Optimizing MiniSAT Variable Orderings for the Relational Model Finder Kodkod

Diploma Thesis
of

Markus Iser

Automated Software Analysis Group
Institute for Theoretical Computer Science
Karlsruhe Institute of Technology

Advisor:	JProf. Dr. Mana Taghdiri
Second Advisor:	Dr. Carsten Sinz

Period of Research: 2012-01-01 – 1012-06-30

German Abstract. Kodkod ist ein Constraint Solver für eine relationale Prädikatenlogik erster Ordnung. Über einem endlichen Universum von Atomen übersetzt Kodkod relationale Constraint Satisfaction Problems (CSP) in erfüllbarkeitsäquivalente Formeln in Aussagenlogik. Unter Zuhilfenahme einer der mitgelieferten SAT Solver können diese Probleme in Folge auf Erfüllbarkeit getestet werden. Die Motivation für diese Arbeit ergab sich aus der Entdeckung, dass zulässige Permutationen von Aussagen in Kodkod zu ernormen Laufzeitveränderungen beim SAT Solver führen können. Das liegt daran, dass diese Permutationen zu unterschiedlichen Eingangsordnungen der aussagenlogischen Variablen führen, da sich in der Übersetzung die Reihenfolge der erzeugten Klauseln und Variablen ändert.

Wir haben das Interface zwischen Kodkod und MiniSAT um die Möglichkeit erweitert, Prioritäten für Variablen zu übergeben. Wir nutzen diese Prioritäten in zweierlei Weise. Zum einen können wir dafür sorgen, dass MiniSAT diese verwendet um den eigenen dynamischen Score, nach dem MiniSAT Variablen normalerweise ordnet, teilweise statisch zu überschreiben. Zum anderen können wir mit diesen Prioritäten den MiniSAT-internen Score initialisieren, um kontrollierte Anfangsbedingungen zu schaffen. Wir stellen eine Methode vor die solche Prioritäten anhand von struktureller Analyse des Kodkod-internen Abstract Syntax Trees (AST) extrahiert.

In einem ersten naiven Ansatz beschränken wir MiniSATs binäre Suche auf Kodkods Primärvariablen und erklären Zusammenhänge mit ähnlichen Ansätzen, die in der Literatur zu finden sind. Mit diesem ersten Ansatz können wir die Laufzeit von MiniSAT auf erfüllbaren Instanzen beträchtlich steigern. Dafür verschlechtert sich aber die Laufzeit auf unerfüllbaren Instanzen. Dieses Phänomen bestätigt bekannte beweistheoretische Ergebnisse.

In einem zweiten Ansatz können wir mithilfe einer weicheren Priorisierung von Kodkods Primärvariablen die guten Laufzeiten auf erfüllbaren Instanzen erhalten und gleichzeitig die Performance-Einbußen auf unerfüllbaren Problemen stark dämpfen.

Der dritten Ansatz dreht sich um die strukturelle Analyse von Kodkods Abstract Syntax Tree (AST). Wir berechnen fein verteilte Prioritäten, um Kodkods Primärvariablen untereinander zu ordnen. Dabei wollen wir erreichen, dass der SAT solver die Suche mit den Variablen beginnt, für die die meisten Bedingnungen gelten, vor dem Hintergrund, dass dadurch Sackgassen im Suchbaum früher identifiziert werden können und Folgebedingungen eher propagiert werden. Wir konnten dadurch leichte Verbesserungen erzielen gegenüber der gleichmäßigen Priorisierung von Primarvariablen, insbesondere aber im Bereich der unerfüllbaren Probleme. Für die strukturelle Analyse des AST haben wir Methoden entwickelt, die auch für zukünftige Ansätze interessant sind.

Abstract. Kodkod is a constraint solver for bounded relational first-order logic which translates problems into equisatisfiable propositional formulas, in turn using an off-the-shelf SAT solver to check their satisfiability. This work is motivated by the observation that permuting top-level propositions in Kodkod can lead to great runtime variations in the SAT solver. The reason for this phenomenon are different initial variable orderings, as Kodkod processes and translates the constraints in the given order.

We extended the interface between Kodkod and MiniSAT to accept priorities for variables and used them in two distinct ways. One approach involves a partial override of MiniSAT's native variable ordering score, whereas the other is based on score initialization, leaving MiniSAT a great liberty during the curse of evaluation. We device a method of extracting variable priorities by structural analysis of Kodkod's Abstract Syntax Tree (AST).

In a simple initial approach, we restricted the search to Kodkod's primary variables and explain the correspondence to similar experiments regarding input restricted branching found in literature. While that first naive approach leads to considerable performance boosts on satisfiable problems, a general performance deterioration is encountered on unsatisfiable instances. This phenomenon empirically confirms important proof-theoretic results.

In a second approach, we uniformly initialized the MiniSAT score related to Kodkod's primary variables, hereby strongly dampening the performance deterioration on unsatisfiable problems while still yielding the performance improvement on satisfiable problems expectable due to the initial experiment.

By structural analysis of Kodkod's AST we calculated a more fine-grained set of priorities to modify the inner ordering of Kodkod's primary variables. The objective is to prioritize the most constraint variables for early discovery of dead ends in the search tree and to maximize boolean constraint propagation. This third and last approach leads to runtime improvements on both satisfiable *and* unsatisfiable instances. In order to be able to analyze Kodkod's abstract syntax tree, we developed methods which may yet remain interesting for future approaches.

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Karlsruhe, 2012-06-30

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Kodkod [26] is a model finder for bounded constraint satisfaction problems in relational logic, that is a first-order logic combined with relational operators. The language is augmented with transitive closure, integer arithmetic, and cardinality operators. Kodkod translates the constraints formulated in its bounded relational first-order logic into an equisatisfiable formula in propositional logic. It interfaces several SAT solvers and uses one of these to solve the constraint satisfaction problem. It has been developed as a core API for the Alloy Analyzer [15] [28], a tool used in wide range of applications, due to its ability of describing general structures and their exploration. The Alloy language has been used to check security properties of network protocols and for exploration of their topology. It has also been used to formally verify access control and authentication protocols. More examples are the modeling of file-systems, software, railway control, etc. Until today the new core API of Alloy Analyzer 4 (Kodkod) is used in many other programs in the realm of declarative coding and automated code checking (e.g. TACO [10], JForge [8], Nitpick [4]).

Kodkod introduced a unique technique to handle types and partial instances efficiently. The declaration of a relation includes the specification of a lower and an upper bound, where the lower bound contains all the tuples the relation *must* contain (partial instantiation), and the upper bound contains all the tuples the relation *may* contain (thereby partitioning the universe into types).

The universe of a Kodkod problem is bounded, that means it contains only a fixed number of atoms. Quantifiers can thus be resolved and the problem translated to propositional logic. The benefit of improved SAT solver performance is not a mere reduction in runtime, but also an increased bound on the universe, leading to higher confidence in the generated proofs.

Amongst other SAT solvers Kodkod supports MiniSAT [9], an award-winning SAT solver that can easily be extended due to its open architecture. A SAT solver's performance is very sensitive to the applied branching heuristic that decides which variable to assign next. In most modern DPLL-based SAT solvers a dynamic variable ordering heuristic is implemented to optimize their performance. Variants of a method called Variable State Independent Decaying Sum (VSIDS) which was introduced with the SAT solver Chaff [24] are now very common to DPLL-based satisfiability testing. With VSIDS SAT solvers maintain a counter for each variable that is dynamically adjusted during runtime. At each state this counter induces an ordering on the set of boolean variables. Also MiniSAT uses an implementation of VSIDS to determine a variable ordering.

1.1 Objective and Motivation

Experiments with Alloy models have shown that the permutation of propositions in a top-level conjunction may lead to significant variations in MiniSAT's runtime. The reason is that Kodkod processes these propositions in the given order and that affects the order in which it produces constraints and boolean variables. Although the resulting propositional formulas in Conjunctive Normal Form (CNF) are isomorphic, the input order of the clauses determines the entry-point where a SAT solver start its search, given that the chosen solver does not perform an isomorphism-invariant order initialization.

Our goal is to devise a method which allows to determine a good entry-point for a SAT solver by extracting information from Kodkod's internal data-structure, the Abstract Syntax Tree (AST). Structural properties of a Kodkod problem that can efficiently be extracted in its first-order relational representation get lost in its propositional encoding. Therefore the objective is to find effective ways to influence the search order of SAT solvers by exploitation of additional information that is available at the high-level problem description underlying.

1.2 Methodology and Results

For each non-constant relation Kodkod generates boolean variables, each of which determines whether a certain tuple belongs to that relation or not. Those are called the primary variables, in distinction to the auxiliary variables emerging during the translation. A Kodkod model is satisfiable if and only if there exists a satisfying assignment for its primary variables. Since primary variables usually make up for a few percent of the total amount of variables, restricting the solver to primary variables leads to a

tremendous reduction of the search space. This method is usually referred to as Input Restricted Branching and numerous case studies can be found in literature about its application to several problem domains. We will see that MiniSAT benefits a lot from Input Restricted Branching on satisfiable Kodkod problems. However, the performance on unsatisfiable problems deteriorates on application of Input Restricted Branching. We will show how this result corresponds to theory.

We also present a method that tampers with the initial variable ordering of MiniSAT. Since MiniSAT orders variables by a dynamic score — its activity — that is initialized with zero by default, we extended the interface between Kodkod and MiniSAT to be able to initialize the activity array. On uniform initialization of primary variables' activity we encounter similar performance boosts on satisfiable Kodkod problems like in Input Restricted Branching. Furthermore the performance deterioration on unsatisfiable problems vanishes.

Kodkod's relations induce a partition on the primary variables. We present a method that analyses the Abstract Syntax Tree of Kodkod and distribute scores to relations that are constraint by subformulas that we classify as being highly constraining. For each relation the accumulated scores form a weight that induces a partial ordering among the relations. We will show how these weights can be used to adjust the initial inner order of the primary variables to match that partial ordering. That finegrained initialization of variable activity gives still a little performance boost compared to the uniform initialization method; especially on unsatisfiable problems.

A spin-off of the scoring algorithm is a function that quantifies the sensitivity of a boolean formula on assignments to a specific input variable. For our special case we can give an efficient fuzzy algorithm to evaluate that function on the formulas emerging in the translation of Kodkod expressions.

1.3 Example

Figure 1.1 gives an example problem for Kodkod as it could be formally described. Since Kodkod provides an API for other programs to formulate such problems they are usually represented by Kodkod's internal data-structures. In the example we declare a state transition relation ($trans$) and a self transition relation ($self$) over a universe of two symbolic states $\{a_0, a_1\}$. Furthermore we declare a legal transition relation ($legal$) and constrain it to be a subset of $trans$, and to contain the same self transitions as the ones in $self$. The upper bounds of $trans$ and $legal$ are the cross product of all possible states to allow for arbitrary values, while the upper bound of $self$ constrains

$\mathsf{example} = \langle \mathsf{universe},\ \mathsf{rels},\ \mathsf{vars},\ \mathsf{decl},\ \mathsf{formula} \rangle$

$\mathsf{universe} = \{a_0, a_1\}$
$\mathsf{rels} = \{trans, self, legal\}$
$\mathsf{vars} = \emptyset$
$\mathsf{decl}(trans) = \langle \{\}, \{\langle a_0, a_0\rangle, \langle a_0, a_1\rangle, \langle a_1, a_0\rangle, \langle a_1, a_1\rangle\}\rangle$
$\mathsf{decl}(self) = \langle \{\}, \{\langle a_0, a_0\rangle, \langle a_1, a_1\rangle\}\rangle$
$\mathsf{decl}(legal) = \langle \{\}, \{\langle a_0, a_0\rangle, \langle a_0, a_1\rangle, \langle a_1, a_0\rangle, \langle a_1, a_1\rangle\}\rangle$
$\mathsf{formula} = (legal \subseteq trans) \wedge (legal \cap iden = self)$

Figure 1.1: Kodkod Example Problem

it to only contain self transitions. The syntax and semantics of Kodkod problems is further explained in chapter 2. There we will also show how Kodkod would translate this problem to propositional logic.

1.4 Outline

Chapter 2 provides background information about the relational model finder Kodkod, its models and their translation to propositional logic. The SAT solver MiniSAT is explained with respect to the general framework of CDCL solvers and the utilized variable ordering heuristic. A comprehensive section about related work can also be found in chapter 2 (see section 2.3). Possibilities and methods of influencing the variable ordering of MiniSAT by Kodkod are discussed in chapter 3. An algorithm is presented (algorithm 3.2) that calculates a partial order among Kodkod's primary variables which are further used to influence MiniSAT's variable ordering. We also devise a method that quantifies the sensitivity of boolean formulas to singular variable assignments in section 3.2.3. Several experiments have been conducted with variants of the described methods. The results are presented in chapter 4. The thesis concludes with a summary and elaboration of future prospects in chapter 5.

Chapter 2

Background

Kodkod [26] is a constraint solver for bounded relational first order logic. Kodkod translates Constraint Satisfaction Problems (CSP) into equisatisfiable propositional formulas, in turn using a SAT solver to determine satisfiability of those problems.

2.1 Kodkod

Kodkod's language is a first-order logic combined with relational operators like union or relational composition. It is also capable of encoding integer arithmetic and provides cardinality operators. Another prominent candidate here is the unary operator 'reflexive closure'. Kodkod provides a rich language for the formulation of relational constraints. The problems are bounded so that sets and relations are drawn from a finite universe, quantifiers can be resolved and the problems translated to propositional logic. Kodkod interfaces with several SAT solvers including SAT4J, several versions of MiniSAT and Lingeling, to name a few. It has been developed as a superset of the Alloy language and as of Alloy Analyzer 4 it serves as its core component [27]. Today many programs utilize Kodkod's powerful features by its well documented API, for example TACO [10], JForge [8] or Nitpick [4].

2.1.1 Syntax

A problem in Kodkod is represented by a tuple $\langle \mathsf{U}, \mathsf{rels}, \mathsf{vars}, \mathsf{decl}, \mathsf{F} \rangle$. The universe $\mathsf{U} = \{a_1, a_2, \ldots, a_n\}$ is a finite set of atom symbols and F is a relational first order formula expressing the constraints. The set $\mathsf{rels} = \{R_1, R_2, \ldots, R_n\}$ is a set of relation symbols, and the function $\mathsf{decl} : \mathsf{rels} \to 2^{2^U} \times 2^{2^U}$ assigns to each relation $R \in \mathsf{rels}$ a tuple $\langle R_l, R_u \rangle$, where R_l denotes the relations lower bound and R_u denotes its upper bound.

$problem ::= \langle U,\ rels,\ vars,\ decl,\ F \rangle$

$U ::= \{\text{`}\texttt{a}_1\text{'}, \ldots, \text{`}\texttt{a}_n\text{'}\}$
$rels ::= \{\text{`}\texttt{R}_1\text{'}, \ldots, \text{`}\texttt{R}_j\text{'}\}$
$vars ::= \{\text{`}\texttt{v}_1\text{'}, \ldots, \text{`}\texttt{v}_k\text{'}\}$
$decl ::= \{R \to \langle R_l, R_u \rangle \mid R \in rels,\ R_l \subseteq R_u \subseteq U^d,\ d > 0\}$
$F ::= formula$

$varDecl ::= v : exp,\ v \in vars$

$exp ::= rel \mid const \mid v, v \in vars$
$\mid unaryExp \mid binaryExp$
$\mid$ '{' $varDecl^+$ '|' $formula$ '}'
$\mid$ '`Int`' $intExp$
$\mid$ '`if`' $formula$ '`then`' exp '`else`' exp

$const ::=$ '`none`' | '`iden`' | '`univ`' | '`ints`'

$unaryExp ::=$ '~'exp | '*'exp | '^'exp

$binaryExp ::= exp$ '$\cup$' $exp \mid exp$ '$\cap$' $exp \mid exp$ '$\setminus$' exp
$\mid exp$ '$\to$' $exp \mid exp$ '.' $exp \mid exp$ '$+\!\!+$' exp

$intExp ::=$ number | '#'exp | '`int`' $var \mid binaryIntExp$
$\mid$ '`if`' $formula$ '`then`' $intExp$ '`else`' $intExp$

$binaryIntExp ::= intExp$ '$+$' $intExp \mid intExp$ '$-$' $intExp$
$\mid intExp$ '$*$' $intExp \mid intExp$ '$/$' $intExp$

$formula ::=$ '`not`' $formula$
$\mid exp$ '`in`' $exp \mid exp$ '$=$' exp
$\mid intExp$ '$<$' $intExp \mid intExp$ '$=$' $intExp \mid intExp$ '$\leq$' $intExp$
$\mid formula$ '$\wedge$' $formula \mid formula$ '$\vee$' $formula$
$\mid formula$ '$\Leftrightarrow$' $formula \mid formula$ '$\Rightarrow$' $formula$
$\mid$ '`no`' $exp \mid$ '`one`' $exp \mid$ '`lone`' $exp \mid$ '`some`' exp
$\mid$ '`all`' $varDecl^+$ '|' $formula$
$\mid$ '`some`' $varDecl^+$ '|' $formula$

The abstract syntax of a Kodkod problem, defined by a tuple containing the finite set of atoms U (the universe), a formula F, a finite set of relation symbols *rels*, a finite set of variable symbols *vars* and a function *decl* of relation declarations, mapping each relation symbol R to its lower and upper bound $\langle R_l, R_u \rangle$ of an implicitly defined dimensionality d.

