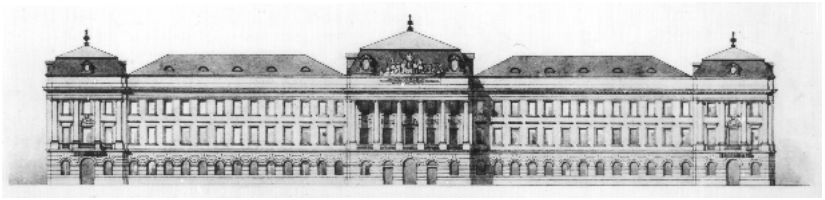
DBAI

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Alternation as a Programming Paradigm

DBAI-TR-2008-64

Wolfgang Dvořák **Georg Gottlob** **Reinhard Pichler**
Stefan Woltran

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2009

TU
TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2008-64, 2009

Alternation as a Programming Paradigm

Wolfgang Dvořák [1] **Georg Gottlob** [2] **Reinhard Pichler** [3]
Stefan Woltran [4]

Abstract. In this work, we show how the expressive power of an imperative programming language can be augmented by integrating alternation into it. In particular, we present Alter-Java – an extension of Java by language constructs to express alternation, i.e., a sequence of "there exists" and "for all" statements. Indeed, many practical problems have a very natural and succinct description in terms of alternation. Two-player games are obvious representatives of this species. But alternation is by no means limited to games. Also many problems with a procedural description of the solutions have a very compact declarative description via alternation, among them the evaluation of logic programs.
Alternation is commonly applied to complexity theoretical studies where it has proved very useful in various complexity classifications. In this work, we demonstrate that alternation is also a practically relevant programming paradigm which nicely fits into general purpose programming languages. In order to guarantee an efficient execution of such programs, we have introduced several optimizations. We also report on experiments with our implementation of Alter-Java. The results thus obtained illustrate that our alternation framework leads to competitive running times while the code to be written is significantly shorter than without this new language feature.

[1]Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria. E-mail: dvorak@dbai.tuwien.ac.at
[2]Computing Laboratory, Oxford University, Oxford OX1 3QD,UK. E-mail: georg.gottlob@comlab.ox.ac.uk
[3]Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria. E-mail: pichler@dbai.tuwien.ac.at
[4]Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria. E-mail: woltran@dbai.tuwien.ac.at

Copyright © 2009 by the authors

1 Introduction

The goal of this work is to augment the expressive power of an imperative programming language by integrating *alternation* into it. In particular, we present Alter-Java – an extension of Java by language constructs to express alternation, i.e., a sequence of "there exists" and "for all" statements. Indeed, many practical problems have a very natural and succinct description in terms of alternation.

For instance, the model-checking problem in temporal logic (and thus problems as program verification) can be succinctly described via alternating automata, albeit non-deterministic descriptions lead to an exponential blow-up in general (see, e.g., [23, 33]). More obvious representatives, which possess a very natural description in terms of alternation, are two-player games. For instance, Player 1 has a winning strategy if there exists a move of Player 1, such that for all moves of Player 2, there exist a move of Player 1, such that ... eventually a winning position for Player 1 is reached. Also many problems with a procedural description of the solutions have a very compact declarative description via alternation with logic programming being the most prominent one [32]. For instance, in propositional logic programming, a fact A is derivable from a program P if there exists a fact or a rule in P (whose head atom is A), such that for all atoms in the body of the rule, there exists a fact or a rule in P, such that ... eventually no further rule is needed.

Alternation is commonly applied to complexity theoretical studies where it has proved very useful in various complexity classifications. In this work, we demonstrate that alternation is also a practically relevant programming paradigm which nicely fits into general purpose programming languages. Integrating alternation into an imperative programming language has several advantages.

- Many problems admit a very natural formulation in terms of alternation. A program using alternation as a language feature is thus very close to the problem specification. On the one hand, this makes the programming task much simpler. On the other hand, the readability is improved and also the verification of such programs is greatly simplified.

- As will become clear when we present Alter-Java, the language extension required in order to express alternation only adds a very small number of new constructs. Hence, the resulting programming language is intuitive and easy to learn.

- Integrating alternation into an imperative language allows the programmer to specify the action required in each state by means of the imperative language while the overall structure of the algorithm is specified in terms of the declarative concept of "states".

- The programmer has to specify the states (indicating for each state if it is of type "exists" or "for all", the action carried out in each state, and the successor state) while the framework executing alternating algorithms takes care of the actual computation. Generally applicable optimizations can thus be incorporated into the framework and are not the programmer's concern.

