following property of CNF formulas: a partial Boolean assignment \( \mu \) satisfies \( \text{CNF}(\psi) \) exactly if, for each clause \( \phi \) of \( \text{CNF}(\psi) \), \( \mu \) assigns 1 to at least one positive literal in \( \phi \), or 0 to at least one negative literal in \( \phi \). Item 2 follows from the former.

It is sufficient that each domain \( D_i \) of \( \text{CSP}(\psi) \) contains the Boolean values 0 and 1 for the above result to hold. Thus, one could have values other than \( v \) (and 0 and 1) in the CSP modelling to mark some variables with different 'levels of relevance' for deciding the satisfiability of a formula. The choice for just one non-Boolean value as in Definition 7.2.1 suffices for our purposes.

### 7.3 Modal Formulas as Layers of Constraint Satisfaction Problems

In this section we recall the basics of modal logic and provide a link between solving modal satisfiability and CSPs.

#### 7.3.1 Modal Formulas as Layers of Propositions

We refer to [Blackburn et al., 2001] for extensive details on modal logic. To simplify matters, we will focus on the basic mono-modal logic \( \mathcal{K} \), although our approach can easily be generalised to a multi-modal version.
Modal formulas. $\mathcal{K}$-formulas are defined as follows. Let $P$ be a finite set of propositional variables. Then $\mathcal{K}$-formulas over $P$ are produced by the rule

$$
\phi \ ::= \ p \mid \neg \phi \mid \phi \land \phi \mid \Box \phi
$$

where $p \in P$. The formula $\Diamond p$ abbreviates $\neg \Box \neg p$, and the other Boolean connectives are defined in terms of $\neg, \land$ as usual. A formula of the form $\Box \phi$ is called a box formula.

Here and in the remainder, we always assume that $P$ is implicitly ordered to avoid modal formulas only differing in the order of their propositional variables. Furthermore, standard propositional simplifications such as the removal of double occurrences of $\neg$ are implicitly performed on modal formulas.

Modal Layers and Propositional Approximations

The satisfiability procedure for $\mathcal{K}$-formulas that we develop in this section revolves around two main ideas:

- the stratification of a modal formula into layers of formulas of decreasing modal depth;
- the approximation and solving of such formulas as propositions.

Let us make these ideas precise, starting with the former. In words, the modal depth of a formula measures the maximum number of nested boxes. Formally, the modal depth of a formula $\phi$ is defined by

$$
\begin{align*}
\text{modal\_depth}(p) & = 0, \\
\text{modal\_depth}(\neg \phi) & = \text{modal\_depth}(\phi), \\
\text{modal\_depth}(\phi_1 \land \phi_2) & = \max(\text{modal\_depth}(\phi_1), \text{modal\_depth}(\phi_2)), \\
\text{modal\_depth}(\Box \phi) & = \text{modal\_depth}(\phi) + 1.
\end{align*}
$$

For instance, we have $\text{modal\_depth}(\Diamond \Box p \lor \Box \neg q) = 2$.

Testing if a modal formula is satisfiable involves stratifying it into layers of subformulas (or Boolean combinations of these) of decreasing modal depth. At each such layer, modal formulas are approximated and solved as propositions. Formally, the propositional approximation $\text{Prop}(\phi)$ of a formula $\phi$ is the propositional formula defined inductively by

$$
\begin{align*}
\text{Prop}(p) & = p \\
\text{Prop}(\neg \phi) & = \neg \text{Prop}(\phi) \\
\text{Prop}(\phi_1 \land \phi_2) & = \text{Prop}(\phi_1) \land \text{Prop}(\phi_2) \\
\text{Prop}(\Box \phi) & = x[\Box \phi].
\end{align*}
$$
We denote here by $x[\Box \phi]$ a fresh propositional variable that is associated with one specific occurrence of $\Box \phi$. Different occurrences of $\Box \phi$ lead to different variables which are distinguished by an index.

For instance, the formula $\phi = p \wedge \Box q \vee \neg \Box q$ is approximated by the propositional formula $Prop(\phi) = p \wedge x_1[\Box q] \vee \neg x_2[\Box q]$. The variables of $\phi$ are $\{p, q\}$ while the variables of $Prop(\phi)$ are $\{p, x_1[\Box q], x_2[\Box q]\}$.

### 7.3.2 $\mathcal{K}$-satisfiability and the $k_{\text{sat}}$ Schema

We formalise here $\mathcal{K}$-satisfiability, and present the general algorithm schema $k_{\text{sat}}$ for deciding the satisfiability of $\mathcal{K}$-formulas. It is given in Fig. 7.1. The $k_{\text{sat}}$ algorithm schema is the base of the KSAT algorithm of [Giunchiglia and Sebastiani, 2000].

**$\mathcal{K}$-satisfiability**

At this point we have to make a choice between a standard characterisation of the semantics of $\mathcal{K}$-formulas, and one closer to the semantics of the solving algorithm. We choose for the latter as this allows us to come more quickly and concisely to the matter of interest. For the standard characterisation we refer to [Blackburn et al., 2001], for example.