Figure 2.1: Kodkod Abstract Syntax

For each $R \in \mathsf{rels}$ its declaration $\mathrm{decl}(R)$ has to satisfy the restriction $R_l \subseteq R_u \subseteq U^d$ where d is referred to as the arity of relation R. The lower bound contains all the tuples a relation *must* contain, whereas the upper bound contains all the tuples a relation *may* contain. Hence for each relation R of arity d and its lower and upper bounds R_l and R_u the following partial order holds: $R_l \subseteq R \subseteq R_u \subseteq U^d$. In Kodkod the upper bound is used to introduce types and the lower bound is used to specify partial instances.

A relation R is constant if and only if R_l equals R_u. The built-in constant relations '`none`', '`iden`', '`univ`', and '`ints`' denote the empty set, the identity relation, the universal set (of all atoms) and the set of all integer atoms (for a user-defined bit-width), respectively.

Kodkod provides relational operators to construct complex relational expressions like union '$\mathtt{R} \cup \mathtt{S}$', intersection '$\mathtt{R} \cap \mathtt{S}$', difference '$\mathtt{R} \setminus \mathtt{S}$' and cartesian product '$\mathtt{R} \to \mathtt{S}$'. Furthermore '`R.S`' denotes relational join (composition), and '$\mathtt{R} +\!\!+ \mathtt{S}$' denotes relational override. The unary operators '`~R`', '`^R`', '`*R`' denote transpose, transitive closure and reflexive transitive closure, respectively, and are applicable to binary relational expressions only. Set comprehensions and ternary if-then-else expressions are also supported. According to the semantics of the expression operators the expressions have to satisfy the operator specific arity restrictions, mixed arity expressions for example are not allowed (see [28]).

Integer expressions evaluate to integer values and are constructed from numbers, arithmetic operators, set cardinality '#', and if-then-else expressions. The cast operators '`int`' and '`Int`' give the integer value corresponding to an integer atom, and vice versa.

Basic Kodkod formulas are constructed using the subset, equality, and integer comparison operators, and are combined using the usual logical operators. The formulas '`no E`', '`one E`', and '`lone E`' constrain the cardinality of a relational expression '`E`' to be zero, one, and at most one, respectively. The quantifiers '`all`' and '`some`' denote the universal and existential quantifiers.

Figure 2.1 gives an abstract syntax for Kodkod's input logic. Internally a Kodkod formula is represented as a DAG (directed acyclic graph) and that data-structure is referred to as the Abstract Syntax Tree (AST). A formula is structured like a tree, but as it comes to expressions and relations (each of which might be constraint by multiple formulas at the leafs of the formula tree) it becomes a DAG. Since the language of Kodkod is a superset of the Alloy language, details on the semantics of all operators can also be looked-up in [15]. As Kodkod translates its formulas into equisatisfiable propositional formulas the semantics now follow from the semantics of the boolean formula constructed during translation.

2.1.2 Translation

Kodkod problems are translated to equisatisfiable propositional formulas via a chain of matrix manipulations [26]. The constraints are first encoded in the circuit data-structure CBC (Compact Boolean Circuit) — similar to binary decision diagrams (BDD) —that has been specifically developed for Kodkod, and are then translated from there to CNF (Conjunctive Normal Form).

Translation of Relations

Given a finite universe of atom symbols $U = \{a_1, \ldots, a_n\}$, Kodkod encodes a k-ary relation $R \subseteq U^k$ as a k-dimensional boolean matrix M over U, whose entries are defined as follows:

$$m_{\{i_k\}} = \begin{cases} 0, & \text{if } \langle a_{i_1}, \ldots, a_{i_k} \rangle \notin R_u \\ 1, & \text{if } \langle a_{i_1}, \ldots, a_{i_k} \rangle \in R_l \\ r_{\{i_k\}}, & \text{otherwise} \end{cases}$$

where $\{i_k\}$ denotes the sequence $i_1, \ldots, i_k$ and $r_{\{i_k\}}$ is a unique boolean variable (for $i_1, \ldots, i_k \in [1, n]$). A tuple belongs to a relation if and only if its corresponding matrix element is set to one (either during the translation or in a satisfying instance found by the SAT solver).

Figure 2.2 shows how the relations declared in the introductory example (figure 1.1) are translated to boolean matrices according to their bounds. All relations are instantiated according to their upper and lower bounds. The upper bounds of the transition relations *trans* and *legal* contain all possible tuples $\langle a_i, a_j \rangle \in \{a_0, a_1\} \times \{a_0, a_1\}$ and their lower bounds are empty. So for each tuple $\langle a_i, a_j \rangle$ in the upper bound of *trans* a boolean variable t_{ij} is created, as well as a boolean variable l_{ij} for each tuple $\langle a_i, a_j \rangle$ in the upper bound of *legal*. Only two variables s_{00} and s_{11} are created for the self transition relation $self$, as its upper bound only contains the diagonal elements $\langle a_0, a_0 \rangle$ and $\langle a_1, a_1 \rangle$. The non-diagonal elements of $self$ are constantly *false* or zero, as they are excluded by the relation bounds. There is no need to individually declare the constant

$$trans = \begin{bmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{bmatrix} \quad self = \begin{bmatrix} s_{00} & 0 \\ 0 & s_{11} \end{bmatrix}$$

$$legal = \begin{bmatrix} l_{00} & l_{01} \\ l_{10} & l_{11} \end{bmatrix} \quad iden = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure 2.2: Example: Translation of Relations

$$legal \cap iden = \begin{bmatrix} l_{00} \wedge 1 & l_{01} \wedge 0 \\ l_{10} \wedge 0 & l_{11} \wedge 1 \end{bmatrix} = \begin{bmatrix} l_{00} & 0 \\ 0 & l_{11} \end{bmatrix}$$

Figure 2.3: Example: Translation of Expressions

relation *iden* since this is a built-in constant binary relation that always contains exactly the diagonal elements. If it would have to be declared explicitly then both the upper and the lower bound would have to contain the tuples $\langle a_0, a_0 \rangle$ and $\langle a_1, a_1 \rangle$ (to demonstrate the purpose of lower bounds).

Translation of Expressions

Relational expressions are translated by subsequent composition of the boolean matrices emerging in the translation of relations. Intersection and union of two relations, for example, are encoded as element-wise conjunction and disjunction of their matrices, respectively, and relational composition is encoded as matrix multiplication. Propositional formulas thus determine the membership of tuples in an expression, like propositional variables determine the membership of tuples in a relation. The primary variables generated for relations serve as input variables to the formulas emerging in the expression translation. Figure 2.3 shows the translation of the intersection from our introductory example (figure 1.1). It also demonstrates how Kodkod eagerly evaluates constant formulas whenever possible.

Figure 2.4 shows the matrix representation `M` of Kodkod's binary expressions, which is recursively defined over the matrix entries $f_{\{i_d\}}$ and $g_{\{i_d\}}$ of the enclosed expressions '`F`' and '`G`', respectively.

$$\begin{aligned}
\texttt{M}(\text{`}\texttt{F} \cup \texttt{G}\text{'}) &= \left(m_{\{i_d\}} = f_{\{i_d\}} \vee g_{\{i_d\}}\right) \\
\texttt{M}(\text{`}\texttt{F} \cap \texttt{G}\text{'}) &= \left(m_{\{i_d\}} = f_{\{i_d\}} \wedge g_{\{i_d\}}\right) \\
\texttt{M}(\text{`}\texttt{F} \setminus \texttt{G}\text{'}) &= \left(m_{\{i_d\}} = f_{\{i_d\}} \wedge \neg g_{\{i_d\}}\right) \\
\texttt{M}(\text{`}\texttt{F} \rightarrow \texttt{G}\text{'}) &= \left(m_{\{i_m\},\{j_n\}} = f_{\{i_m\}} \wedge g_{\{i_n\}}\right) \\
\texttt{M}(\text{`}\texttt{F.G}\text{'}) &= \left(m_{\{i_m\},\{j_n\}} = \bigvee_k \left(f_{\{i_m\}.k} \wedge g_{k,\{j_n\}}\right)\right) \\
\texttt{M}(\text{`}\texttt{F} + \texttt{G}\text{'}) &= \left(m_{i_0,\{i_n\}} = g_{i_0,\{i_n\}} \vee f_{i_0,\{i_n\}} \bigwedge_{\{j_n\}} \neg g_{i_0,\{j_n\}}\right)
\end{aligned}$$

Figure 2.4: Translation of Binary Expressions

$$
\begin{array}{ll}
(legal \subseteq trans): & (l_{00} \implies t_{00}) \wedge (l_{01} \implies t_{01}) \wedge (l_{10} \implies t_{10}) \wedge (l_{11} \implies t_{11}) \\
(legal \cap iden = self): & (l_{00} \iff s_{00}) \wedge (l_{11} \iff s_{11}) \\
\text{final formula:} & (l_{00} \implies t_{00}) \wedge (l_{01} \implies t_{01}) \wedge (l_{10} \implies t_{10}) \wedge \\
 & (l_{11} \implies t_{11}) \wedge (l_{00} \iff s_{00}) \wedge (l_{11} \iff s_{11})
\end{array}
$$

Figure 2.5: Example: Translation of Formulas

Translation of Formulas

Relational formulas are translated as boolean constraints over the matrix entries of Kodkod's expressions and relations. Subset-formulas for example are translated as an element-wise implication over the matrix entries of the enclosed expressions. Figure 2.5 shows the stepwise translation of the formula in our introductory example (figure 1.1). The subset-constraint is translated as an element-wise implication. The equivalence-constraint is translated as an element-wise bi-implication.

2.1.3 Conclusion

The basic concept in Kodkod is that of a relation. A model in Kodkod specifies for each relation which tuple it contains and which not. For each relation R and each tuple $t \in R_u \setminus R_l$ a boolean variable is generated. These variables — each of which corresponds to a membership relation of a tuple and a relation — are called *primary variables*. A Kodkod problem is satisfiable if and only if there exists a satisfying assignment for its primary variables. During the translation from CBC to CNF other variables are generated. The value of each of those *auxiliary variables* can be expressed as a boolean function over a subset of the primary variables. The concept of primary variables thus corresponds to that of an Independent Variable Set (further explained in section 2.3).

2.2 MiniSAT

Kodkod comes with interfaces to several SAT solvers including MiniSAT, an award-winning SAT solver widely used in applications. It is also used in research to implement and test improvements to the state-of-the-art procedure. MiniSAT like most successful complete SAT solvers is based on the DPLL-algorithm [7] (the abbreviation stands for the names of the involved authors Davis, Putnam, Logemann and Loveland). MiniSAT is a CDCL (Conflict Driven Clause Learning) solver. *Clause Learning* is one of the most important advances in modern backtracking SAT solvers.

2.2.1 DPLL Procedure

The DPLL algorithm tries to incrementally build a satisfying truth assignment for a set of clauses. If a clause is satisfied by the current partial assignment, it is removed from the clause-set. A literal that is set to false by the current partial assignment is removed from its clause. If an empty clause is encountered due to all its literals being set to false, that indicates a conflicting assignment. A satisfying solution is found when all clauses are satisfied.

At each stage of the procedure, the value of a variable can either be inferred by the *unit-clause rule*, or else a variable is selected for *branching*. The *unit-clause rule* is still one of the key techniques in modern SAT solvers. If a clause contains only one literal (unit-clause), this literal has to be assigned such that it satisfies the clause. This is also referred to as boolean constraint propagation (BCP) or *unit-propagation*. *Branching* on a variable means that the variable is assigned one of the possible values in $\{0, 1\}$. On encounter of an empty clause, which means that all its literals are set to false by the current partial assignment, the formula can not be satisfied by the current partial assignment. In that case the solver tries to undo a former branching; this step is called *backtracking*. If such a branching exists, the conflict and the trailing assignment now imply the opposite branch. If both branches of all trailing decision variables lead to a conflicting assignment, the formula is unsatisfiable. Otherwise the procedure continues until it either finds a satisfying solution or another conflict occurs that can not be resolved by backtracking.

In addition to the unit-clause rule the original DPLL procedure contained another inference rule called the *pure literal rule*. A literal p is pure if $\neg p$ does not occur in any non-satisfied clause of the given clause-set (with respect to the current partial assignment). Since the DPLL procedure eagerly seeks to satisfy the given formula, the pure literal rule states to assign p such that all clauses containing p are satisfied. Late implementations of the DPLL procedure dropped the pure literal rule, due to the expenses of pure literal detection at each recursion.

2.2.2 Clause Learning

Clause learning denotes a set of techniques for adding clauses learnt during the search by conflict analysis to the clause database. This section gives a quick overview on the underlying techniques. A brief description can also be found in the chapter about CDCL solvers [23] in the Handbook of Satisfiability [3].

Resolution

Given two clauses c_1 and c_2 and a literal l such that $l \in c_1$ and $\neg l \in c_2$ the resolvent $c_1 \odot c_2$ is the new clause $(c_1 \setminus \{l\}) \cup (c_2 \setminus \{\neg l\})$. Resolution is a sound and complete calculus for refutation proofs by deduction of the empty clause, given an unsatisfiable clause set. However, clause learning is a sequence of selective resolution operations to generate a new clause that helps pruning the search space. Clause learning does not affect soundness or completeness. A formula f is satisfiable if and only if $f \cup w$ is satisfiable for all its resolvents w.

Conflict Analysis

Starting with zero, whenever branching occurs a global counter indicating the current decision level is incremented by one. For each assignment (also inferred assignments) the *decision level* at which the assignment was made is stored. If a conflict occurs the reasons for assignment are determined for each literal of the conflict clause. This determination of reasons is recursively repeated for all implied literals at the *current decision level* until the decision-variable of the current decision level is reached. The result is a DAG-like (Directed Acyclic Graph) structure that is referred to as the *implication graph*. In the implication graph each vertex symbolizes a variable. For each variable of the current decision level that is implied by unit-propagation, incoming edges are drawn rooting at the false literals of the clause that has become unit (due to assignments in a former decision level). A conflict clause can now be obtained by a cut in the implication graph, where the antecedents are on the reason side and the consequences are on the conflict side. So the conflict clause contains all literals on the reason side that have an edge to an implied literal on the conflict side. This can be formally described with subsequent resolution of the conflict clause with the clauses that have become unit at the current decision level.

Different learning schemes have been proposed and they mainly differ in which cut they choose and how many clauses are learnt at each conflict. One important notion that remains is that of a Unit Implication Point (UIP). UIP's are dominators in the implication graph, that is each path from the decision variable to the conflict variable includes the UIP. The most effective SAT solvers make a cut at the first UIP instead of proceeding with resolution until the current decision variable is reached. This also results in shorter learnt clauses.

2.2.3 Non-Chronological Backtracking

CDCL solvers also incorporate *non-chronological backtracking* as a result of the deeper conflict analysis in the learning step. As a clause is learnt it is always unit in the conflicting literal and therefore assertive. The learnt clause remains unit until backtracking occurs to the highest decision level of its other literals. Modern SAT solvers like MiniSAT always take that backtracking step immediately.

2.2.4 Random Restarts

Random restarts help a state of the art SAT solver to leave critical regions of the search space when a bad decision is made early in the decision tree. A restart means that all decisions are taken back and the solver starts anew, thereby getting a chance to select other decision variables first and explore a different region of the search space. Despite of frequent restarts completeness is still guaranteed since information about the search's progress is kept by learnt clauses. Frequent random restarts boost SAT solver performance a lot and also MiniSAT uses a restart strategy with increasing intervals based on the Luby-sequence [21] which is a proven statistical optimum.

2.2.5 Branching Heuristic

Making a good choice on which variable to branch on is important. Given the same algorithm framework, different choices may produce search trees of dramatically different sizes.

MOMS (Maximum Occurrences in Minimum Sized clauses)

Early branching heuristics kept track of statistical information about variable occurrences. They preferred the literals occurring in the maximum number of minimum sized clauses (see e.g. [18]). Intuitively the literals belonging to the shortest clauses are the most constraint literals in the formula. Branching on them first might help in early discovery of dead ends in the search tree. Assigning variables occurring in short clauses also produces new unit-clauses and more assignments can be inferred by unit-propagation. Variants of the MOMS (Maximum Occurrences in Minimum Sized clauses) heuristic were very successful in the 1990ies.

VSIDS (Variable State Independent Decaying Sum)

The original VSIDS (Variable State Independent Decaying Sum) was proposed in the year 2001 by the authors of the Chaff solver [24]. They maintain a counter for each literal which is initialized by zero. Each time a clause *or* a learnt clause is added to the database the counters of its literals are incremented by one. At each decision the literal with the highest counter is chosen to be satisfied. Periodically, all counters are divided by a constant.

VSIDS is a dynamic conflict driven heuristic that directs the search to variables that were recently involved in conflicts. One of its key advances is its very low overhead compared to MOMS-like heuristics that need to evaluate statistical information about the formula that always changes during search. The reason why VSIDS was introduced is in fact the presence of lazy data-structures in modern SAT solvers, where such statistical information can not easily be extracted. Variants of VSIDS are implemented in most successful modern SAT solvers.

MiniSAT's implementation of VSIDS [9] is a little different from the original. The main difference is that only variables have counters and not literals. Therefore the polarity of the next assignment is decoupled from the ordering heuristic. A variables' counter — its activity — is incremented by a number (starting with 1) each time a variable occurs in a conflict. At each conflict this increment number decays by a variable decay factor (set to 0.95 by default). If one counter crosses the constant threshold 10^{100} all counters are divided by 10^{100}. Since MiniSAT's variable ordering is solely based on its activity — which is always zero right after initialization — and not on its initial occurence-count in clauses, the initial ordering is highly sensitive to the order in which clauses are read.

2.2.6 CDCL Algorithm

Algorithm 2.1 gives an outline of the general layout a modern CDCL algorithms based on MiniSAT. At each step of the iteration unit-propagation is executed until all unit-clauses are satisfied (line 1). If no conflict occurs and there are still unassigned variables, one variable is selected (line 6), utilizing the implemented variable ordering heuristic. The variable's assignment is added to the current partial assignment and the iteration continues with unit-propagation respecting the new partial assignment. On encounter of a conflict, conflict-analysis is executed (line 10) and if backtracking is still possible the generated conflict clause is added to the database (line 13) and backtracking occurs (line 20). Depending on the restart heuristic the solver might also backtrack all

assignments at that point (line 18).

Algorithm 2.1: CDCL (based on MiniSAT)

```
   // CDCL solver with activity-based VSIDS similar to MiniSAT
   Data: clauses: set of clauses
   Result: true if clauses are satisfiable, false otherwise
 1 while true do
 2     unit-propagation
 3     if not conflict then
 4         if all variables assigned then
 5             return true
 6         else
 7             decision-variable ← select variable with highest activity
 8             value ← select polarity
 9             assign(decision-variable, value)
10     else
11         analyze-conflict
12         if top-level conflict found then
13             return false
14         add conflict-clause
15         foreach variable ∈ conflict-clause do
16             increment-activity(variable)
17         uniform-activity-decay
18         if restart heuristic activates then
19             backtrack all decisions
20         else
21             backtrack while conflict-clause is unit
```

2.3 Related Work

Given an input formula on a set of variables N, an 'Independent Variable Set' (IVS) is a (usually small) subset S of N whose assignment is sufficient to determine the truth value of all variables in N. Input restricted branching is a way of exploiting the existence of an IVS in SAT solvers by branching only on variables of S, thereby reducing the search-space from $2^{|N|}$ to $2^{|S|}$.

There are numerous case-studies on input restricted branching and the results are mixed. Giunchiglia et al. [13] report positive results on application of restricted branching to planning problems. They report in another work [12] about their experiments with restricted branching on problems from several domains under utilization of different branching heuristics. They showed that the outcome is not only specific to the type of problem but also depends on the applied branching heuristic and backtracking scheme. Marques Silva and Lynce show in [22] that restricted branching stabilizes and improves solver runtimes on certain encodings of cardinality constraints that produce many auxiliary variables.

A P-Solver is a SAT solver that either rejects a formula or solves it in polynomial time. Williams et al. introduce the notion of backdoor sets [29]. A backdoor is a set of variables for which there *exists* an assignment such that the simplified problem is accepted by a P-Solver. A strong backdoor is a set of variables for which *every* assignment simplifies the problem such that it is accepted by a P-Solver. Small backdoor sets exist in many practical problems and their exploitation leads to a tremendous reduction of the search-space. However the detection of small backdoors is algorithmically hard.

Notably an IVS is a particular strong backdoor, but possibly not of minimum cardinality. However, the proof-theoretic analysis of Järvisalo et al. [17] [16] shows that in general the power of propositional proof-systems is deteriorating on their extension by input restricted branching. They utilize *proof complexity* [6] [2] as a measure to analyze the efficiency of a propositional proof system. The proof complexity $C_T(f)$ of a (unsatisfiable) propositional formula f in a proof system T is the size of the shortest refutation for f in T.

They compare two proof systems by their relative efficiency using *polynomial simulation*. A proof system T' simulates another proof system T if for all infinite problem families $\{P_n\}$ there is a polynomial p such that for all P_n $p(C_T(P_n)) \geq C_{T'}(P_n)$. In [17] they describe a tablaux method for satisfiability testing on boolean circuits that corresponds to DPLL. That way they prove that input restricted DPLL (DPLL^{i}) cannot simulate DPLL, because there exist problem families $\{P_n\}$ such that for all polynomials p it holds that $p(C_{\text{DPLL}}(P_n)) < C_{\text{DPLL}^{\text{i}}}(P_n)$.

A clause learning proof system CL is provably more powerful than DPLL. Beame et al. have shown earlier that DPLL cannot simulate CL [1]. In [16] Järvisalo et al. prove that also input restricted CL (CL^{i}) cannot simulate CL. CL^{i} cannot even simulate basic DPLL.

A backbone is a set of variables which have the *same* value in every satisfying assignment of a (satisfiable) propositional formula [20]. The size of backbone sets has been

shown to correlate with problem hardness. Parkes [25] has shown for satisfiable random 3-SAT instances that under-constrained problems have a strong tendency of having very small backbone sets, whereas highly constrained satisfiable problems tend to have very big backbones sets.

The notion of careset was introduced by Ganai [11]. A careset is a subset of variables in a boolean formula that *must* be assigned in any minimally satisfying assignment of a formula F. A minimally satisfying assignment for F is an assignment where all remaining unit propagations are executed and under which F is accepted by a P-Solver. Furthermore — for minimality — it must satisfy the condition that when taking a variable away from the assignment it loses the former properties. In contrast to variables in a backbone set the variables in a careset only have to be assigned and not necessarily to unique values. Compared to backdoors which are *sufficient* sets for P-Solver acceptance, a careset is a *necessary* set for P-Solver acceptance and might therefore be smaller than a backdoor set. The notion of careset is extended to unsatisfiable formulas by its definition on maximum satisfiable subsets. Ganai presents a method that extracts a careset at the application level and thereby was able to boost solver performance via restricted branching on that careset.

Chapter 3

Design and Implementation

As the objective is to determine a reasonable Kodkod specific variable ordering for MiniSAT, in particular to determine a good entry-point to the search space, the approach is twofold. First we determine possibilities to influence Minisat's variable ordering by external priorities. And then we specify criteria to be used to calculate these priorities.

3.1 Influencing MiniSAT's variable ordering

We extended the JNI interfaces of Kodkod and MiniSAT to accept an array of priorities, each of which is associated with a variable. Additionally we introduced a flag to switch between *two* possible ways for MiniSAT to deal with the given priorities both of which are explained below.

3.1.1 Activity Overriding

In this approach, the external priorities were used to statically override MiniSAT's native score (the variables' activity). That means variables were ordered by the external score. Only if the external scores for two variables were equal the variables activity was used to break ties.

3.1.2 Activity Initialization

In case of activity initialization, the external priorities were used to initialize the values of MiniSAT's activity array. Since by default the activity of each variable is initialized with zero, small values suffice to influence the initial order. Naturally the effect of this method on variable ordering blurs when MiniSAT adjusts scores during runtime.

3.2 Extracting Criteria for Prioritization

Our focus is on primary variables since we have direct access to their semantics via the Abstract Syntax Tree (AST). Kodkod's primary variables constitute an Independent Variable Set (IVS) as it is explained in section 2.3. We thus start with mimicking input restricted branching for Kodkod problems.

3.2.1 Prioritizing Input Variables

The set of primary variables constitutes only a small fraction of the search space, they usually make up for only a few percent of the total amount of variables. By restricting the search to primary variables, the size of the search-space is reduced by high orders of magnitude. We mimicked input restricted branching by *uniform* activity overriding (see subsection 3.1.1) on all primary variables to force MiniSAT to assign primary variables first. Since a complete assignment can be inferred from the assignment of primary variables this approach corresponds to input restricted branching.

In a second approach we uniformly initialized MiniSAT's variable activity (see subsection 3.1.2) of all primary variables. In this approach the solver only starts in the realm of primary variables, but then is allowed to rearrange the variables and to also make assignments on intermediate variables. Both approaches are evaluated in chapter 4.

3.2.2 Highly Constraining Formulas

Utilization of the Kodkod specific partitioning of input variables into relations gives a more fine-grained control on the priorities to be enforced. By extraction of information from Kodkod's AST, and local evaluation of distinct formula nodes, we decide on which subset of the primary variables (in terms of a relation) the solver should start with branching.

The proposed algorithm distributes scores to relations constrained by formulas classified as being highly constraining (HCF). A formula is considered being HCF if exponential decay in the number of possible solutions is encountered under application of that formula. The idea is to first assign the primary variables that stem from the most highly constraint relation. Thereby the number of unit propagations should increase and large parts of the search space can be dropped early due to faster detection of dead ends in the decision tree.

Consider two unary relations R and S with $|R| = |S| = n$. Without further restrictions the total number of possible instantiations of both relations is $2^{2n} = 4^n$. Given a

constraint category	formula	solution space without constraint	solution space with constraint
relational comparison	'R in S'	4^n	3^n
	'R = S'	4^n	2^n
cardinality bounding	'#R = c'	2^n	$\binom{n}{c}$
	'#R ≤ c'	2^n	$\sum_{k=0}^{c} \binom{n}{k}$
	'#R < c'	2^n	$\sum_{k=0}^{c-1} \binom{n}{k}$
	'no R'	2^n	1
	'one R'	2^n	n
	'lone R'	2^n	$n+1$

Table 3.1: Highly constraining formulas (HCF) and their effects on the size of the solution space for a universe of size n, unary relations 'R' and 'S', and a constant number 'c'.

subset constraint 'R ⊆ S' the number of possible instantiations deteriorates from 4^n to 3^n (by a factor of $(1.5)^n$). Introduction of an equality constraint 'R = S' leaves a total of 2^n possible instantiations. Cardinality constraints of the type '#R = c' for *small* c are even more restrictive. For $c = 1$ the solution space decreases from 2^n to n. Since n increases with the bounds, c is always considered being small with respect to n. Table 3.1 displays an overview of the formulas that are classified as HCF.

By assigning the highly constraint subset of primary variables *first*, we expect the initial amount of boolean constraint propagations to increase. Intuitively this is best captured by observation of equality formulas. If one side of the equality is fixed, the other side can be inferred by it.

Another justification for that approach can be found in literature. As described in subsection 2.3, Parkes [25] has shown for satisfiable 3-SAT instances that the more constraints are added (in terms of clauses) the larger the backbone size of the overall formula gets. In highly constraint subsets of Kodkod's primary variables the probability for each variables being backbone should therefore increase. Since backbone variables have the same polarity in all satisfying truth assignments, assigning them first can also lead to early discovery of dead ends in the search space.

Dominance of Relations in Expressions

Since Kodkod formulas constrain relational expressions and not necessarily relations — relations are leaf-expressions — we need to devise a measure to be able to quantify the importance of a single relation in an arbitrary expression, as we want to distribute scores to the relations enclosed by highly constraint expressions. Constraining the union '`R` ∪ `S`' for example — consider the case where the upper and lower bound of '`R`' and '`S`' share tuples ($R_u \cap S_u \neq \emptyset$) — might not have as much of a constraining effect as two separate constraints on '`R`' and '`S`'. That means we need to moderate the score we distribute to '`R`' and '`S`'.

In our introductory example 1.1, we detect two highly constraining formulas: ($legal \subseteq trans$) and ($legal \cap iden = self$). In the subset formula ($legal \subseteq trans$) both relations $legal$ and $trans$ are directly constraint; therefore we can distribute whatever score we chose for subset-formulas directly to those relations. We encounter another situation by looking at the equality constraint ($legal \cap iden = self$). Here, only the relation $self$ is directly constraint by equivalence. On the left side though, the intersection $legal \cap iden$ — the element-wise conjunction of $legal$ and $iden$ — is constraint by equivalence (figure 2.3). Since $iden$ is a constant relation, still only elements of $legal$ are constraint, but due to non-diagonal elements of $iden$ constantly being zero, only half of the elements of $legal$ are constraint.

We want to quantify the *dominance* of each relation in a relational expression by looking at the effect their evaluation has on the expression. Note that not only the expression type but also the bounds of the enclosed relations have to play an important role in such a measure. The bounds of each relation determine the width of the boolean formulas emerging in the expression translation and especially whether they are constant or not. Therefore we evaluate that relation dominance we are looking for tuple-wise.

Before giving the complete formula, the needed function is abstractly defined in definition 3.1. The missing function $\delta(v, f)$ measures the dominance of a relation's primary variable in an expression's matrix entry (the propositional formulas emerging in expression translation). We deduce such a measure in subsection 3.2.3.

Definition 3.1 (Average Dominance)**.** The average dominance of a relation R (with primary variables $r_{\{i_n\}}$) in an expression E (with matrix entries $e_{\{j_m\}}$) is the aggregated dominance $\delta\big(r_{\{i_n\}}, e_{\{j_m\}}\big)$ of its primary variables $r_{\{i_n\}}$ in the boolean formulas $e_{\{j_m\}}$.

$$\mathrm{dom}(R, E) = \frac{1}{\big|\{r_{\{i_n\}}\}\big|} \sum_{\{i_n\}} \sum_{\{j_m\}} \delta\big(r_{\{i_n\}}, e_{\{j_m\}}\big)$$

3.2.3 The Sensitivity of Boolean Formulas to Singular Assignments

Let $\mathcal{F}$ be the set of propositional formulas and $\mathcal{V}$ be the set of boolean variables. In the following the set $\{0, 1, \neg, \wedge, \vee\}$ of syntactic symbols is used as an algebraic basis for boolean formulas. Every variable $v \in \mathcal{V}$ is a boolean formula, every constant $c \in \{0, 1\}$ is a boolean formula, for every formula $g \in \mathcal{F}$ also $\neg g$ and for every pair of formulas $g_1, g_2 \in \mathcal{F}$ also $g_1 \wedge g_2$ and $g_1 \vee g_2$ are boolean formulas with the usual semantics.

Solution Rate

Let $\text{vars} : \mathcal{F} \to 2^{\mathcal{V}}$ be a function assigning to every formula $g \in \mathcal{F}$ its set of input variables. Let $|g| := |\operatorname{vars}(g)|$ denote the number of input variables of g. Then $2^{|g|}$ is the number of possible assignments to the variables of g. The possible assignments to input variables of a formula g can be divided into satisfying and non-satisfying assignments. Let $\#g$ denote the number of satisfying assignments to variables of g and let $\not\#g$ denote the number of non-satisfying assignment to variables of g. So it clearly holds that $\#g + \not\#g = 2^{|g|}$.

Definition 3.2 (Solution Rate)**.** Given a propositional formula g, the solution rate is the satisfying fraction of all assignments denoted by

$$\rho^+(g) = \frac{\#g}{2^{|g|}}$$

For convenience the abbreviation $\rho(g) = \rho^+(g)$ will be used.

Example 3.1 (Solution Rate)**.** *Given the formula $g = a \vee (b \wedge \neg c)$ where $a, b, c \in \mathcal{V}$ are variables, the solution rate is given by*

$$\rho(g) = \frac{\#g}{2^{|g|}} = \frac{5}{2^3} = \frac{5}{8}$$

Trivially the solution rate of constant formulas is $\rho(0) = 0$ and $\rho(1) = 1$ and the solution rate of all variables $v \in \mathcal{V}$ is $\rho(v) = \frac{1}{2}$.

Corollary 3.1 (Inverse Solution Rate)**.** The inverse solution rate $\rho^-(g) = 1 - \rho^+(g)$ denotes the non-satisfying fraction $\dfrac{\not\#g}{2^{|g|}}$ of all assignments of g.

The solution rate of a propositional formula g practically denotes the probability of a randomly chosen assignment to be a satisfying assignment for g. Conversely, the

inverse solution rate of g denotes the probability of a randomly chosen assignment to be a non-satisfying assignment for g. *Our* focus lies on the analysis of the *proportions* of the search space (w.r.t. the number of satisfying and non-satisfying assignments) and how the assignment to a singular variable changes these proportions.

Since the solution rate is based on propositional model counting (#SAT), its calculation is generally #P-complete [14] and therefore algorithmically hard. However, under the assumption that two distinct boolean formulas share no input variable (*independent variables assumption*) the number of models of their conjunction and disjunction can efficiently be calculated by recursion. For *almost all expressions* in Kodkod models this assumption holds, since a relation usually participates in an expression only once, and each relation contributes its unique set of primary variables.

Corollary 3.2 (Recursive Model Counting)**.** Let f and g be boolean formulas. Assuming that $\text{vars}(f) \cap \text{vars}(g) = \emptyset$ the following equations hold

$$\begin{aligned} \#(f \wedge g) &= \#f \cdot \#g \\ \#(f \vee g) &= 2^{|f|+|g|} - \#\neg f \cdot \#\neg g \end{aligned}$$

Utilization of the *independent variables assumption* leads to a simple recursive formulation of the solution rate that is crucial for efficient calculation.