**7.3.1. Definition.** The $\mathcal{K}$-formula $\phi$ is $\mathcal{K}$-satisfiable if

- a Boolean assignment $\mu$ exists that satisfies $Prop(\phi)$,
- for every variable $x[\Box \lambda]$ of $Prop(\phi)$ such that $x[\Box \lambda] \mapsto 0$ is in $\mu$, the $\mathcal{K}$-formula
  $$\neg \lambda \wedge \bigwedge \{ \theta | \mu(x[\Box \theta]) = 1 \}$$
  is $\mathcal{K}$-satisfiable.

We use here, and from now on, a simpler notation for the conjunction of the elements of a set, namely we write

$$\bigwedge S$$

to abbreviate

$$\bigwedge_{\theta \in S} \theta$$

and do so analogously for a disjunction.
7.3. Modal Formulas as Layers of Constraint Satisfaction Problems

\[\text{sat} : \text{propositional formula} \mapsto \text{satisfying assignment or failure}\]

If the formula is propositionally satisfiable, then return a Boolean assignment. Return alternatives on backtracking.

\[\text{k_sat} : \text{modal formula} \psi \mapsto \text{succeeds if} \psi \text{ satisfiable}\]

\[
\begin{align*}
\mu &:= \text{sat}(\text{Prop}(\psi)) \quad // \text{create a choice point} \\
B^+ &:= \{ \theta \mid x[\Box \theta] \mapsto 1 \text{ in } \mu \} \\
B^- &:= \{ \lambda \mid x[\Diamond \lambda] \mapsto 0 \text{ in } \mu \} \\
\Theta &:= \bigwedge B^+ \\
\text{for each} &\quad \lambda \in B^- \quad \text{do} \\
&\quad \text{ksat}(\Theta \wedge \neg \lambda) \quad // \text{backtrack if this fails}
\end{align*}
\]

Figure 7.1: The \textit{k_sat} algorithm schema.

The \textit{k_sat} Algorithm Schema

In the \textit{k_sat} schema, provided in Fig. 7.1, the \textit{sat} procedure determines the satisfiability of the propositional approximation of \(\phi\) by returning a Boolean assignment \(\mu\) as in Definition 7.3.1. Alternative satisfying assignments are generated upon backtracking. If there is no alternative assignment, then the call to \textit{k_sat} fails at this level and backtracking takes place, unless it is the top level in which case it is reported that the formula is unsatisfiable.

In this way, the modal search space gets stratified into modal formulas of decreasing modal depth and explored depth-first. A variable of the form \(x[\square \lambda]\) to which \(\mu\) assigns 0 means that we must “open the box” and check \(\lambda\) against all the formulas \(\theta\) that come with variables of the form \(x[\Box \theta]\) to which \(\mu\) assigns 1. Exactly one proposition is so created and tested satisfiable.

7.3.2. \textbf{THEOREM.} In the \textit{k_sat} algorithm schema given in Figure 7.1, if \textit{sat} is a complete solver for Boolean formulas, then \textit{k_sat} is a decision procedure for \(\mathcal{K}\)-satisfiability.

\textbf{PROOF.} Correctness and completeness of \textit{k_sat} is entailed by the characterisation of \(\mathcal{K}\)-satisfiability in Definition 7.3.1. \textit{k_sat} terminates since the modal depth and the number of propositional variables of a modal formula are bounded. \(\square\)

7.3.3 The KCSP Algorithm

We now devise a modal decision procedure based on the \textit{k_sat} schema, parameterised by a constraint solver as the underlying propositional solver \textit{sat}. We first
provide the intuition by an example.

7.3.3. **EXAMPLE.** Consider the modal formula

\[ \phi = \neg \Box (p \lor \bot) \land (\Box r \lor \Box p). \]

(The symbol \( \bot \) abbreviates the 'always false' formula and could be defined as \( p' \land \neg p' \) for some arbitrary \( p' \in P \).) The propositional approximation \( \text{Prop}(\phi) \) of \( \phi \), can be turned into a CSP according to Def. 7.2.1. We obtain

- three variables \( x[\Box (p \lor \bot)] \), \( x[\Box r] \), and \( x[\Box p] \), each with domain \( \{0, 1, u\} \),
- two constraints,
  - one for \( \Box (p \lor \bot) \), forcing the assignment 0 to \( x[\Box p \lor \bot] \),
  - one for \( (\Box r \lor \Box p) \), requiring 1 to be assigned to \( x[\Box r] \) or \( x[\Box p] \).

Assigning the value \( u \) to a variable means not committing to any decision concerning its Boolean values, 0 and 1. This CSP is given to the constraint solver, which may return the assignment

\[ \mu_1 = \{ x[\Box (p \lor \bot)] \mapsto 0, \]
\[ x[\Box r] \mapsto u, \]
\[ x[\Box p] \mapsto 1 \}. \]

Then, for all the variables \( x[\Box \ldots] \) to which \( \mu_1 \) assigns 1 (in this case only \( x[\Box p] \)), the formulas within the scope of \( \Box \) are joined in a conjunction \( \Theta \). So we have

\[ \Theta = p. \] (UT)

Then all the box variables to which \( \mu_1 \) assigns 0 are considered, in this case only \( x[\Box (p \lor \bot)] \). Thus, the formula \( p \lor \bot \) is negated, simplified (translated in CNF when needed), and we obtain

\[ \neg \lambda = \neg p. \] (ET)

The conjunction \( \Theta \land \neg \lambda \) is given to the sat solver. In this case, the clause passed on is \( p \land \neg p \). It is translated into a new CSP and given to the constraint solver, which results in failure due to its inconsistency.

On the subsequent backtracking, we may obtain the alternative assignment

\[ \mu_2 = \{ x[\Box (p \lor \bot)] \mapsto 0, \]
\[ x[\Box r] \mapsto 1, \]
\[ x[\Box p] \mapsto u \}. \]

This assignment leads to \( \Theta = r \), and thus to the formula \( r \land \neg p \) which is successfully tested satisfiable. In turn, satisfiability of \( \phi \) is concluded.
7.4. Constraint-Based Modelling

Notice the key points about (UT) and (ET): we only consider the box variables \( x[\Box \ldots] \) to which a Boolean value, 0 or 1, is assigned. The box variables to which \( u \) is assigned are disregarded, safely so because of Theorem 7.2.2. We show in the following section that the availability of values other than 0 and 1 has a number of advantages.

7.3.4. Definition. The KCSP algorithm is defined as follows. In the \( k\text{-sat} \) schema, the sat function whose input is a formula \( \phi \), is instantiated with a complete solver for finite CSPs whose input is \( CSP(Prop(\phi)) \).

We can state the following by Theorems 7.2.2 and 7.3.2.

7.3.5. Corollary. KCSP is a decision procedure for \( \mathcal{K} \)-satisfiability.

7.4 Constraint-Based Modelling

In this section we discuss the constraints into which we translate a modal formula. We begin with a base modelling, and proceed to an improved modelling that possesses some desirable properties.

When modelling a problem as a CSP for solving it in a propagation & search-based solver, one generally has two options for the user-defined constraints. Either one implements a custom-built constraint propagation procedure for such a constraint, or one rewrites it into constraints for which propagation algorithms are available. Obviously the latter approach is preferable, all other things being equal. We follow it here.

7.4.1 Base Modelling

The input to KCSP is a formula in conjunctive normal form. We translate it into a CSP clause-wise, each clause contributing one constraint.

Aspect 1: Clauses as Constraints

We distinguish four disjoint sets of variables in a clause: propositional variables and variables representing box formulas, and both subdivided according to polarity. We denote these sets \( P^+, P^-, B^+, B^- \), resp. Therefore, a clause can be written as

\[
\begin{align*}
\vee \{ p \mid p \in P^+ \} & \vee \{ \neg p \mid p \in P^- \} \\
\vee \{ x[\Box \phi] \mid x[\Box \phi] \in B^+ \} & \vee \{ \neg x[\Box \phi] \mid x[\Box \phi] \in B^- \}
\end{align*}
\]

Such a clause is now viewed as a constraint on the variables in the four sets,

\[
\text{clause\_constraint}(P^+, P^-, B^+, B^-).
\]
Chapter 7. Constraint-Based Modal Satisfiability Checking

It holds if *at least one* variable in the set \( P^+ \cup B^+ \) is assigned a 1, or one variable in \( P^- \cup B^- \) is assigned a 0; recall Definition 7.2.1. We decompose this constraint now in terms of a simpler constraint \(*at least one*, which is defined on a set of variables and parameterised by a constant, and which requires the latter to occur in the variable set. This constraint, or a related one, is available in many constraint programming systems. The constraint library of ECLIPS\(s^x\) contains a predefined constraint with the meaning of \(*at most one*, which can be employed to imitate \(*at least one*.

Thus, as the first step to obtain a method for propagation of clause constraints, we reformulate \(*clause constraint* as the disjunctive constraint

\[
*at least one*(P^+ \cup B^+, 1) \lor *at least one*(P^- \cup B^-, 0).
\]

**Aspect 2: Disjunctions as Conjunctions**

Propagating disjunctive constraints, if supported at all, is generally difficult for constraint solvers. It is therefore preferable to avoid them when modelling, and in our situation that can be done rather elegantly. The disjunctive clause constraint is transformed into a conjunction with the help of a single auxiliary link variable. We obtain

\[
*at least one*(P^+ \cup B^+ \cup \{\ell\}, 1) \land *at least one*(P^- \cup B^- \cup \{\ell\}, 0).
\]

The link variable \( \ell \in \{0, 1\} \) selects implicitly which of the two constraints must hold. For example, observe that \( \ell = 0 \) selects the constraint on the left: it forces \(*at least one*(P^+ \cup B^+, 1) \) and satisfies \(*at least one*(P^- \cup B^- \cup \{0\}, 0).

It is useful to remark that this way of rewriting the clause constraint does not hinder constraint propagation in the sense that if GAC is established on the two \(*at least one* constraints separately then the clause constraint is GAC; by Lemma 2.1.7 and since the two conjuncts share no variables except \( \ell \).

### 7.4.2 Advanced Modelling

We extend the base constraint model so as to generate partial assignments, avoid CNF conversion, and take into account multiple occurrences of box formulas.