Lemma 3.1 (Recursive Solution Rate)**.** Let f and g be boolean formulas. Assuming that both formulas have their unique set of input variables ($^* \text{vars}(g) \cap \text{vars}(f) = \emptyset$) the solution rate of formulas where f and g occur as sub-formulas can recursively be expressed in terms of the solution rates of its child formulas.

$$\rho(\neg g) = 1 - \rho(g) \tag{3.1a}$$

$$\rho(f \wedge g) \overset{*}{=} \rho(f) \cdot \rho(g) \tag{3.1b}$$

$$\rho(f \vee g) \overset{*}{=} 1 - \rho^-(f) \cdot \rho^-(g) = \rho(f) - \rho(f) \cdot \rho(g) + \rho(g) \tag{3.1c}$$

Proof. 3.1a

$$\rho(\neg g) = \frac{\#(\neg g)}{2^{|g|}} = \frac{\#\neg g}{2^{|g|}} = \rho^-(g) = 1 - \rho(g)$$

□

Proof. 3.1b

$$\begin{aligned}\rho(f \wedge g) &= \frac{\#(f \wedge g)}{2^{|(f \wedge g)|}} \\ &\overset{*}{=} \frac{\#f \cdot \#g}{2^{|f|+|g|}} \\ &= \frac{\#f}{2^{|f|}} \cdot \frac{\#g}{2^{|g|}} = \rho(f) \cdot \rho(g)\end{aligned}$$

□

Proof. 3.1c

$$\begin{aligned}\rho(f \vee g) &= \frac{\#(f \vee g)}{2^{|(f \vee g)|}} \\ &\overset{*}{=} \frac{2^{|f|+|g|} - \#f \cdot \#g}{2^{|f|+|g|}} \\ &= 1 - \frac{\#f \cdot \#g}{2^{|f|+|g|}} \\ &= 1 - \frac{\#f}{2^{|f|}} \cdot \frac{\#f}{2^{|f|}} = 1 - \rho^{-}(f) \cdot \rho^{-}(g) \\ &= \rho(f) - \rho(f) \cdot \rho(g) + \rho(g)\end{aligned}$$

□

Example 3.2 (Recursive Calculation)**.** *Given again the formula $g = a \vee (b \wedge \neg c)$ where $a, b, c \in \mathcal{V}$ are variables, the solution rate can also be calculated by recursive application of Lemma 3.1.*

$$\begin{aligned}\rho(g) &= \rho(a) - \rho(a) \cdot \rho(b \wedge \neg c) + \rho(b \wedge \neg c) \\ &= \rho(a) - \rho(a) \cdot \rho(b) \cdot (1 - \rho(c)) + \rho(b) \cdot (1 - \rho(c)) \\ &= \frac{1}{2} - \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{5}{8}\end{aligned}$$

Derived Formulas

The solution rate provides information about the proportions of the search space. An assignment of a variable v splits the search space of a formula g into that of the two derived formulas g_v and $g_{\bar{v}}$ with usually different proportions.

Definition 3.3 (Derived Formula)**.** For each $g \in \mathcal{F}$ and variable $v \in \mathcal{V}$ let g_v denote the formula derived from g where every occurrence of v is replaced by 1 and let $g_{\bar{v}}$ denote the *derived formula* where v is replaced by 0.

Derived formulas are pure syntactic constructs that are used to simulate partial assignments. Defining it at the syntax level has the advantage that the input variable used for the derivation vanishes. Thus g, g_v and $g_{\bar{v}}$ are distinct formulas and $\text{vars}(g_v) = \text{vars}(g_{\bar{v}}) = \text{vars}(g) \setminus \{v\}$.

Example 3.3 (Solution Rate of Derived Formulas)**.** *Given the formula $g = a \vee (b \wedge \neg c)$ where $a, b, c \in \mathcal{V}$, the solution rates of the derived formulas are given by*

$$\rho(g_a) = 1 \qquad \rho(g_{\bar{a}}) = \frac{1}{4}$$
$$\rho(g_b) = \frac{3}{4} \qquad \rho(g_{\bar{b}}) = \frac{2}{4}$$
$$\rho(g_c) = \frac{2}{4} \qquad \rho(g_{\bar{c}}) = \frac{3}{4}$$

We will refer to the solution rate of derived formulas later on with the term *derived solution rate*. So for each variable $v \in \mathcal{V}$ and formula $g \in \mathcal{F}$ there are two possible derived solution rates, the solution rate of g_v denoted by $\rho(g_v)$ and the solution rate of $g_{\bar{v}}$ denoted by $\rho(g_{\bar{v}})$. Note that if $v \notin \text{vars}(g)$ it always holds that $\rho(g_v) = \rho(g_{\bar{v}}) = \rho(g)$.

Lemma 3.2 (Coherence)**.** For any formula g and input variable v the following equation holds

$$\rho(g_v) + \rho(g_{\bar{v}}) = 2\rho(g)$$

Proof.

$$\rho(g_v) + \rho(g_{\bar{v}}) = \frac{\#g_v}{2^{|g_v|}} + \frac{\#g_{\bar{v}}}{2^{|g_{\bar{v}}|}} = \frac{\#g_v + \#g_{\bar{v}}}{2^{|g|-1}} = \frac{2\#g}{2^{|g|}} = 2\rho(g)$$

□

From the coherence lemma an upper bound for derived solution rates can be directly formulated. We need those bounds as well as the coherence lemma later on for fuzzy optimization techniques.

$$\rho(g_v) \leq 2\rho(g)\,, \qquad \rho(g_{\bar{v}}) \leq 2\rho(g)$$

Sensitivity

Given a formula $g \in \mathcal{F}$ and a variable $v \in \text{vars}(g)$ it is now possible to measure the sensitivity of g on assignments to v by comparing the search space proportions of the two derived formulas g_v and $g_{\bar{v}}$. The idea is that variables that are capable of setting a formula constant (or almost constant) are more dominant than others whose assignment

has little effect on the search space proportions of their derived formulas. Note that for variables that are capable of setting a formula constant, either one of its derived solution rates is either 1 or 0.

Definition 3.4 (Sensitivity)**.** The sensitivity of a formula g to assignments of a variable v is the difference between the solution rates of g_v and $g_{\bar{v}}$.

$$\sigma(g, v) = \rho(g_v) - \rho(g_{\bar{v}})$$

Example 3.4 (Sensitivity)**.** *Given the formula $g = a \vee (b \wedge \neg c)$ its sensitivity to each variable $a, b, c \in \mathcal{V}$ is given by*

$$\sigma(g, a) = \frac{3}{4} \qquad \sigma(g, b) = \frac{1}{4} \qquad \sigma(g, c) = -\frac{1}{4}$$

Note that variable a in example 3.4 clearly dominates the given formula $g = a \vee (b \wedge \neg c)$, since it is capable of setting the whole formula constant, and thereby dominating all assignments to the other variables b and c. This is reflected in the higher magnitude of the formula's sensitivity to a.

Solution Rate of Kodkod specific Boolean Formulas

Under application of Lemma 3.1 the solution rates of the matrix entries corresponding to Kodkod expressions can be expressed recursively in terms of the solution rates of the matrix entries of their child expressions. For all relations R the solution rate of the corresponding matrix entries $r_{\{i_n\}}$ is given by

$$\rho\big(r_{\{i_n\}}\big) = \begin{cases} 0 & \text{if } \langle a_{i_0}, a_{i_1}, \ldots, a_{i_n}\rangle \notin R_u \\ 1 & \text{if } \langle a_{i_0}, a_{i_1}, \ldots, a_{i_n}\rangle \in R_l \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

For union, intersection, and difference expressions E, the solution rates of the corresponding matrix entries $e_{\{i_n\}}$ is given by

$$\rho\big(e_{\{i_n\}}\big) = \begin{cases} \rho\big(f_{\{i_n\}}\big) \cdot \rho\big(g_{\{i_n\}}\big) & \text{if } E = \text{`}\mathtt{F} \cap \mathtt{G}\text{'} \\ 1 - \rho^{-}\big(f_{\{i_n\}}\big) \cdot \rho^{-}\big(g_{\{i_n\}}\big) & \text{if } E = \text{`}\mathtt{F} \cup \mathtt{G}\text{'} \\ \rho\big(f_{\{i_n\}}\big) \cdot \rho^{-}\big(g_{\{i_n\}}\big) & \text{if } E = \text{`}\mathtt{F} \setminus \mathtt{G}\text{'} \end{cases}$$

For product and join expressions E the solution rate of the matrix entries $e_{\{i_n\},\{j_m\}}$ is given by

$$\rho\big(e_{\{i_n\},\{j_m\}}\big) = \begin{cases} \rho\big(f_{\{i_n\}}\big) \cdot \rho\big(g_{\{j_m\}}\big) & \text{if } E = \text{`F} \to \text{G'} \\ 1 - \prod_k \big(1 - \rho\big(f_{\{i_n\},k}\big) \cdot \rho\big(g_{k,\{j_m\}}\big)\big) & \text{if } E = \text{`F.G'} \end{cases}$$

For override expressions E = 'F ++ G' the solution rate for each matrix entry $e_{i_0,\{i_n\}}$ is given by

$$\rho\big(e_{i_0,\{i_n\}}\big) = 1 - \rho^-\big(g_{i_0,\{i_n\}}\big) \cdot \rho^-\big(f_{i_0,\{i_n\}}\big) \cdot \prod_{j \neq i} \rho\big(g_{i_0,\{j_n\}}\big)$$

Dominance

The sensitivity is a function that assigns to every tuple $(g, v) \in \mathcal{F} \times \mathcal{V}$ a value in the closed interval $[-1, 1]$. The sign preserves polarity information that indicates which polarity of an assignment produces the largest number of satisfying solutions. The magnitude of the sensitivity can be used to measure the dominance of a variable in a formula.

Definition 3.5 (Dominance)**.** The dominance $\delta(v, g)$ of a variable v in a formula g is the magnitude of the formula's sensitivity on v

$$\delta(v, g) = |\sigma(g, v)|$$

Now we can finally augment our definition mentioned at the beginning, where we defined the average dominance of a relation in an expression (definition 3.1), by the above definition of the dominance of a singular variable in a propositional formula.

Definition 3.6 (Average Dominance (Final))**.** The average dominance of a relation R (with primary variables $\{r_{\{i_n\}}\}$) in an expression E (with matrix entries $\{e_{\{j_m\}}\}$) is the aggregated dominance $\delta\big(r_{\{i_n\}}, e_{\{j_m\}}\big) = \big|\sigma\big(e_{\{j_m\}}, r_{\{i_n\}}\big)\big|$ of its primary variables $r_{\{i_n\}}$ in the boolean formulas $e_{\{j_m\}}$.

$$\mathrm{dom}(R, E) = \frac{1}{\big|\{r_{\{i_n\}}\}\big|} \sum_{\{i_n\}} \sum_{\{j_m\}} \big|\sigma\big(e_{\{j_m\}}, r_{\{i_n\}}\big)\big|$$

Example

We now show how the algorithm would distribute scores to the highly constraint expressions in our introductory example 1.1 based on the average dominance of relations in expressions (definition 3.6). First we look at the constraint ($legal \subseteq trans$). Recapitulate the translation of the relations *legal* and *trans* (Table 3.2).

$$trans = \begin{bmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{bmatrix} \quad legal = \begin{bmatrix} l_{00} & l_{01} \\ l_{10} & l_{11} \end{bmatrix}$$

Table 3.2: Translation Revisited 1

The algorithm would distribute an unmoderated score of 1 to either one of the participating expressions, since the relations are directly constraint and the average dominance of a relation in itself is 1. We can calculate that with the given formula.

$$\begin{aligned}
\mathrm{dom}(trans, trans) =& \frac{1}{|trans|} \sum_{v \in trans} \sum_{f \in trans} |\sigma(f, v)| \\
=& \frac{1}{4} \cdot \big(|\sigma(t_{00}, t_{00})| + \overbrace{|\sigma(t_{00}, t_{01})| + |\sigma(t_{00}, t_{10})| + |\sigma(t_{00}, t_{11})|}^{=0} + \\
& |\sigma(t_{01}, t_{01})| + \overbrace{|\sigma(t_{01}, t_{00})| + |\sigma(t_{01}, t_{10})| + |\sigma(t_{01}, t_{11})|}^{=0} + \\
& |\sigma(t_{10}, t_{10})| + \overbrace{|\sigma(t_{10}, t_{00})| + |\sigma(t_{10}, t_{01})| + |\sigma(t_{10}, t_{11})|}^{=0} + \\
& |\sigma(t_{11}, t_{11})| + \overbrace{|\sigma(t_{11}, t_{00})| + |\sigma(t_{11}, t_{01})| + |\sigma(t_{11}, t_{10})|}^{=0} \big) \\
=& \frac{1}{4} \cdot \big(\overbrace{|\sigma(t_{00}, t_{00})|}^{=1} + \overbrace{|\sigma(t_{01}, t_{01})|}^{=1} + \overbrace{|\sigma(t_{10}, t_{10})|}^{=1} + \overbrace{|\sigma(t_{11}, t_{11})|}^{=1} \big) = 1
\end{aligned}$$

This is a very trivial case and we present the details to show how the result corresponds to intuition. Most of the summands above evaluate to zero, since the primary variables in *trans* are not sensitive to each other. Trivially, for any variable $v \in \mathcal{V}$ and formula $f \in \mathcal{F}$ when $v \notin \mathrm{vars}(f)$ it holds that $\sigma(f, v) = 0$. And for all variables $v \in \mathcal{V}$ it holds that its sensitivity to assignments to itself is $\sigma(v, v) = 1 - 0 = 1$. Like the above calculation for *trans* the average dominance of *legal* in itself is $\mathrm{dom}(legal, legal) = 1$.

Now we have a look at the second constraint ($legal \cap iden = self$). Recapitulate the translations of $legal \cap iden$ and $self$ (Table 3.3). For the built-in constant relation *iden* no variables are generated, so there is no dominance to be calculated for *iden*. The relation *legal* gets 4 variables, but only the diagonal elements of them are being

$$legal \cap iden = \begin{bmatrix} l_{00} \wedge 1 & l_{01} \wedge 0 \\ l_{10} \wedge 0 & l_{11} \wedge 1 \end{bmatrix} = \begin{bmatrix} l_{00} & 0 \\ 0 & l_{11} \end{bmatrix}$$

$$self = \begin{bmatrix} s_{00} & 0 \\ 0 & s_{11} \end{bmatrix}$$

Table 3.3: Translation Revisited 2

constraint by equivalence due to the intersection with *iden*. The relation $self$ gets 2 variables and both of them are constraint by equivalence. We will see how this is reflected in the average dominance by calculating it.

$$\begin{aligned}
\mathrm{dom}(legal, legal \cap iden) =& \frac{1}{|legal|} \sum_{v \in legal} \sum_{f \in legal \cap iden} |\sigma(f, v)| \\
=& \frac{1}{4} \cdot \big(|\sigma(l_{00}, l_{00})| + \overbrace{|\sigma(l_{00}, l_{01})| + |\sigma(l_{00}, l_{10})| + |\sigma(l_{00}, l_{11})|}^{=0} + \\
& \overbrace{|\sigma(0, l_{01})| + |\sigma(0, l_{00})| + |\sigma(0, l_{10})| + |\sigma(0, l_{11})|}^{=0} + \\
& \overbrace{|\sigma(0, l_{10})| + |\sigma(0, l_{00})| + |\sigma(0, l_{01})| + |\sigma(0, l_{11})|}^{=0} + \\
& |\sigma(l_{11}, l_{11})| + \overbrace{|\sigma(l_{11}, l_{00})| + |\sigma(l_{11}, l_{01})| + |\sigma(l_{11}, l_{10})|}^{=0} \big) \\
=& \frac{1}{4} \cdot \big(\overbrace{|\sigma(l_{00}, l_{00})|}^{=1} + \overbrace{|\sigma(l_{11}, l_{11})|}^{=1} \big) = \frac{1}{2}
\end{aligned}$$

The average dominance of *legal* in the intersection is $\frac{1}{2}$ and that is exactly the fraction of variables in *legal* that participates in the constraint. However, the average dominance of $self$ is again just 1 since all of its variables participate in the constraint. We just skip the zeros in the following equation.

$$\mathrm{dom}(self, self) = \frac{1}{2} \cdot \big(|\sigma(s_{00}, s_{00})| + |\sigma(s_{11}, s_{11})| \big) = 1$$

Again those are trivial cases, but they perfectly demonstrate how the average dominance works for arbitrary expressions. The example 3.4 we used for the deduction of a formula's sensitivity to singular assignments gives a glimpse into what the dominance of relations in more complex expressions looks like.

Formulas like that in example 3.4 ($a \vee (b \wedge \neg c)$) emerge in the translation of expressions like $A \cup (B \setminus C)$ where A, B and C are relations and a, b and c are primary variables. Depending on the bounds of those relations some tuples may contribute a dominance corresponding exactly to the values calculated in example 3.4. Others might contribute a dominance of 0 or 1 depending on the relation bounds.

3.2.4 Fast Fuzzy Calculation of Sensitivity-based Dominance

To calculate the dominance of a relation R in an expression E, a naive implementation of the nested sum in definition 3.6 would have to iterate over all propositional formulas in the expression's matrix and calculate the derived solution rates for each primary variable of R. There exist several optimizations to that naive approach.