**Aspect 3: Partial Assignments by Constraints**

While any solution of the CSP induced by a formula at some layer satisfies the formula at that layer, it is useful to obtain satisfying partial Boolean assignments that mark as irrelevant as many box formulas in this layer as possible. This causes fewer subformulas to enter the propositions generated for the subsequent layer. We use the extra value \( u \) to mark irrelevant variables.
7.4.1. **EXAMPLE.** Take the clause $p \lor q \lor r \lor s$. The corresponding clause constraint is on the four groups of variables $P^+ = \{p\}$, $P^- = \{q\}$, $B^+ = \{x[r]\}$, and $B^- = \{x[s]\}$, that is (in simpler notation),

$$\text{clause}\_\text{constraint}(p, q, x[r], x[s]).$$

Some solutions to this constraint are better than others. For instance,

- $\mu_1 = \{p \leftarrow 0, q \leftarrow 0, x[r] \leftarrow u, x[s] \leftarrow u\}$,
- $\mu_2 = \{p \leftarrow 0, q \leftarrow 1, x[r] \leftarrow u, x[s] \leftarrow u\}$,
- $\mu_3 = \{p \leftarrow 0, q \leftarrow 0, x[r] \leftarrow u, x[s] \leftarrow u\}$,
- $\mu_4 = \{p \leftarrow 0, q \leftarrow 1, x[r] \leftarrow 1, x[s] \leftarrow u\}$,
- $\mu_5 = \{p \leftarrow 0, q \leftarrow 1, x[r] \leftarrow u, x[s] \leftarrow 0\}$,

are 'good' solutions since they assign a Boolean value to as few variables representing box formulas as possible. Note that, within these five assignments, $\mu_1, \mu_2, \mu_3$ are preferable, but since $p, q$ may also be constrained by other clauses, we must admit $\mu_5, \mu_6$ as well.

In contrast, for example

- $\mu_6 = \{p \leftarrow 1, q \leftarrow 0, x[r] \leftarrow 0, x[s] \leftarrow u\}$,
- $\mu_7 = \{p \leftarrow 0, q \leftarrow 1, x[r] \leftarrow 1, x[s] \leftarrow 0\}$,

are undesirable, as box formulas are unnecessarily marked to be considered in the next layer.

We formalise this observation.

7.4.2. **FACT.** Consider a CSP containing a clause constraint $C$ on the propositional variables $P = P^+ \cup P^-$ and the variables representing box formulas $B = B^+ \cup B^-$. Suppose $\mu$ is a partial assignment that is not on the variables $P \cup B$.

- The assignment $\mu' = \mu \cup \{p \leftarrow 1\} \cup \{x[r] \leftarrow u \mid x[r] \in B\}$, where $p \in P^+$, satisfies $C$. (Analogously for $p \in P^-$.)

- The assignment $\mu' = \mu \cup \{x[r] \leftarrow 1\} \cup \{x' \leftarrow u \mid x' \in B - \{x[r]\}\}$, where $x[r] \in B^+$, satisfies $C$. (Analogously for $x[r] \in B^-$.)

If $\mu$ can be extended to a total assignment satisfying the CSP than also $\mu'$ can be so extended. \(\square\)

Note that the variables in $B$ are constrained only by $C$, and recall Theorem 7.2.2.

In other words, if satisfying the propositional part of a clause suffices to satisfy the whole clause, then all box formulas in it can be marked irrelevant. Otherwise, all box formulas except one can be marked irrelevant.

Let us transfer this observation into a clause constraint model. First, we rewrite the base model so as to
Chapter 7. Constraint-Based Modal Satisfiability Checking

• separate the groups of variables (in propositional and box variables),
• and convert the resulting disjunctions into conjunctions, again with the help of extra linking variables.

Next, we replace the at_least_one constraint for variables representing box formulas by an exactly_one constraint. This simple constraint is commonly available as well; ECL\textsuperscript{PS}e offers the more general occurrences constraint, which forces a certain number of variables in a set to be assigned to a specific value. We obtain

\[
\text{clause\_constraint}(P^+, P^-, B^+, B^-) = \\
\text{at\_least\_one}(P^- \cup \{\ell^+_P\}, 1) \land \text{exactly\_one}(B^+ \cup \{\ell^-_B\}, 1) \land \\
\text{at\_least\_one}(P^- \cup \{\ell^-_P\}, 0) \land \text{exactly\_one}(B^- \cup \{\ell^-_B\}, 0) \land \\
\text{clause\_link}(\ell^-_P, \ell^-_B, \ell^+_P, \ell^+_B)
\]

The variable domains are: \(p \in \{0, 1\}\) for \(p \in P\), next \(x_B \in \{1, u\}\) for \(x_B \in B^+\), and \(x_B \in \{0, u\}\) for \(x_B \in B^-\).

The essential four linking variables are constrained as in the following logical formula or the equivalent table.

\[
\text{clause\_link}(\ell^+_P, \ell^-_P, \ell^-_B, \ell^+_B) = \\
(\ell^+_P = 1 \land \ell^-_P = 0) \iff (\ell^-_B = u \lor \ell^+_B = u) \land \\
\ell^-_B = 1 \lor \ell^+_B = 0
\]

\[
\begin{array}{cccc}
\ell^+_P & \ell^-_P & \ell^+_B & \ell^-_B \\
1 & 0 & 1 & u \\
1 & 0 & u & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
1 & 1 & 1 & 0
\end{array}
\]

Observe that the 5 tuples in the table correspond to the situations that we wish to permit — the clause is satisfied by either a positive or a negative box formula (but not both at the same time) or a positive or a negative propositional variable (maybe both at the same time). Compare also with \(\mu_{1,6}\) in Example 7.4.1.