Crisp Expression Bounds

Most matrix entries of an expression $E \subseteq U^n$ are constant. By recursive descent to the enclosed relations and based on their bounds, the expression specific upper and lower bounds can be calculated. For each expression E and its boolean matrix $(e_{\{i_d\}})$ its upper bound is given by $E_u = \{\langle a_{i_0}, a_{i_1}, \ldots, a_{i_d}\rangle \mid e_{\{i_d\}} \not\equiv 0\}$. Accordingly its lower bound is given by $E_l = \{\langle a_{i_0}, a_{i_1}, \ldots, a_{i_d}\rangle \mid e_{\{i_d\}} \equiv 1\}$. Having calculated the expression bounds, dominance calculation can safely be restricted to the non-constant tuples in $E_u \setminus E_l$.

Leaf Variables

Without further inspection of an expression it might not be obvious which subset of a relation's primary variables is input to which entry in an expression's boolean matrix. On the other hand, iteration over all primary variables for each entry in $E_u \setminus E_l$ to calculate its sensitivity to that variable — thereby getting many zeros — is too expensive. The presented algorithm makes no eager assumption about the participation of variables, and descents into E only once for each tuple $t \in E_u \setminus E_l$. In a list it keeps track of the variables encountered in the recursion, and the derived solution rates for all of them can be calculated in parallel.

Coherence

By refinement of Lemma 3.2 it is sufficient to calculate only *one* derived solution rate for each variable. If the *plain* solution rate of the formula is known, the sensitivity is

then implied by the coherence.

$$\begin{aligned} \rho(g_v) + \rho(g_{\bar{v}}) = 2\rho(g) &\implies \\ \rho(g_v) - \rho(g_{\bar{v}}) = 2\rho(g) - 2\rho(g_{\bar{v}}) &\implies \\ \underline{\sigma(g,v) = 2(\rho(g) - \rho(g_{\bar{v}}))} & \end{aligned} \tag{3.2}$$

Fuzzy Expression Bounds

As expressions' depth and dimensionality increase and especially in case of matrix multiplications the number of input variables for each formula in the expression bounds increases exponentially. Calculation of all derived solution rates for those variables therefore is algorithmically intractable. Fortunately, the sensitivity of such large formulas to singular assignments is generally small enough to be safely ignored. As a rule of thumb we can state that the wider an or-gate gets, the closer its solution rate gets to 1 and the wider an and-gate gets, the closer its solution rate gets to 0. With respect to the solution rate of a formula we can proof two upper bounds (Lemma 3.3 and 3.4) for the magnitude of its sensitivity to any variable.

Lemma 3.3 (Sensitivity Bound Zero)**.** For all variables $v \in \mathcal{V}$ the double of the solution rate of g is an upper bound for the magnitude of the sensitivity of g on v.

$$|\sigma(g,v)| \leq 2\rho(g)$$

Proof. The proof has to be taken two ways. First we proof that $\sigma(g,v) \leq 2\rho(g)$ and second we proof that $\sigma(g,v) \geq -2\rho(g)$. We start with the refinement of the coherence lemma we already got from equation 3.2.

$$\sigma(g,v) = 2\rho(g) - 2\rho(g_{\bar{v}}) \implies \underline{\sigma(g,v) \leq 2\rho(g)}$$

For the second part of the proof we utilize the inequality $\rho(g_{\bar{v}}) \leq 2\rho(g)$ which follows directly from the coherence lemma 3.2.

$$\sigma(g,v) = 2\rho(g) - 2\rho(g_{\bar{v}}) \implies \sigma(g,v) \geq 2\rho(g) - 4\rho(g) \implies \underline{\sigma(g,v) \geq -2\rho(g)}$$

□

For the second bound (Lemma 3.4) we need some intermediate steps to be able to proof it. First we give an inverted definition of sensitivity (corollary 3.3) and then follows an inverted formulation of the coherence lemma (corollary 3.4).

Corollary 3.3 (Inverted Sensitivity)**.** The sensitivity can be rewritten in its inverted form.

$$-\sigma(g,v) = \rho^-(g_v) - \rho^-(g_{\bar{v}})$$

Proof.

$$\begin{aligned}
\sigma(g,v) = \rho(g_v) - \rho(g_{\bar{v}}) &\implies \\
-\sigma(g,v) = \rho(g_{\bar{v}}) - \rho(g_v) &\implies \\
-\sigma(g,v) = (1-\rho(g_v)) - (1-\rho(g_{\bar{v}})) &\implies \\
\underline{-\sigma(g,v) = \rho^-(g_v) - \rho^-(g_{\bar{v}})} &
\end{aligned}$$

□

Corollary 3.4 (Inverted Coherence)**.** The coherence lemma can be rewritten in its inverted form.

$$\rho^-(g_v) + \rho^-(g_{\bar{v}}) = 2\rho^-(g) \tag{3.3}$$

Proof.

$$\rho^-(g_v) + \rho^-(g_{\bar{v}}) = \frac{\#g_v}{2^{|g_v|}} + \frac{\#g_{\bar{v}}}{2^{|g_{\bar{v}}|}} = \frac{\#g_v + \#g_{\bar{v}}}{2^{|g|-1}} = \frac{2\#g}{2^{|g|}} = 2\rho^-(g)$$

□

Lemma 3.4 (Sensitivity Bound One)**.** For all variables $v \in \mathcal{V}$ the double of the inverted solution rate of g is an upper bound for the magnitude of the sensitivity of g on v.

$$|\sigma(g,v)| \leq 2(1-\rho(g)) \tag{3.4}$$

Proof. Again the proof is taken two ways. First we prove that $-\sigma(g,v) \leq 2(1-\rho(g))$ and second we prove that $-\sigma(g,v) \geq -2(1-\rho(g))$. We start with the inverted definition of sensitivity we gave in corollary 3.3.

$$\begin{aligned}
\rho^-(g_v) + \rho^-(g_{\bar{v}}) = 2\rho^-(g) &\implies \\
\rho^-(g_v) - \rho^-(g_{\bar{v}}) = 2\rho^-(g) - 2\rho^-(g_{\bar{v}}) &\implies \\
-\sigma(g,v) = 2\rho^-(g) - 2\rho^-(g_{\bar{v}}) &\implies \\
-\sigma(g,v) \leq 2\rho^-(g) &\implies \\
\underline{-\sigma(g,v) \leq 2(1-\rho(g))} &
\end{aligned}$$

For the second part of the proof we utilize the inequality $\rho^-(g_{\bar{v}}) \leq 2\rho^-(g)$ which follows directly from the inverted coherence given in corollary 3.4.

$$\begin{aligned} -\sigma(g,v) = 2\rho^-(g) - 2\rho^-(g_{\bar{v}}) &\implies \\ -\sigma(g,v) \geq 2\rho^-(g) - 4\rho^-(g) &\implies \\ -\sigma(g,v) \geq -2\rho^-(g) &\implies \\ \underline{-\sigma(g,v) \geq -2(1-\rho(g))} & \end{aligned}$$

□

We conclude that, when the solution rate of a formula is very small or very close to one, its sensitivity to any assignment is very small. Since most highly constraint expressions contribute relatively big scores, small values (≤ 0.01) are hardly recognized in the accumulated score. Furthermore, such small scores never change the order induced by the accumulated weight.

Our presented algorithm incorporates an early detection of such formulas by first calculating the solution rate for each tuple in the expression bounds. We introduced a constant ϵ (feasible values are e.g. $10^{-3} \leq \epsilon \leq 10^{-2}$), and refer to ϵ as the amount of fuzziness. If the solution rate is smaller than ϵ then the tuple is removed from the expression's upper bound. If the solution rate is greater than $1 - \epsilon$ the tuple is added to the expression's lower bound. The algorithm thus further treats those formulas like constants, and expensive parts of the calculation that would only add microscopic weights are skipped.

Algorithm 3.1 summarizes the procedure. First the crisp upper and lower bounds of the expressions are determined by recursion to the relations (line 2). Then for each tuple in the crisp bounds the solution rate is calculated. A predefined constant specifies the amount of fuzziness (line 1). If a tuple's solution rate is very close to one (line 7) it is added to the expression's fuzzy lower bound. If a tuple's solution rate is very close to zero (line 5) it is removed from the expression's fuzzy upper bound. Now for dominance calculation only those tuples are respected that add a reasonably large score to the sum. For each tuple and each variable one derived solution rate is calculated (line 10). Based on the tuple's solution rate and a variable's derived solution rate the dominance of that variable can be calculated (line 13). Depending on which relation the variable belongs to (line 12) that singular dominance is added to the relation's dominance. The average dominance of one relation is obtained by division through the total number of variables of that relation (line 15).

Algorithm 3.1: CalculateDominances(exp)

Data: exp: An Expression Node of the AST
Result: The dominance of each leaf relation in exp

```
1  calculate crispUpper and crispLower bounds of exp
2  ε ← 10^-2
3  fuzzyUpper ← crispUpper
4  fuzzyLower ← crispLower
5  foreach tuple ∈ crispUpper \ crispLower do
6      if ρ(tuple) ≤ ε then
7          remove tuple from fuzzyUpper
8      if ρ(tuple) ≥ 1 − ε then
9          add tuple to fuzzyLower
10 foreach tuple ∈ fuzzyUpper \ fuzzyLower do
11     derivedSolutionRates ← calculate ρ(tuple_{var=0}) for each var ∈ tuple
12     foreach ρ(tuple_{var=0}) ∈ derivedSolutionRates do
13         relation ← determine relation where var stems from
14         dom(relation, exp) +← 2 · |ρ(tuple) − ρ(tuple_{var=0})|
15 foreach relation do
16     dom(relation, exp) /= nvars(relation)
```

3.2.5 A Weighing Algorithm for Relations

Algorithm 3.2 performs a depth-first search on the AST and determines highly constraining formula nodes — a concept described in subsection 3.2.2. For each expression that is constraint by such a formula it distributes a weight to the participating relations (lines 10, 12 and 16). For each relation its individual weight is determined by calculation of its dominance in the currently examined expression. An additional penalty is calculated based on the width of the enclosing disjunction (line 4), taking into account the decaying constrainedness when several paths in the AST can satisfy the formula. For the same reason existential quantifiers are skipped. The disjunction size of existentially quantified formulas depends on the size of the quantified relation, so the penalty would be very high anyway. Apparently speaking of high constrainedness makes no sense when a formula only has to hold for an arbitrary small number of elements in a relation.

Three special cases are handled for comparison formulas. First of all (line 6), self-equivalences are simply skipped since they merely add tautological information as soon as Kodkod has created all primary variables. In fact they are only present to let Kodkod also create the variables for unconstrained relations. Also comparison formulas stating the equivalence between variables declared over the same relation are included in the check for self-equivalence. Another special case are subset constraints where the left expression's cardinality is bounded (line 7). In such cases the constraining effect of the subset formula is restrained to a few elements before violating the cardinality constraint; hence such subset-constraints are ignored. Comparison formulas where at least one expression is constant are skipped as well (line 8). Most of such constraints can be evaluated during translation.

Algorithm 3.2: DistributeScores(f)

Data: formula: A Formula Node of the AST
Result: Assigns a score to each non-constant relation

```
/* execute depth-first search on the subtree and distibute weights
   along expressions involved in formula-nodes considered to be
   highly constraining                                              */
```

1 **if** formula *is Existential Quantifier* **then return**

2 **foreach** child $\in$ formula **do**
3 `DistributeScores(child)`

4 w $\leftarrow$ number of disjoint paths
5 penalty $\leftarrow 2^{w}$

6 **if** formula *is ComparisonFormula* **then**
`/* formula is of the form` $exp_1 \subseteq exp_2$ `or` $exp_1 = exp_2$ `*/`
7 **if** $exp_1 \equiv exp_2$ **then skip**
8 **if** $\exists$ *CardinalityBound on* $exp_1 \wedge$ formula *is Subset* **then skip**
9 **if** exp_1 *is constant* **or** exp_2 *is constant* **then skip**
10 **foreach** leaf $\in exp_1$ **do**
11 weight(leaf) $\overset{+}{\leftarrow} \mathrm{dom}(leaf, exp_1) \cdot$ penalty
12 **foreach** leaf $\in exp_2$ **do**
13 weight(leaf) $\overset{+}{\leftarrow} \mathrm{dom}(leaf, exp_2) \cdot$ penalty
14 **else if** formula *is CardinalityBound* **then**
`/* formula is of the form` *one*(exp) `or` *no*(exp) `or` *lone*(exp) `or` $\neg$*some*(exp) `*/`
`/* or` intExpr $<$ c `or` intExpr $\leq$ c `or` intExpr $=$ c `*/`
`/* or` $\neg$(intExpr $>$ c) `or` $\neg$(intExpr $\geq$ c) `*/`
15 **foreach** exp *whose cardinality is bounded by* formula **do**
16 **foreach** leaf $\in$ exp **do**
17 weight(leaf) $\overset{+}{\leftarrow} \mathrm{dom}(leaf, exp) \cdot$ penalty

Chapter 4

Evaluation

In this section we present our evaluation of the several possibilities to prioritize primary variables. We start with giving the specifications of the machine that ran the benchmarks and the software we used in section 4.1. A comprehensive list of the benchmark problems used with additional details about the generated propositional formulas can be found in section 4.2, whereas section 4.3 contains a full runtime-analysis of the devised methods. In section 4.4 we analyze the overhead produced by the weighing algorithm with respect to different amounts of fuzziness. All devised methods are revisited and analyzed in section 4.5, with respect to the possibility of incrementing the bounds.

4.1 Specifications

For all experiments we used Kodkod version 1.5.1 and MiniSAT version 2.2. They were conducted on a Linux system (Ubuntu 11.04) with an AMD Athlon II X4 640 Processor and 3.6 GiB of main memory. In all experiments the default configuration options of Kodkod and MiniSAT were used, with the sole exception being the solver Kodkod uses, that would have been Sat4J by default.

4.2 Models

For evaluation we chose a set of Alloy models and increased the bounds to an extent where runtimes are expressive (≥ 1 second) yet small enough to allow for a large number of experiments. For a few problems however, the bounds could not be increased to achieve higher default runtimes, since their memory consumption ended up exceeding any sensible limits. Those were basically simple satisfiable problems.

The following table lists some details about the benchmark models we used in our experiments. All models are off-the-shelf Alloy models and most of them can be found in the set of examples shipped with Alloy Analyzer 4. We additionally used some models created by Eunsuk Kang (MIT) [19]. The table is organized as follows: In the first column the filename is given where the model is defined. Those marked with an asterisk can be found in the examples shipped with Alloy Analyzer 4. In the second column the command line is given, specifying the name of the constraint and the bounds. There might be several commands per file. For each command the total amount of clauses and variables after translation is given, followed by the clause-variable-ratio and the fraction of variables being primary. The last column specifies whether it is a satisfiable instance or not.

Table 4.1: Benchmark Problems

file	**command**	**clauses**	**variables**	**c/v**	**p/v**	
dijkstra.als*	Check DijkstraPrevents-Deadlocks for 12 State, 12 Process, 10 Mutex	217380	62925	3.45	0.05	unsat
	Run ShowDijkstra for 28 State, 7 Process, 7 Mutex	163698	57406	2.85	0.05	sat
opt-spantree.als*	Run BadLivenessTrace for 6 but 8 State	43771	21109	2.07	0.03	unsat
	Check Closure for 6 but 8 State	27650	14699	1.88	0.05	unsat
	Run SuccessfulRun for 10 State, exactly 12 Process, 8 Lvl	338063	138359	2.44	0.02	sat
	Run TraceWithoutLoop for 6 but 8 State	62006	56888	1.09	0.01	unsat
peterson.als*	Check NotStuck for 16 but 3 pid, 4 priority, 5 label_t	29470	12204	2.41	0.07	unsat
	Check Safety for 20 but 3 pid, 4 priority, 5 label_t	28640	12977	2.21	0.08	unsat
	Run ThreeRun for 64 but 5 pid, 5 priority, 6 label_t	205066	83937	2.44	0.08	sat

Benchmark Problems

file	command	clauses	variables	c/v	p/v	
	Run TwoRun for 64 but 5 pid, 5 priority, 6 label_t	201320	81999	2.46	0.08	sat
ringlead.als*	Check LeaderHighest for 8	176912	90731	1.95	0.04	unsat
	Check Liveness for 6 but 8 Msg, 2 Bool, 4 NodeState	97095	47680	2.04	0.05	unsat
	Run NeverFindLeader for 6 but 8 Tick, 2 Bool, 4 NodeState	86284	44452	1.94	0.05	sat
	Check OneLeader for 6 but 2 Bool, 4 NodeState	63912	33824	1.89	0.05	unsat
	Run SomeLeader for 5	33260	17858	1.86	0.06	unsat
stablemutexring.als*	Check Closure for 5 but 4 Process, 6 Val	12012	6266	1.92	0.04	unsat
	Run TraceShorterThanMaxSimpleLoop for 24 but 12 Process, 12 Val	346824	173383	2.00	0.03	sat
	Run TwoPrivileged for 20 but 10 Process, 10 Val	200071	95743	2.09	0.03	sat
stableorientring.als*	Check Closure for 6 but 12 Tick, 2 Bool, 18 Process	471365	199341	2.36	0.03	sat
	Run SomeState for 6 but 12 Tick, 2 Bool, 18 Process	452374	193645	2.34	0.03	sat
stableringlead.als*	Run CBadLivenessTrace for 9	328530	93238	3.52	0.01	unsat
	Check CMustConverge for 8 but 10 State	294418	85493	3.44	0.01	sat
	Run CTraceWithoutLoop for 12 but 8 State	474156	130373	3.64	0.01	sat
	Run ConvergingRun for 10 but 8 State	350296	96480	3.63	0.01	sat

Benchmark Problems

file	command	clauses	variables	c/v	p/v	
	Run DBadLivenessTrace for 6	99446	28660	3.47	0.01	sat
	Run DTraceWithoutLoop for 7	159315	51185	3.11	0.01	sat
chordbug-model.als*	Run FindSuccessorWorks for 5 but 2 State	106611	66625	1.60	0.01	unsat
	Run ShowMe3 for 12 but 5 State	1163524	488585	2.38	0.01	sat
	Run ShowMeCorrectSuccessorEg for 11 but 3 State	895295	364163	2.46	0.01	sat
	Check StrongerFindSuccessorWorks for 12 but 4 State	758912	293567	2.59	0.02	sat
chord2.als*	Check FindSuccessorWorks for 5 but 2 State, 4 Node, 4 NodeData	117556	79063	1.487	0.008	unsat
chord.als*	Check FindSuccessorWorks for 12 but 4 State	838020	330401	2.54	0.02	sat
	Check InjectiveIds for 10	588499	183858	3.20	0.03	unsat
	Check Same1 for 8 but 1 State	103321	51509	2.01	0.05	unsat
	Check Same2 for 8 but 1 State	98362	48592	2.02	0.05	unsat
	Check SameCPF for 5 but 2 State	89751	36712	2.44	0.02	unsat
	Check SameCPF1 for 5 but 2 State	59447	35510	1.67	0.02	unsat
	Check SameCPF2 for 6 but 2 State	91695	35176	2.61	0.03	unsat
	Check SameFC for 5 but 1 State	76361	28753	2.66	0.02	unsat
	Check SameFP for 9 but 2 State	585296	254883	2.30	0.01	sat
	Check SameFP1 for 6 but 1 State	95641	44817	2.13	0.02	unsat

Benchmark Problems

file	command	clauses	variables	c/v	p/v	
	Check SameFP2 for 5 but 1 State	51015	23800	2.14	0.03	unsat
	Run ShowMe1 for 13	2353289	780261	3.02	0.02	sat
	Run ShowMe1Node for 24 but 8 State, 8 Node	2958093	1203655	2.46	0.02	sat
	Run ShowMe2 for 12	2024161	749698	2.70	0.01	sat
	Run ShowMeCPF for 10 but 2 State	1885166	1362123	1.38	0.00	sat
	Run ShowMeFC for 16 but 8 State	3744016	1301265	2.88	0.02	sat
sync.als*	Run SyncSpecNotUnique for 7	40410	17659	2.29	0.02	unsat
com.als*	Check Theorem1 for 10	150064	76677	1.96	0.03	unsat
	Check Theorem2 for 10	149654	76677	1.95	0.03	unsat
	Check Theorem3 for 14	486222	245657	1.98	0.02	unsat
	Check Theorem4a for 10	149863	76785	1.95	0.03	unsat
	Check Theorem4b for 14	486657	245881	1.98	0.02	unsat
firewire.als*	Check AtMostOneElected for 7 Op, 2 Msg, 3 Node, 6 Link, 3 Queue, 9 State	33581	17813	1.89	0.03	unsat
	Check NoOverflow for 7 Op, 2 Msg, 3 Node, 6 Link, 5 Queue, 9 State	39282	20207	1.94	0.03	unsat
iolus.als*	Check OutsiderCantRead for 5 but 3 Member	129049	52659	2.45	0.01	unsat
abstract-Filesystem.als	Check WriteIdempotent for 5 but 5 int, 8 seq	32780	12236	2.68	0.04	unsat
	Check WriteLocal for 7 but 7 int, 14 seq	114000	40022	2.85	0.04	unsat
	Run run1 for 25 but 14 AbsFsys	655263	334273	1.96	0.04	sat
	Run run2 for 24 but 13 AbsFsys	676767	320168	2.11	0.04	sat

Benchmark Problems

file	command	clauses	variables	c/v	p/v	
block-Manager.als	Run gcTest for 8 but 8 int, 8 seq, 5 BlockManager, 5 GarbageCollector, 5 Device, 12 Block, 12 PhysicalAddr	558963	182740	3.06	0.02	sat
consensus-Paxos_v2.als	Run run1 for 10 but 2 Round, exactly 6 Acceptor, exactly 2 Proposer, exactly 8 Phase	1073762	329226	3.26	0.02	unsat
flash.als	Check ProgramIdempotent for 5	30586	11409	2.68	0.04	unsat
	Check ProgramLocal for 9	143103	52618	2.72	0.04	unsat
	Run eraseSome for 14	394872	159793	2.47	0.04	sat
	Run readSome for 16	570783	221189	2.58	0.05	sat
	Run writeSome for 12	286331	107017	2.68	0.04	sat
naming.als	Check WriteLocal for 9	3459520	6364003	0.54	0.00	unsat
	Check WriteRemoveConsistent for 6	397726	462610	0.86	0.01	unsat
	Run run1 for 10 but 5 Namespace	3143334	3430274	0.92	0.00	sat
naming-Object0.als	Check RenamePreservesStructure for 7 but 3 Namespace	385704	269440	1.43	0.01	unsat
	Check RenameSamePath for 5 but 3 Namespace	98991	66681	1.48	0.01	unsat
	Run remove for 10 but 5 Namespace	2875739	3469862	0.83	0.00	sat
	Run show_rename for 9 but 5 Namespace	1845678	2136894	0.86	0.00	sat
	Run write for 9 but 5 Namespace	1734984	2111058	0.82	0.00	sat
peterson2p.als	Check MutualExclusionSatisfied for 30 but exactly 2 System, exactly 8 Proc	235398	91714	2.57	0.06	unsat

Benchmark Problems

file	command	clauses	variables	c/v	p/v	
	Check ProgressGuaranteed for 30 but exactly 2 System, exactly 8 Proc	234725	89064	2.64	0.06	sat
	Run run1 for 30 but exactly 2 System, exactly 8 Proc	225259	86087	2.62	0.06	sat
planner.als	Run run1 for 28	3517236	1082092	3.25	0.02	sat
javatypes.als*	Check TypeSoundness for 4	34305	16579	2.07	0.04	unsat
lists.als*	Check reflexive for 10	54607	16379	3.33	0.03	unsat
	Run show for 28	1018738	270502	3.77	0.01	sat
	Check symmetric for 8	28939	9188	3.15	0.03	unsat
marksweep-gc.als*	Check Completeness for 7	40412	17170	2.35	0.05	unsat
	Check Soundness1 for 9	80651	30633	2.63	0.06	unsat
	Check Soundness2 for 7	40286	17163	2.35	0.05	unsat
views.als*	Check zippishOK for 8 but 10 State, 4 ViewType	799675	248128	3.22	0.01	sat

4.3 Experiments

In the first experiments (subsections 4.3.1 and 4.3.2) all primary variables were uniformly prioritized under application of the two methods proposed in section 3.1. In a third experiment (subsection 4.3.3) the weights calculated by algorithm 3.2 were used to initialize MiniSAT's activity array. All runtime comparisons have been conducted with a timeout of five minutes. We also compared all methods with respect to the maximum bound to be solved within 60 seconds (subsection 4.5).

4.3.1 Uniform Overriding

With Uniform Overriding, equal priorities were distributed to all primary variables and the overriding flag was activated as described in subsection 3.1.1. The solver was thereby forced to only assign primary variables. Due to the primary variables constituting an Independent Variable Set (IVS), the approach corresponds to Input Restricted Branching. The inner order of the primary variables remained untouched and MiniSAT's native

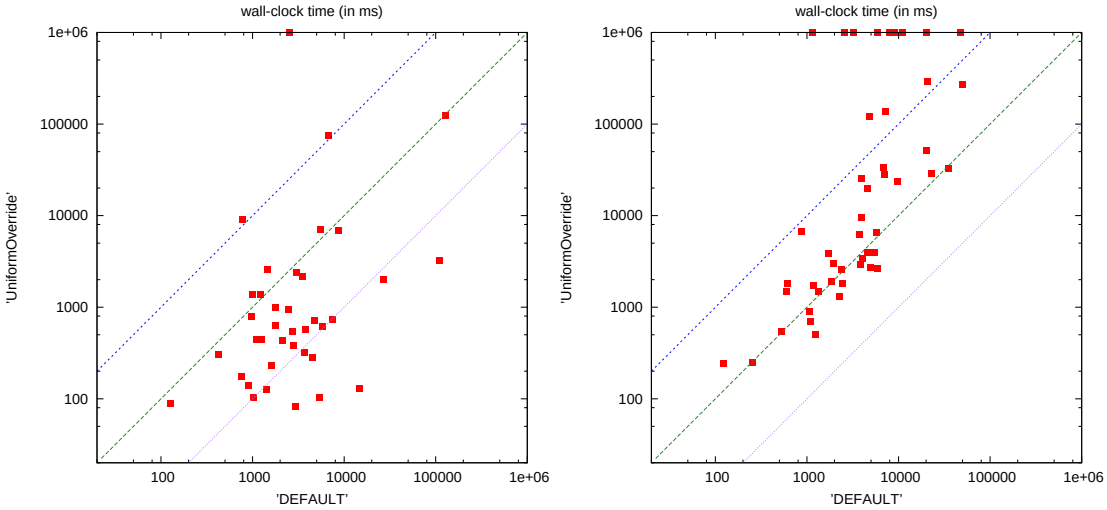

Figure 4.1: Uniform Activity Overriding (SAT / UNSAT)

activity-driven ordering was used to break ties.

The two scatter plots in figure 4.1 show a runtime comparison between the default runtime of MiniSAT and the runtimes of MiniSAT under application of uniform activity overriding on primary variables. The left side displays the speedup encountered on satisfiable instances. The considerable runtime improvement on satisfiable instances can be explained by the exponential decay in the size of the search space. However, there are also several runtime deteriorations, especially one timeout (indicated by a red dot at the upper border of the scatter-plot).

Many runtime deteriorations are encountered on unsatisfiable problems, among those were nine timeouts. As Järvisalo et al. [16] prove, CDCL solvers generally benefit from additional variables allowing them to produce exponentially shorter proofs. The runtime deterioration on unsatisfiable problems displayed in the right scatter plot of figure 4.1 empirically confirms their theoretical results.

4.3.2 Uniform Initialization

Now applying Uniform Initialization, we distributed equal priorities to all primary variables as before, except for mere initialization of MiniSAT's native activity array as described in subsection 3.1.2.

Figure 4.2 shows a runtime comparison between MiniSAT's default runtime and MiniSAT's runtime after uniform activity initialization of primary variables. The activity of all primary variables was initialized with a value of 3. The left scatter plot shows the

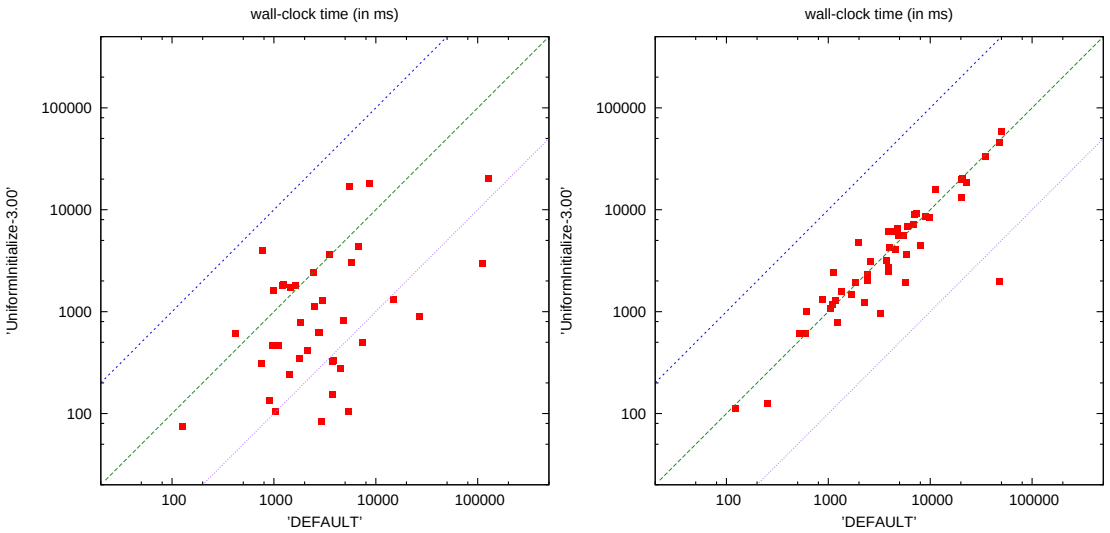

Figure 4.2: Uniform Activity Initialization (SAT / UNSAT)

performance gain on satisfiable problems, although the effect of activity initialization blurs as MiniSAT adjusts activities. Due to the blurring, the runtime deterioration on unsatisfiable problems vanishes as can be seen by the right scatter plot. Non-primary variables are still allowed for assignment, depending on how MiniSAT adjusts their activity.

As small activity values are already sufficient to boost the runtime, one might expect this effect to change or even converge to the results of activity overriding as higher values are selected for activity initialization. Figure 4.3 indicates that this assumption is wrong. On the contrary, the runtimes are in fact fluctuating depending on the chosen initial value. The cactus plots show the accumulated runtimes, separated by satisfiable and unsatisfiable benchmark problems, after activity initialization with the values 5, 233, 1346269 and 7778742049 (an arbitrary selection of fibonacci-numbers). The plots reveal no pattern correlating the size of the initial activity value with the runtime. The results show how rapidly the effect of activity initialization blurs, thereby also demonstrating how learning affects the search a runtime long.

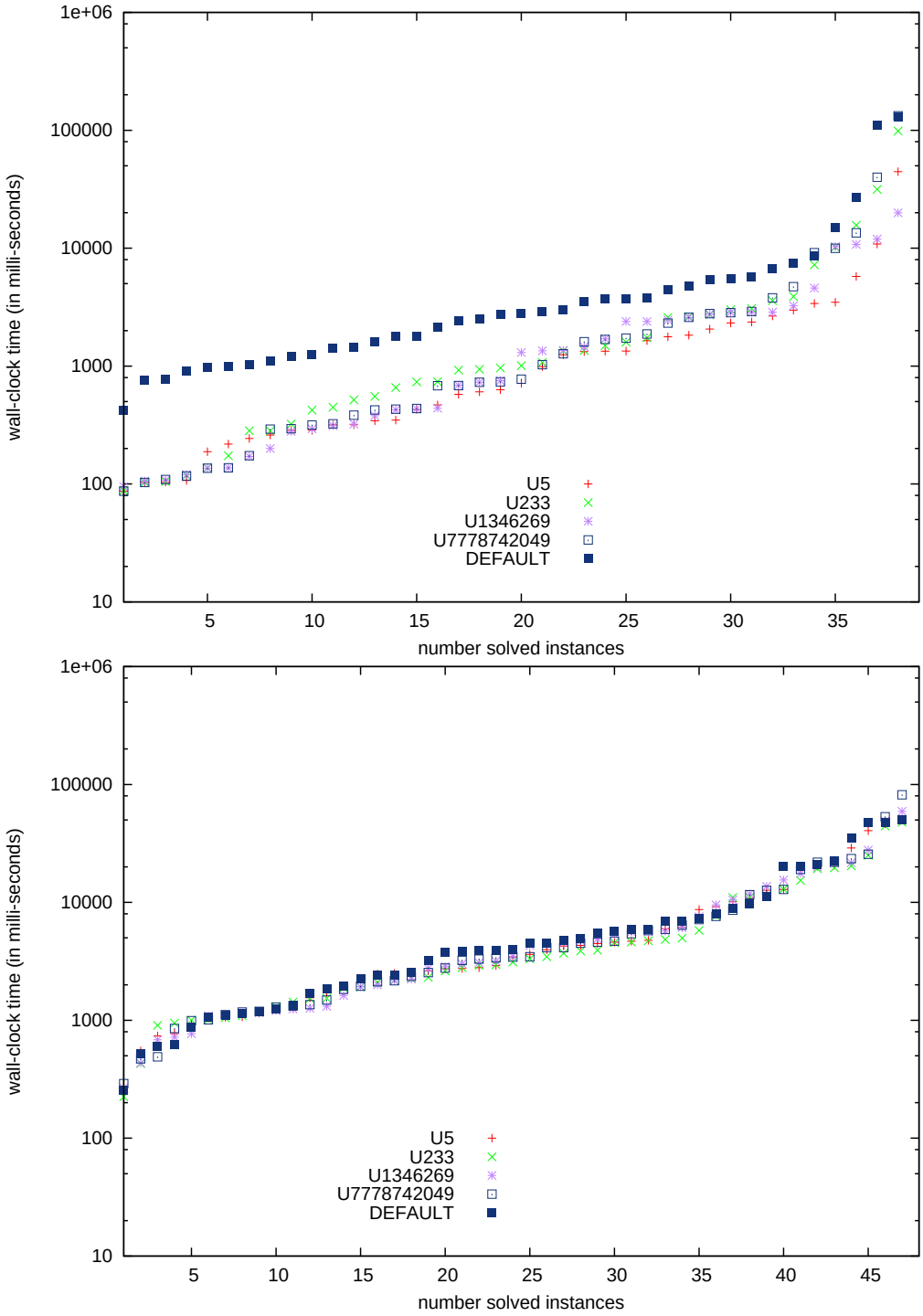

Figure 4.3: Uniform Activity Initialization with different initial values (SAT / UNSAT)