ECL\textsuperscript{PS}e can accept the linking constraint in logical-operator form, in which case it is internally rewritten into several arithmetic constraints. Alternatively, we can compile the defining table into a set of membership rules, for instance by the method described in Section 5.4.4. Examples for the generated rules are

\[
\ell^+_P = 0 \implies \ell^-_B \neq u, \ell^+_B \neq u, \\
\ell_P = 0, \ell^-_B = 1, \ell^-_B = 0 \implies \ell^+_P \neq 1.
\]

We found in our experiments that the linking constraint, among all constraints, is the one whose propagation is executed most often. Hence propagating it efficiently is particularly relevant. Using the generated membership rules and the corresponding scheduler (Chapter 3.4.2) proved to be the fastest way of propagating the linking constraint of several methods we tested.
Constraint-Based Modelling

Aspect 4: A Negated-CN F Constraint

Except for the initial input formula to KCSP which is in conjunctive normal form, the input to an intermediate call to the sat function of KCSP (see the algorithm in Figure 7.1) has the form \( \Theta \land \neg \lambda \) where both \( \Theta \) and \( \lambda \) are formulas in CNF. A naive transformation of \( \neg \lambda \) into CNF will result in an exponential increase in the size of the formula. We deal with this problem by treating \( \neg \lambda \) as a constraint. The following holds.

7.4.3. FACT. The constraint \( \neg \lambda \) is satisfiable if and only if \( \lambda \) (which is a conjunction of clauses) has at least one unsatisfiable clause. \( \square \)

We formulate the constraint corresponding to \( \neg \lambda \) consequently as a disjunctive constraint, each disjunct standing for a negated clause. This disjunctive constraint is converted into a conjunction using a set \( L = \{ \ell_1, \ldots, \ell_m \} \) of linking variables, one for each of the \( m \) disjuncts. Every \( \ell_i \) ranges over \( \{1, u\} \). The case \( \ell_i = 1 \) means the \( i \)th disjunct holds, i.e., the \( i \)th clause in \( \lambda \) is unsatisfied. Instead of imposing at least one \( (L, 1) \) to select one disjunct to hold, however, we require exactly one \( (L, 1) \), in line with our goal of obtaining minimal partial Boolean assignments. \( \ell_i = u \) means the \( i \)th disjunct (clause) is irrelevant. We then forcibly mark all box formulas by \( u \), but ignore the propositional variables.

The definition of the negated-clause constraint on the variables in \( P^+, P^-, B^+, \) and \( B^- \), and the linking variable \( \ell_i \) in logical form is

\[
\text{negated_clause}(P^+, P^-, B^+, B^-, \ell_i) =
\]

\[
\ell_i = 1 \quad \rightarrow \quad ( \forall p \in P^+. p = 0 \land \forall p \in P^- . p = 1 ) \quad (NC_1)
\]

\[
\ell_i = 1 \quad \leftrightarrow \quad ( \forall b \in B^+. b = 0 \land \forall b \in B^- . b = 1 ) \quad (NC_{2a})
\]

\[
\ell_i = u \quad \leftrightarrow \quad ( \forall b \in B^+. b = u \land \forall b \in B^- . b = u ) . \quad (NC_{2b})
\]

Constraint Propagation

Let us sketch the process of developing a constraint propagation procedure for negated_clause as an example for a specialised constraint. Notice first that negated_clause consists of conjuncts \( NC_1 \) and \( NC_2 \). We discuss them separately.

From \( NC_1 \), we can immediately read off the membership rule

\[
\ell_i = 1 \quad \rightarrow \quad p \neq 1
\]

where \( p \in P^+ \) or \( p \in P^- \). The domain of \( p \) is \( \{0, 1\} \).

A membership rule as \( r_1 \) that is atomic (has just equalities in the condition) corresponds to a non-solution of the associated constraint; recall Note 5.3.4. If
we wish to obtain all correct atomic membership rules for a constraint, we can construct each atomic membership rule from each non-solution, and combine the rules. See the treatment of this issue in the context of incremental rule generation, in particular Section 5.4.3.

The (partial) non-solution \((1,1)\) for \(\langle \ell, p \rangle\) in \(NC_1\), which underlies the rule \(r_1\), admits exactly one other rule, namely

\[
p = 1 \rightarrow \ell_i \neq 1
\]

No other correct membership rule is possible for conjunct \(NC_1\).

Propagating the conjunct \(NC_2\) is substantially simplified by the constraint propagation rule

\[
NC_2 \rightarrow b_1 = b_2
\]

for all \(b_1, b_2 \in B^+\), or \(b_1, b_2 \in B^-\). In presence of equality constraint propagation, we need thus only deal with two representative variables \(b_1^+ \in B^+\), \(b_2^- \in B^-\), which are constrained with the linking variable by

\[
\langle \ell_i, b_1^+, b_2^- \rangle \in \{ (1, 0, 1), (u, u, u) \}.
\]

This restriction is just a simple extensionally defined constraint. Membership rules for it can be generated as described in Section 5.4.4, for example.