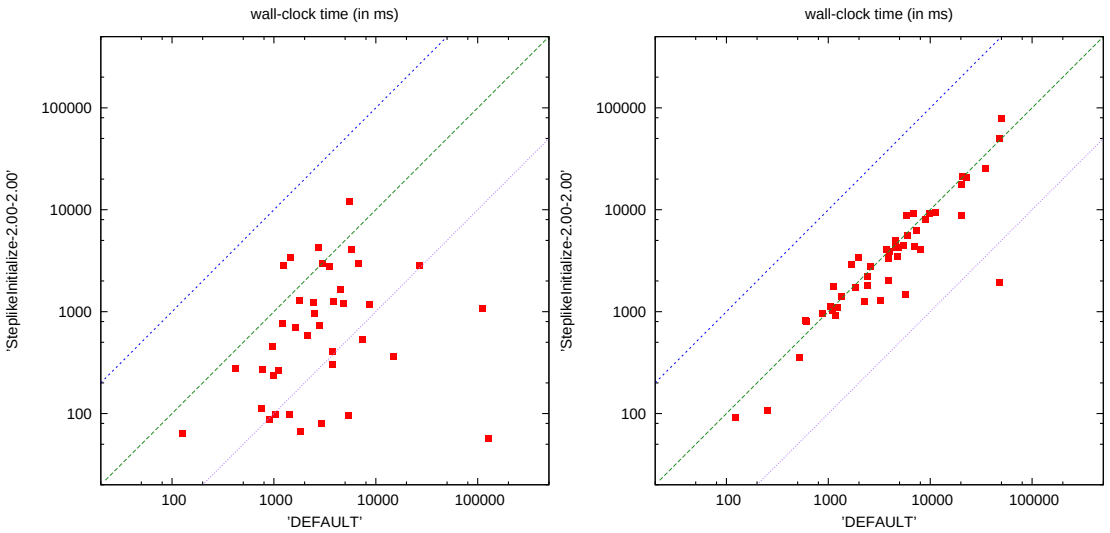

Figure 4.4: Step-like Activity Initialization (SAT / UNSAT)

4.3.3 Step-like Initialization

The set of relations induces a partition $\{V_0, \ldots, V_n\}$ on the primary variables such that V_i denotes the variables corresponding to relation R_i. Algorithm 3.2 generates for each relation R_i a weight $w(R_i)$ inducing an ordering such that $w(R_i) \geq w(R_{i+1})$. Since the absolute value of those weights strongly depends on the underlying problem, the weights were normalized with respect to the biggest weight. Let $w(R_0)$ be the biggest weight, then the normalized weight of each relation R_i is given by the formula $n(R_i) = w(R_i)/w(R_0)$. The normalized weights can now serve as input to a linear function $act(V_i) = b + f \cdot n(R_i)$ — with a scaling factor f and a baseline b — to calculate the initial activity for each primary variable $v \in V_i$ corresponding to a relation R_i. We could thus establish an initial ordering among the primary variables by activity initialization in a step-like manner. This differs from the uniform initialization approach, where the inner order of primary variables was not touched.

The two scatter plots in figure 4.4 show the results of a structure-based step-like activity initialization of primary variables. The parameter setting for the initialization function $act(V_i) = b + f \cdot n(R_i)$ was such that $b = 2$ and $f = 2$. Increased performance is encountered in both satisfiable and unsatisfiable problems. The performance increase on unsatisfiable problems is only slight. As we have seen in the former approaches, prioritizing primary variables is generally not a good idea for unsatisfiable problems. Prioritizing highly constraint primary variables, however, breaks the ties a bit.

As uniform activity initialization already performs very well, the scatter plot in fig-

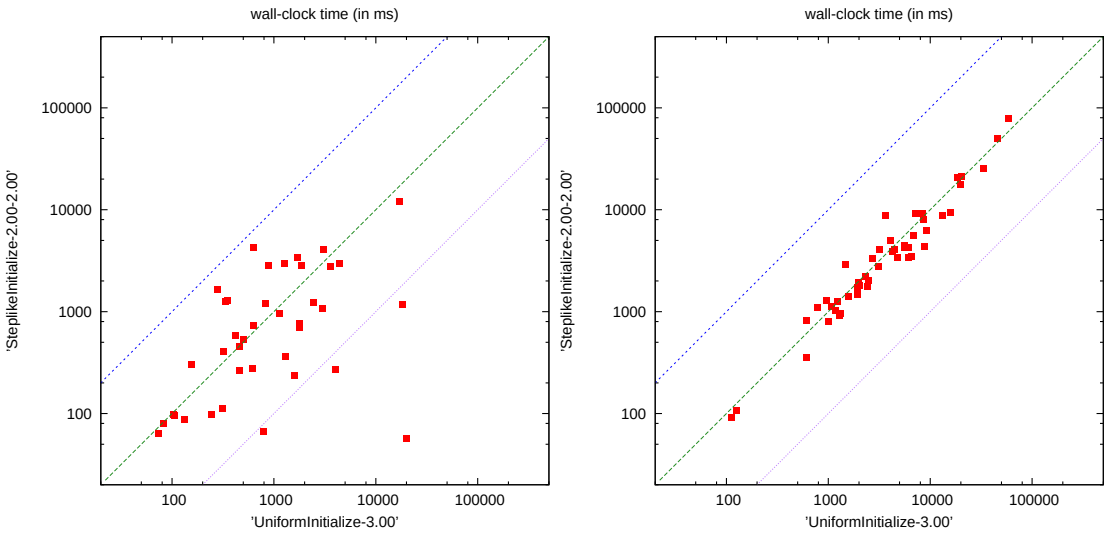

Figure 4.5: Uniform vs. Step-like Activity Initialization (SAT / UNSAT)

ure 4.5 displays a cross-comparison between runtimes after uniform activity initialization and structure-based step-like initialization. Most problems benefit from the more fine-grained structure-based initialization, but the improvement is small since there are also several deteriorations encountered compared to the uniform approach. Although hardly recognized, the step-like initialization approach outperforms uniform initialization on unsatisfiable problems.

Like in the Uniform Initialization approach we experimented with different parameter settings. The scatter plots in figure 4.6 display the accumulated runtimes with different scaling factors f, separated by satisfiable and unsatisfiable problems. Again the plots reveal no pattern correlating the size of the initial activity value with the runtime.

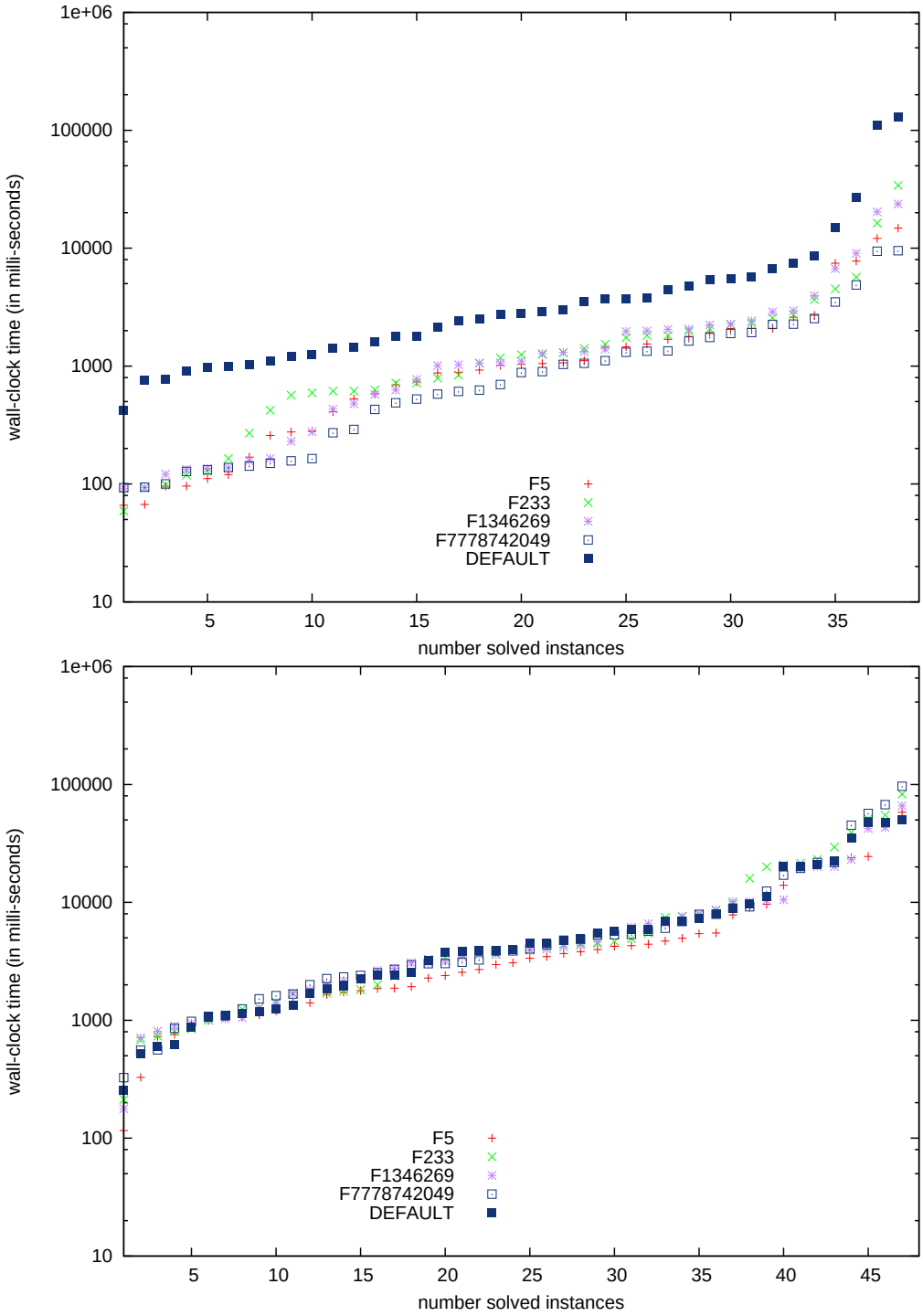

Figure 4.6: Step-like Activity Initialization with different scaling factors (SAT / UNSAT)

4.4 Overhead

The methods *Uniform Activity Overriding* and *Uniform Activity Initialization* produce close to no overhead due to the information about which variables are primary being already present in Kodkod's data-structures. However, the weighing algorithm (algorithms 3.1 and 3.2) used for *Step-like Activity Initialization* produces a significant amount of overhead. We have dealt with this problem by introducing a certain amount of fuzziness into the calculation of relation dominance and other algorithmic optimizations (subsection 3.2.4).

Table 4.2 lists the benchmark problems and the generated overhead caused by the weighing algorithm. The overhead strongly depends on the chosen amount of fuzziness, and as stated earlier, without a tiny amount of fuzziness the weighing approach would be algorithmically infeasible. The table compares weighing overhead with respect to two different amounts of fuzziness $\epsilon = 10^{-3}$ and $\epsilon = 10^{-2}$. As fuzziness increases, some runs of the weighing algorithm are further accelerated, so in our experiments we used the higher amount of fuzziness ($\epsilon = 10^{-2}$). The two rightmost columns show the highest accumulated score a relation achieved with respect to different amounts of fuzziness, in order to show how the precision is slightly deteriorating for some problems, and to give an impression of the order of magnitude of the accumulated scores. Runtimes are printed boldface if the runtime improvement gained by increased fuzziness was greater than 1 second. Scores are printed bold whenever increased fuzziness changed their precision.

Table 4.2: Overhead and highest score w.r.t. different amounts of fuzziness ϵ

problem	**overhead in ms**		**highest score**	
	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
dijkstra DijkstraPreventsDeadlocks	203	218	20.75	20.75
dijkstra ShowDijkstra	77	77	15.00	15.00
opt_spantree BadLivenessTrace	33	37	8.97	8.97
opt_spantree Closure	33	39	8.97	8.97
opt_spantree SuccessfulRun	66	57	2.00	2.00
opt_spantree TraceWithoutLoop	28	30	8.97	8.97
peterson NotStuck	25	27	18.08	18.08
peterson Safety	32	37	20.08	20.08
peterson ThreeRun	151	150	23.32	23.32
peterson TwoRun	147	148	22.32	22.32
ringlead LeaderHighest	61	473	25.81	25.81
ringlead Liveness	45	208	21.51	21.51
ringlead NeverFindLeader	129	132	22.37	22.37

Overhead and highest score w.r.t. different amounts of fuzziness ϵ

problem	**overhead in ms**		**highest score**	
	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
ringlead OneLeader	177	107	22.32	22.32
ringlead SomeLeader	125	53	20.60	20.60
stable_mutex_ring Closure	12	13	**7.36**	**7.51**
stable_mutex_ring TraceShorterTMSL	160	160	6.15	6.15
stable_mutex_ring TwoPrivileged	80	81	10.09	10.09
stable_orient_ring Closure	80	84	**25.75**	**26.18**
stable_orient_ring SomeState	80	84	**26.75**	**27.18**
stable_ringlead CBadLivenessTrace	5	5	1.00	1.00
stable_ringlead CMustConverge	4	4	0.25	0.25
stable_ringlead ConvergingRun	5	5	1.00	1.00
stable_ringlead CTraceWithoutLoop	6	6	1.00	1.00
stable_ringlead DBadLivenessTrace	10	9	0.50	0.50
stable_ringlead DTraceWithoutLoop	13	14	0.30	0.30
chord2 FindSuccessorWorks	795	848	**18.19**	**18.26**
chordbug FindSuccessorWorks	**117**	**2800**	**12.86**	**12.90**
chordbug ShowMe3	119	121	43.53	43.53
chordbug ShowMeCorrectSuccessorEg	76	79	43.53	43.53
chordbug StrongerFindSuccessorWorks	84	86	14.67	14.67
chord FindSuccessorWorks	116	110	14.67	14.67
chord InjectiveIds	87	96	16.07	16.07
chord Same1	54	49	36.69	36.69
chord Same2	53	49	**21.72**	**22.24**
chord SameCPF1	21	15	**34.49**	**36.30**
chord SameCPF	21	15	36.69	36.69
chord SameCPF2	33	24	12.71	12.71
chord SameFC	59	26	**19.60**	**19.98**
chord SameFP1	23	49	12.45	12.45
chord SameFP2	26	27	**29.20**	**29.46**
chord SameFP	45	73	**12.11**	**12.19**
chord ShowMe1	175	176	**15.06**	**15.50**
chord ShowMe1Node	341	405	**15.82**	**15.90**
chord ShowMe2	148	148	**20.61**	**20.64**
chord ShowMeCPF	75	450	27.59	27.59
chord ShowMeFC	327	308	**18.19**	**18.26**
com Theorem1	23	44	**9.13**	**9.49**
com Theorem2	24	44	**8.63**	**8.99**

Overhead and highest score w.r.t. different amounts of fuzziness ϵ

problem	**overhead in ms**		**highest score**	
	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
com Theorem3	61	73	**9.92**	**9.99**
com Theorem4a	24	44	**9.63**	**9.99**
com Theorem4b	62	72	**10.42**	**10.49**
firewire AtMostOneElected	13	17	19.78	19.78
firewire NoOverflow	16	23	19.78	19.78
iolus OutsiderCantRead	29	117	29.57	29.57
sync SyncSpecNotUnique	4	4	29.62	29.62
abstractFilesystem run1	191	195	9.00	9.00
abstractFilesystem run2	165	172	15.75	15.75
abstractFilesystem WriteIdempotent	7	6	7.88	7.88
abstractFilesystem WriteLocal	26	26	7.88	7.88
blockManager gcTest	137	147	23.91	23.91
consensusPaxos run1	276	298	**29.69**	**29.72**
flash eraseSome	128	131	13.12	13.12
flash ProgramIdempotent	9	8	**12.12**	**12.13**
flash ProgramLocal	34	41	8.24	8.24
flash readSome	195	199	7.88	7.88
flash writeSome	79	80	9.84	9.84
namingObject0 remove	426	455	**84.75**	**84.94**
namingObject0 RenamePreservesStructure	106	117	46.79	46.79
namingObject0 RenameSamePath	**44**	**4328**	**116.20**	**116.21**
namingObject0 show_rename	303	322	**73.42**	**73.44**
namingObject0 write	301	319	46.81	46.81
naming run1	331	386	**104.09**	**104.22**
naming WriteLocal	428	1069	87.35	87.35
naming WriteRemoveConsistent	92	92	87.35	87.35
Peterson2P MutualExclusionSatisfied	115	115	30.50	30.50
Peterson2P ProgressGuaranteed	114	116	30.50	30.50
Peterson2P run1	114	115	30.50	30.50
planner run1	248	250	14.00	14.00
javatypes TypeSoundness	27	75	16.76	16.76
lists reflexive	5	10	**7.42**	**7.61**
lists show	43	45	7.75	7.75
lists symmetric	5	6	8.80	8.80
marksweepgc Completeness	10	11	9.53	9.53
marksweepgc Soundness1	16	22	**8.93**	**8.95**

Overhead and highest score w.r.t. different amounts of fuzziness ϵ

problem	**overhead in ms**		**highest score**	
	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$
marksweepgc Soundness2	11	10	9.53	9.53
views zippishOK	95	95	**16.60**	**16.66**

Figure 4.7 displays the scatter plots comparing the runtimes of the step-like initialization approach to the default runtimes, this time including the overhead caused by the weighing algorithm. The benefit of a faster runtime is still recognized. However, as the benefit compared to the uniform initialization approach was already small, it vanishes as we include weighing overhead. Figure 4.8 shows the new cross-comparison including the overhead. Especially the small clusters of dots which were already close to the diagonal in figure 4.5 are now blurred across the diagonal in figure 4.8.

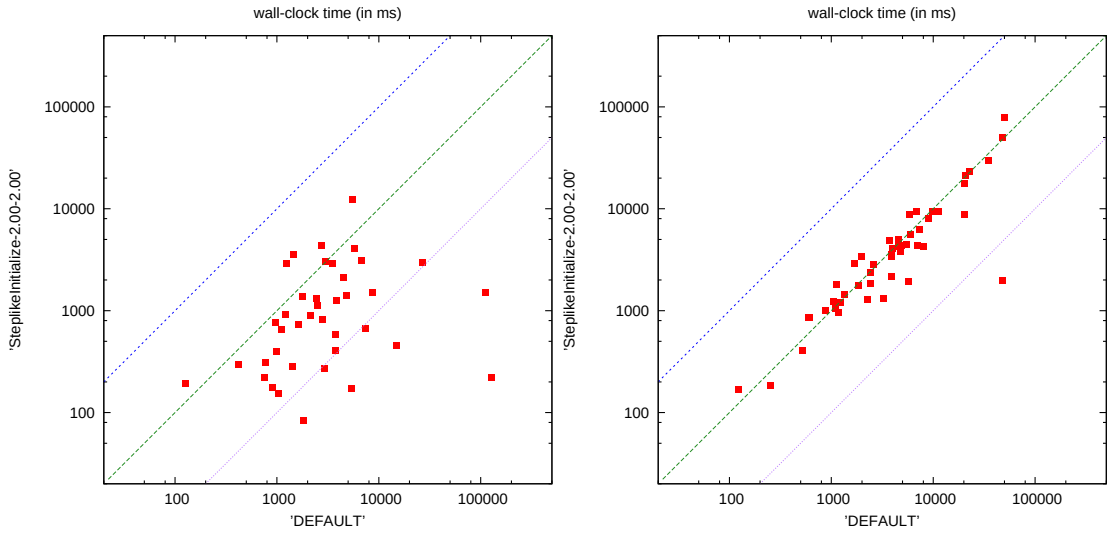

Figure 4.7: Step-like Activity Initialization with Overhead (SAT / UNSAT)

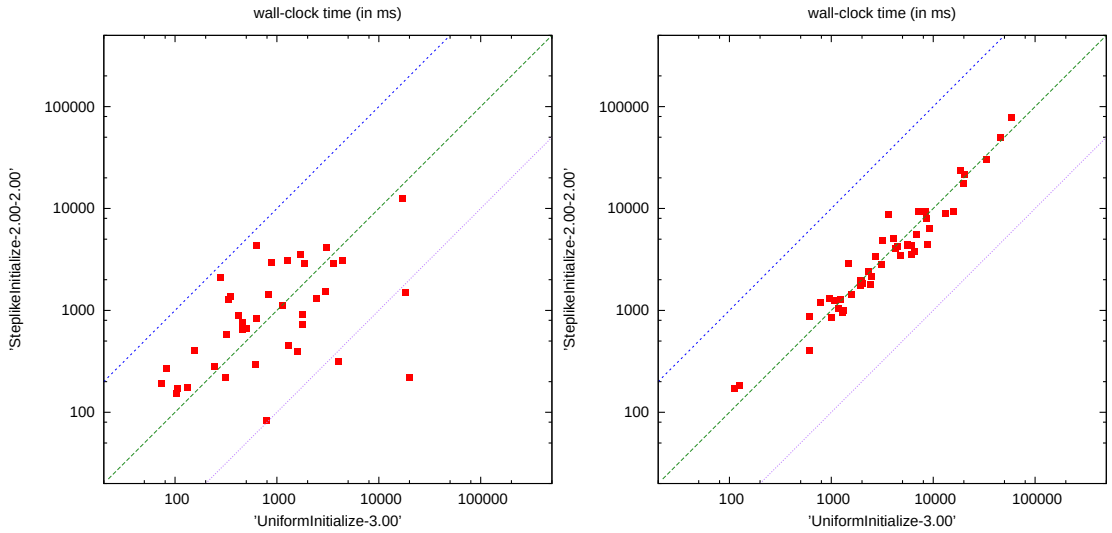

Figure 4.8: Uniform vs. Step-like Activity Initialization with Overhead (SAT / UNSAT)

4.5 Incrementing Bounds

Since not only the runtime but also the maximum feasible bound on the universe is important in bounded model checking, we ran another series of benchmarks in which we subsequently increased the overall bound of our Alloy models. Note that the semantics of a few Alloy models might thus have changed, as some of them depend on a more fine grained bound initialization (In Alloy-models universe bounds can be distributed signature-wise, where a signature basically is a relation, but all signatures are disjoint).

Tables 4.3 and 4.4 list the maximum bound solved within a timeout interval of one minute for each benchmark model. Results are separated by satisfiable (table 4.3) and unsatisfiable (table 4.4) problems. Many *satisfiable* problems included in the original benchmarks had to be dropped since the maximum bound for the default case is already high enough for the solver to run out of memory before finding a bound that takes longer than one minute to solve.

Column *default* lists the maximum bounds for the default method. Column *restricted* lists the maximum bounds for uniform activity overriding, whereas *uniform* lists the maximum bounds for uniform activity initialization of all primary variables. Column *step-like* lists the maximum bounds for step-like activity initialization of all primary variables. The smallest bound reached is *always* colored gray, so there might be rows with all bounds printed gray when no method could reach a higher bound. A bound is printed **bold** when *no other* method could reach a bound that high. The sign $^{\geq}$ indicates that the experiment had to be stopped as the system ran out of memory.

For the satisfiable problems (table 4.3) input restricted branching is a clear winner.

problem		default	restricted	uniform	step-like
opt_spantree SuccessfulRun	sat	13	**16**	14	14
opt_spantree TraceWithoutLoop	sat	10	**12**	11	11
ringlead NeverFindLeader	sat	15	15	14	15
stable_mutex_ring TraceShorterTMSL	sat	13	15	14	**32**$^{\geq}$
stable_orient_ring Closure	sat	18	19	19	17
stable_orient_ring SomeState	sat	20	**26**	20	24
stable_ringlead CMustConverge	sat	13	**22**	14	13
stable_ringlead CTraceWithoutLoop	sat	15	**23**	15	14
stable_ringlead ConvergingRun	sat	8	8	8	8
stable_ringlead DTraceWithoutLoop	sat	9	**13**	9	9
chord FindSuccessorWorks	sat	17	19$^{\geq}$	18$^{\geq}$	17
chord SameFP	sat	10	14$^{\geq}$	16$^{\geq}$	13
chord ShowMe2	sat	15	18$^{\geq}$	18$^{\geq}$	19$^{\geq}$
abstractFilesystem run1	sat	41	58$^{\geq}$	58$^{\geq}$	58$^{\geq}$
blockManager gcTest	sat	16	14	**20**	19

Table 4.3: Maximum bounds solved before reaching the timeout of 60 seconds, SAT

In 6 out of the 15 benchmarks it could reach the maximum bounds of all devised methods and in 4 other cases the maximum bound within one minute could not be determined due to memory restrictions. Uniform activity initialization had the best bound in only 1 case and in the same four other cases the maximum bound within one minute could not be determined due to memory restrictions. Also step-like activity initialization could reach a maximum bound in 1 case and additionally ran out of memory afterwards, so that this maximum bound could also be higher. There are cases as well where the best bound is shared among several methods and the cases where it is unclear which one is best. Notably for the satisfiable problems the default method's bound was never best compared to all other methods.

For the unsatisfiable problems (table 4.4) in 2 out of a total of 46 test cases the default method still reaches the best bound compared to all other methods. Also input restricted branching solves the best bound 2 times. Uniform activity initialization reaches the best bound in 3 cases. In 5 cases step-like activity initialization has the best bound for which in 3 cases the experiment had to be stopped since the bounds were getting so high, that the system ran out of memory.

problem		default	restricted	uniform	step-like
dijkstra DijkstraPreventsDeadlocks	unsat	24	21	23	24
dijkstra ShowDijkstra	unsat	10	11	11	10
opt_spantree BadLivenessTrace	unsat	7	7	7	7
opt_spantree Closure	unsat	**11**	7	9	10
peterson NotStuck	unsat	**11**	8	10	10
peterson Safety	unsat	30	30	29	**39**$^{\geq}$
peterson ThreeRun	unsat	17	17	17	17
peterson TwoRun	unsat	16	17	16	16
ringlead LeaderHighest	unsat	9	9	11	11
ringlead Liveness	unsat	9	9	9	9
ringlead OneLeader	unsat	9	9	9	9
ringlead SomeLeader	unsat	6	5	6	6
stable_mutex_ring Closure	unsat	6	**7**	6	6
stable_mutex_ring TwoPrivileged	unsat	18	**22**	18	18
stable_ringlead CBadLivenessTrace	unsat	9	9	9	9
stable_ringlead DBadLivenessTrace	unsat	6	6	6	6
chord InjectiveIds	unsat	14	14	14	14
chord Same1	unsat	9	8	9	9
chord Same2	unsat	10	9	10	9
chord SameCPF	unsat	5	5	5	5
chord SameCPF1	unsat	6	6	6	6
chord SameCPF2	unsat	7	6	7	7
chord SameFC	unsat	5	4	5	5
chord SameFP1	unsat	6	5	7	7
chord SameFP2	unsat	7	5	7	7
chord2 FindSuccessorWorks	unsat	5	5	5	5
chordbug FindSuccessorWorks	unsat	5	5	5	5
firewire AtMostOneElected	unsat	12	13	12	**40**$^{\geq}$
firewire NoOverflow	unsat	11	13	12	**34**$^{\geq}$
iolus OutsiderCantRead	unsat	5	5	6	5
sync SyncSpecNotUnique	unsat	7	5	7	7
abstractFilesystem WriteIdempotent	unsat	9	2	**10**	8
abstractFilesystem WriteLocal	unsat	8	2	8	**12**
flash ProgramIdempotent	unsat	10	3	10	9
flash ProgramLocal	unsat	16	5	17	17
flash eraseSome	unsat	23	24	22	**26**
flash readSome	unsat	25	25	**27**	24
flash writeSome	unsat	20	21	20	21
naming WriteRemoveConsistent	unsat	7	4	7	7
namingObject0 RenameSamePath	unsat	5	5	5	5
javatypes TypeSoundness	unsat	4	3	4	3
lists reflexive	unsat	14	13	**15**	14
lists symmetric	unsat	9	9	9	9
marksweepgc Completeness	unsat	7	5	7	7
marksweepgc Soundness1	unsat	12	7	12	10
marksweepgc Soundness2	unsat	8	6	8	8

Table 4.4: Maximum bounds solved before reaching the timeout of 60 seconds, UNSAT

Chapter 5

Summary and Outlook

In this work two methods have been devised to influence MiniSAT's variable ordering (section 3.1). In the first one we overrode and in the other one we initialized MiniSAT's native activity score. The approaches have been restraint to ways of directing MiniSAT to Kodkod's primary variables (subsection 3.2.1). As the uniform overriding approach plays out its strength mainly on satisfiable problems, strong runtime deteriorations to the point of timeouts are encountered on unsatisfiable problems (subsection 4.3.1). With the uniform initialization approach we could settle these runtime deteriorations and still keep most of the better runtimes on satisfiable problems (subsection 4.3.2).

By structural analysis of Kodkod's Abstract Syntax Tree (AST) and distribution of a "constrainedness" score (subsection 3.2.2), we were able to induce an inner ordering of the primary variables. Utilization of this inner ordering in our step-like initialization approach even lead to slight runtime improvements on unsatisfiable problems and outperformed the uniform initialization approach on unsatisfiable instances.

The results generally confirmed the importance of auxiliary variables for unsatisfiable problems. Therefore future research might include those and their importance to efficient branching heuristics as well. It is remarkable that, starting with highly constraint primary variables, the bounds of some unsatisfiable problems could be increased significantly, although prioritizing primary variables in general seemed not to be such a good idea for unsatisfiable problems. This insight can be used in future approaches prioritizing non-primary variables.

As an intermediate step in the translation to CNF (Conjunctive Normal Form) Kodkod creates a problem representation as a CBC (Compact Boolean Circuit). This representation is similar to Binary Decision Diagrams (BDD) but optimized for the specific needs of Kodkod. As this work was purely based on analysis of Kodkod's AST, ap-

proaches dealing with auxiliary variables could start analysis at the lower representation level of Kodkod's CBC. Utilization of a "constrainedness" score on all variables (*including* auxiliary variables) might boost the performance of unsatisfiable problems as well.

Not only the variable order but also the chosen polarity at each branching is crucial to the performance of SAT solvers. In future research one might thus also try to extract criteria for a suggested polarity for the first assignment of a variable. The polarity of a formula's sensitivity (subsection 3.2.3) might be taken into account to influence branching order.

Cabodi et al. [5] combined BDD-based search and SAT solving by feeding the solver with additional constraints gathered in preliminary BDD traversals, thereby gaining faster runtimes of the SAT solver. This also indicates that extraction of information at the BDD level — or CBC level as it comes to Kodkod — is a fertile ground for future performance optimization of SAT solvers in Kodkod.

We devised a measure that quantifies the sensitivity of boolean formulas to singular assignments (subsection 3.2.3). Furthermore we presented an algorithm that under the *independent variables assumption* can efficiently calculate high sensitivities (subsection 3.2.4). Studying its coherence with existing optimality criteria might give further insight into the development of fast branching heuristics for satisfiability problems or efficient algorithms for building minimal Ordered Binary Decision Diagrams (OBDD).

Bibliography

[1] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[2] P. Beame and T. Pitassi. Propositional proof complexity: Past, present, and future. In *Current Trends in Theoretical Computer Science*, pages 42–70. 2001.

[3] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[4] J. C. Blanchette and T. Nipkow. Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 131–146, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] G. Cabodi, S. Nocco, and S. Quer. Improving sat-based bounded model checking by means of bdd-based approximate traversals. *J. UCS*, 10(12):1696–1730, 2004.

[6] S. A. Cook, Robert, and A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[8] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with sat. In *ISSTA*, pages 109–120, 2006.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518, 2003.

[10] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 25–36, New York, NY, USA, 2010. ACM.

[11] M. K. Ganai. Propelling sat and sat-based bmc using careset. In *FMCAD*, pages 231–238, 2010.

[12] E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables in propositional satisfiability. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, JELIA '02, pages 296–307, London, UK, UK, 2002. Springer-Verlag.

[13] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *AAAI/IAAI*, pages 948–953, 1998.

[14] C. P. Gomes, A. Sabharwal, and B. Selman. *Model Counting*, chapter 20, pages 633–654. Volume 185 of Biere et al. [3], February 2009.

[15] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[16] M. Järvisalo and T. Junttila. Limitations of restricted branching in clause learning. *Constraints*, 14:325–356, September 2009.

[17] M. Järvisalo, T. A. Junttila, and I. Niemelä. Unrestricted vs restricted cut in a tableau method for boolean circuits. *Ann. Math. Artif. Intell.*, 44(4):373–399, 2005.

[18] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[19] E. Kang. Official website. `http://people.csail.mit.edu/eskang/`.

[20] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In M. M. Veloso and S. Kambhampati, editors, *AAAI*, pages 1368–1373. AAAI Press / The MIT Press, 2005.

[21] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *ISTCS*, pages 128–133, 1993.

[22] J. P. Marques Silva and I. Lynce. Towards robust cnf encodings of cardinality constraints. In *Principles and Practice of Constraint Programming (CP'07)*, pages 483–497, 2007.

[23] J. P. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. Volume 185 of Biere et al. [3], February 2009.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *(DAC'01)*, 2001.

[25] A. J. Parkes. Clustering at the phase transition. In *In Proc. of the 14th Nat. Conf. on AI*, pages 340–345. AAAI Press / The MIT Press, 1997.

[26] E. Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.* PhD thesis, MIT, 2008.

[27] E. Torlak and G. Dennis. Kodkod for alloy users, 2006.

[28] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, pages 632–647, 2007.

[29] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178, 2003.