The thus developed propagation rules reflect the propagation algorithm for the negated_clause constraint as implemented in KCSP. It establishes generalised arc-consistency.

**Aspect 5: A Constraint for Factoring**

In our base model, we have treated and constrained the \(i\)th occurrence of a box formula \(\Box \phi\) as a distinct propositional variable \(x_i[\Box \phi]\). For instance, the two occurrences of \(\Box p\) in the formula \(\Box p \land \neg \Box p\) are treated as the two distinct propositional variables \(x_1[\Box p]\) and \(x_2[\Box p]\) in our base model.

We consider here the case that a box formula occurs several times, in several clauses, in any polarity. We prevent assigning conflicting values to different occurrences, by a suitable constraint.

Let us collect in the set \(B_{\Box \phi}\) all variables \(x_i[\Box \phi]\) representing the formula \(\Box \phi\) in the entire CSP. Recall that their domain is \(\{0, 1, u\}\). We state as a constraint that

\[
\forall x_1, x_2 \in B_{\Box \phi}. \ (\neg(x_1 = 1 \land x_2 = 0) \land \neg(x_1 = 0 \land x_2 = 1)).
\]

To see the effect, suppose there is a pair \(x_1, x_2 \in B_{\Box \phi}\) such that \(x_1 \mapsto 1\), \(x_2 \mapsto 0\) in a solution to the CSP without this factoring constraint. This means we obtain both \(\Box \phi \mapsto 1\) and \(\Box \phi \mapsto 0\) in the assignment returned, which results in an
unsatisfiable proposition $\phi \land \neg \phi \land \ldots$ being generated. The factoring constraint just detects such failures earlier. The straightforward modelling idea, namely using one unique variable for representing a box formula in all clauses (‘factoring out’ the formula), clashes with the assumption made for the other partial-assignment constraints that each box formula variable is unique.

Propagation rules for the factoring constraint can be derived in a similar way as for negated_clause, and lead to an implementation that establishes GAC.

7.5 Implementation and Experimental Assessment

Theoretical studies often do not provide sufficient information about the effectiveness and behaviour of complex systems such as satisfiability solvers and their optimisations. Empirical evaluations must then be used. In this section we provide an experimental comparison of our advanced modelling (Section 7.4.2) against the base model (Section 7.4.1), using the test developed in [Heuerding and Schwendimann, 1996].

We find that, no matter what other models and search strategies we commit to, we always get the best results by using constraints for partial assignments as in Section 7.4.2, Aspect 3. In the remainder of this paper, these are referred to as the assignment-minimising, or simply minimising, constraints. We show below how these minimising constraints allow us to better direct the modal search procedure.

We conclude this section by comparing the version of KCSP that features the advanced modelling with KSAT. The constraint solver that we use as the sat function in KCSP is a conventional one, based on search with chronological backtracking and constraint propagation. The propagation algorithms are specialised for their respective constraints and enforce generalised arc-consistency on them, as discussed in Section 7.4 above.

7.5.1 Test Environment

State of the Art

In the area of propositional satisfiability checking there is a large and rapidly expanding body of experimental knowledge; see, e.g. [Gent et al., 2000]. In contrast, empirical aspects of modal satisfiability checking have only recently drawn the attention of researchers. We now have a number of test sets, some of which have been evaluated extensively [Baader et al., 1992, Heuerding and Schwendimann, 1996, Giunchiglia and Sebastiani, 2000, Hustadt and Schmidt, 1997, Horrocks et al., 2000]. In addition, we also have a clear set of guidelines for performing empirical testing in the setting of modal logic [Heuerding and Schwendimann, 1996, Horrocks et al., 2000]. Currently,
there are three main test methodologies for modal satisfiability solvers, one based on hand-crafted formulas, the other two based on randomly generated problems.

To understand on what kinds of problems a particular prover does or does not do well, it helps to work with test formulas whose meaning can (to some extent) be understood. For this reason we opted to carry out our tests using the Heuerding and Schwendimann (HS) test set [Heuerding and Schwendimann, 1996], which was used at the TANCS '98 comparison of systems for non-classical logics [TANCS, 2000].

The HS Test Set

The HS test set consists of several classes of formulas for \( \mathcal{K} \) and other modal logics that we do not consider here. Some problem classes for \( \mathcal{K} \) are based on the pigeon-hole principle (ph) and a two-colouring problem on polygons (poly). We consider the classes branch, d4, dum, grz, lin, path, ph, poly, t4.

Each class is generated from a parameterised logical formula. This formula is either a \( \mathcal{K} \)-theorem, which is thus provable, or only \( \mathcal{K} \)-satisfiable, which is not provable. So the generated class contains either provable formulas or non-provable formulas, and is labelled accordingly by a suffix \(_{p}\) or \(_{n}\).

Some of these parameterised formulas are made harder by hiding their structure or adding extra material. The parameter allows for the creation of modal formulas in the same class but of differing difficulty. Specifically, the formulas in a class are constructed in such a way that the difficulty of proving them should be exponential in the parameter. It is hoped that this kind of increase in difficulty makes differences in the speed of the machines used to run the benchmarks less significant.

Benchmark Methodology

The benchmark methodology is to test formulas from each class starting with the easiest instance (controlled by the parameter), until the provability status of a formula can not be correctly determined within 100 CPU seconds. The test result of this class is the parameter of the largest formula that can be solved within this time limit. The parameter ranges from 1 (easiest) to 21 (hardest).

Implementation

We implemented the KCSP algorithm as a prototype in the constraint programming system ECL\textsuperscript{PC}S\textsuperscript{e}. The HS formulas are negated, reduced to CNF and translated into the format of KCSP.

We add heuristics to KCSP with minimising constraints in an attempt to reduce the depth of the KCSP search tree. The value \( u \) is preferred for box formulas, and among them for positively occurring ones. Furthermore, the instantiation
ordering of variables representing box formulas is along their increasing modal depth, e.g., $x\Box p$ is instantiated before $x\Box\Box p$.

### 7.5.2 Assessment

In this subsection we evaluate the contributions of the various aspects of our advanced modelling.

**Aspect 3: Partial assignments by constraints.** Do minimising constraints make a difference in practice? To address this question, we focus here on the *branch* formulas in the HS test set. This class of formulas is specifically relevant for automated modal theorem proving. Its non-provable variant *branch*\_\_n is recognised as the hardest class of “truly modal formulas” for today’s modal theorem provers [Horrocks et al., 2000]. They are the so-called Halpern and Moses branching formulas that “have an exponentially large counter-model but no disjunction […] and systems that will try to store the entire model at once will find these formulae even more difficult” [Horrocks et al., 2000].

Figure 7.2 plots the run times of KCSP on *branch* formulas, with and without minimising constraints. There is clearly a difference. KCSP with assignment-minimising constraints manages to solve 13 instances of *branch*\_\_n and all 21 of *branch*\_\_p (in less than 2 seconds). Without minimisation, in both cases only 2 instances are solved.

To understand the reasons for the good performance of KCSP with minimising constraints, consider *branch*\_\_p(3), unsolved by KCSP with total assignments. In KCSP with minimising constraints, there are two choices for box formulas at the root layer and none at the subsequent layer of modal formulas obtained by “opening” a box. This small factor results in a modal search tree of just two branches.
In contrast, with total assignments there are 6 extra box formulas at the root layer. The implied extra branching factor is $2^6 = 64$ at the root of the modal search tree only. All 6 box formulas are always carried over to subsequent layers, positively or negatively, adding to the work to be done there.

More generally, the superiority of KCSP with minimising constraints can be explained as follows: the tree-like model that the solver implicitly attempts to construct while trying to satisfy a formula is kept as small as possible by the minimising constraints. In this sense, constraints allow us better control over the modal search than, for instance, unit propagation allows for in DPLL.

Notice finally that the results of KCSP with minimising constraints on the branch class are competitive on the this class with the best optimised modal theorem provers such as *SAT and DLP.

**Aspect 4: Negated-CNF constraint.** In all the HS formula classes, using constraints in place of CNF conversions leaves equal or increases the number of decided formulas. Avoiding CNF conversion by means of negated-CNF constraints has a substantial effect for example in the case of $ph_n(4)$, an instance of the pigeon-hole problem, which is solved in a few seconds. In contrast, by requiring CNF conversion (even with minimising constraints), the execution of KCSP is terminated pre-emptively due to memory exhaustion.

However, the CNF conversion at the root level remains necessary, and prevents formulas $ph(k)$ with $k > 4$ in KCSP for lack of memory.

**Aspect 5: Factoring constraint.** This constraint avoids simple contradictory occurrences of a formula in the layers subsequent to the current one. We remark that this consideration of multiple occurrences of a subformula does not always provide a globally minimal number of box formulas with a Boolean value. Nevertheless, it is beneficial for formulas with the same variables hidden and repeated inside boxes. It proved useful in the cases $grz$, $d4$, $dum_p$, $path_p$, $t4_p$. In the remaining cases the contribution of factoring with constraints is insignificant, except for $path_n$ where searching for candidate formulas to be factored slightly slowed down the search.

**Formula simplifications.** As a preprocess to KCSP, the top-level input formula can be simplified to a logically equivalent formula. We use standard simplification rules for propositional formulas, at all layers, in a bottom-up fashion. In the same way, some modal equivalences such as $\neg \square T \equiv \bot$ are used.

Simplification in KCSP plays a relevant role in the case of lin formulas. Without simplifications and minimising constraints, KCSP takes longer than 5 minutes to return an answer for $lin_n(3)$. With simplifications and minimising constraints, the runtime is reduced to less than 0.4 seconds. By also adding factoring, KCSP solves the most difficult formula of $lin_n$ in 0.06 sec, that of $lin_p$ in 0.01 sec.
7.5. Implementation and Experimental Assessment

<table>
<thead>
<tr>
<th></th>
<th>branch</th>
<th>d4</th>
<th>dum</th>
<th>grz</th>
<th>lin</th>
<th>path</th>
<th>ph</th>
<th>poly</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>np</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
<td>n</td>
<td>p</td>
</tr>
<tr>
<td>KSAT</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>&gt;</td>
<td>11</td>
<td>&gt;</td>
<td>17</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>&gt;</td>
<td></td>
<td></td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>&gt;</td>
<td>6</td>
<td>9</td>
<td>19</td>
<td>12</td>
<td>&gt;</td>
<td>13</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>KCSP/sp</td>
<td>11</td>
<td>&gt;</td>
<td>6</td>
<td>8</td>
<td>17</td>
<td>11</td>
<td>&gt;</td>
<td>10</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 7.1: Results on the HS Benchmark

7.5.3 Results and a Comparison

We compare the performances of KSAT and KCSP on the HS test set in Table 7.1. Each column lists a formula class and the number of the most difficult formula decided within 100 CPU seconds per prover. We write > when all 21 formulas in the test set are solved within this time limit.

**First row: KSAT.** The results for KSAT are taken from [Horrocks et al., 2000]. They reflect a run of a C++ implementation of KSAT with the HS test set on a 350 MHz Pentium II with 128 MB of main memory.

**Second row: KCSP.** We used KCSP with all advanced aspects considered; i.e., partial assignments by constraints, negated-CNF constraints, factoring constraints, and formula simplifications. In the remainder, we refer to this as KCSP. The time taken by the translator from the HS format into that of KCSP is insignificant. The worst case among those in Table 7.1 took less than 1 second (and these timings are taken into account for the table entries). We ran our experiments on a 1.2 GHz AMD Athlon Processor with 512 MB RAM, under Red Hat Linux 8 and ECLIPSE 5.5.

**Third row: KCSP/sp.** To account partially for the different platforms used to run KSAT and KCSP on, we scaled the measured times of KCSP by a factor 350/1200, the ratio of the respective processor speeds. The results are reported in the line KCSP/sp, and *italicized* where different from KCSP.

**Result Analysis**

Some interesting similarities and differences in performance between KSAT and KCSP can be observed. For some formula classes, KCSP clearly outperforms KSAT, for some it is the other way round, and for others the differences do not seem to be significant.

For instance, KCSP performs better in the case of *lin* and *branch* formulas. As pointed out in Subsection 7.5.2, *branch_n* is the hardest "truly modal test class" for current modal provers, and KCSP with partial assignments performs well on this class. Now, similar to KCSP, KSAT features partial assignments in that its
underlying propositional solver is DPLL-based. The differences in performance are then due to our modelling. In more general terms:

- Extended domains and constraints allows for better control over the partial assignments to be returned by the adopted constraint solver than unit propagation allows for in DPLL.

- Constraints allow a compact representation of certain requirements such as that of reducing the number of variables representing box formulas to which a Boolean value is assigned.

The models that KCSP (implicitly) tries to generate when attempting to satisfy a formula remain very small. In the case of branch, searching for partial assignments with minimising constraints yields other benefits per se: the smaller the number of box formulas to which a Boolean value is assigned at the current layer, the smaller the number of propositions in the subsequent layer. In this way, fewer choice points and therefore fewer search tree branches are created. Thereby, the addition of constraints to limit the number of box formulas to reason on, while still exploring fully the purely propositional search space, seems to be a useful idea on the branch class.

In the cases of grz and t4, on the other hand, KSAT is superior to KCSP. KSAT features a number of optimisations (heuristics) for early modal pruning that are absent in KCSP, and these may be responsible for the better behaviour of KSAT on these classes.

We remark finally that KSAT is compiled C++ code while KCSP is interpreted ECL's Prolog code. This makes it interesting to see that KCSP competes well with KSAT.

### 7.6 Final Remarks

We described here a constraint-based approach to modal satisfiability testing. The reasoning process is split into sequences of propositional problems which are solved as separate constraint satisfaction problems. We showed the feasibility of our ideas by discussing an implementation and benchmark results.

Taking advantage of the expressiveness that constraint modelling affords, we integrate control knowledge when translating the propositions into CSPs. In particular, we observed that partial solutions to the propositional subproblems suffice and that some solutions are preferable to others. By using an extra non-Boolean domain value and additional constraints, we obtain preferred partial solutions by modelling instead of by modifying the solver. Such partial propositional assignments reduce substantially the branching factor in the tree-model that our solver implicitly tries to construct. The resulting reduction in search time is significant, as we found empirically.
The generated CSPs use several standard constraints, for which current constraint programming systems provide propagation algorithms. For some specialised constraints we implemented specific propagation algorithms that establish generalised arc-consistency. The underlying propagation mechanisms are derived elegantly as constraint propagation rules.

We conclude by pointing out some possible and interesting extensions.

The current modelling of propositional formulas is clause-based, assuming either CNF or negated-CNF format. A direct mapping to constraints of formulas in any format, i.e., using any Boolean operators, is certainly feasible, and would remove the initial CNF conversion required now. Achieving small partial assignments in this situation appears to be more involved, however.

Simple chronological backtracking to traverse the propositional (and thus also the modal) search tree is probably not the optimal choice. Efficiency can be expected to increase by learning, that is, by remembering previously failed sub-propositions (nogood recording, intelligent backtracking), and also successfully solved sub-problems (lemma caching).

Finally, we mention many-valued modal logics [Fitting, 1992]. These logics allow for propositional variables to have further values than the Boolean 0 and 1. Our approach to modal logics via constraint satisfaction appears to be particularly suitable to be naturally extended to deal with finitely-valued modal logics.