An important challenge faced by alternating algorithms is that, in general, they give rise to exponentially large computation trees. Hence, without further measures, the naive execution of an

alternating algorithm leads to exponential time computation even for problems that are, in principle, solvable in polynomial time. Instead, tabling [8, 31] should be used in order to avoid the repetition of computations that have already been done. An even more severe problem encountered by alternating machines are computation paths that are excessively long or do not halt at all. In games, this means that there is a cycle in the strategy. A straightforward way to cope with this kind of problem is to introduce a counter. However, such a technique has several drawbacks. First, the programmer is requested to provide an upper bound for the counter and, second, such an upper bound is never tight and, thus, usually leads to very inefficient computation. We therefore shift the detection of excessively long or even non-halting paths in alternating programs from the programming level to the framework for executing such algorithms. So our framework handles cycles in programs and the programmer is completely relieved of this concern.

In the first place, the alternating algorithms presented in this paper are designed for decision problems. Consequently, the algorithms only give "yes" and "no" answers telling us *if a solution exists.* In case of games, the solution is a winning strategy, in case of Horn logic programming, the solution is a proof tree, etc. Hence, without further measures, these algorithms do not tell us *what a solution may look like.* However, we shall show that with little extra effort during the execution of the alternating algorithms, we can compute a solution (in case of a "yes" answer) in a simple post-processing step.

Results. The main contributions of this paper are as follows.

- We give examples of programs and problems that demonstrate the existence of interesting and *feasible* alternating problems besides the intrinsically hard strategy games (like chess or Go), which are typically PSPACE-complete or even harder.
- We present Alter-Java – an extension of the Java language for expressing alternation. It turns out that only a very small number of additional language constructs is needed, thus making the resulting programming language easy to learn.
- We have implemented Alter-Java. A compiler (transforming an extended Java program into standard Java) and a framework for executing alternating algorithms are publicly available, see `http://www.dbai.tuwien.ac.at/proj/alternation/`.
- We have incorporated several optimizations into our Alter-Java framework. We also report on experiments with our implementation of Alter-Java. The results thus obtained illustrate that our alternation framework leads to competitive running times while the code to be written is significantly shorter than without this new language feature.

Structure. The remainder of the paper is organized as follows. In the next section, we discuss related work. Then, in Section 3, we recall some basic definitions and properties of alternation. Several typical problems together with succinct alternating algorithms are described in Section 4. In Section 5, we show for the various problems how a possible solution can be computed in case of a "yes" answer. Alter-Java in conjunction with some code examples is presented in Section

6. We report on the implementation and optimization of our Alter-Java framework and on first experimental results with this framework in Sections 7 and 8. A conclusion is given in Section 9.

2 Related Work

The idea to use alternation as a programming paradigm dates back to the work by Harel [19] who suggested a programming language which divides conventional programming constructs into classes of "and-" and "or-subgoals". Thus, this language can be seen as a straightforward abstraction of alternating Turing machines (which we formally introduce below in Section 3). Recent research, in turn, has been mostly devoted to the integration of different programming paradigms in order to combine their advantages; for a good overview see, e.g. [16, 17].

The integration of language features from declarative programming (in particular, from logic programming) into imperative programming languages has a long tradition. In [9], an overview of *non-determinism* is given, including a choice function (to realize guesses) and backtracking to be integrated into imperative programming languages. In [27], language extensions of Pascal with concepts from logic programming are presented. In particular, non-determinism in the form of a "split" construct (to deal with branches which may, in principle, be processed fully in parallel) has thus been incorporated into Pascal. These ideas have been significantly extended in the ALMA-project, where various declarative features (including non-determinism with "some" and "either – orelse" constructs as well as a controlled form of iteration over the backtracking via the "forall" construct) have been built into Modula-2, see [2, 3, 4].

The work that evolved from the ALMA-project is closest to ours. Our approach differs from the previous works in that we aim at a complete, tailor-made solution for problems that are naturally expressed in terms of alternation. As a consequence, our language extension emphasizes the concept of a "state" (which can be either of type "exists" or of type "for all"). The programmer's task is to define the possible states, i.e., specify the type of each state, the actions to be carried out (by using the full power of the underlying imperative language) and the possible successor states. Moreover, our goal is to provide a framework which actually guarantees competitive computing time of the alternating programs.

Our target language is Java. Other attempts to build declarative language features into Java have been, for instance, [1] and [34]. The goal of those works was to integrate declarative constraint handling mechanisms into Java.

Conversely, also imperative programming features have been integrated into declarative languages. In particular, imperative extensions of logic programming as provided by Prolog interpreters like the SICStus Prolog C interface, or the Foreign Language Interface of SWI-Prolog have been accomplished (see also [26]). Recently, also answer-set programming has been extended by such features (realized via so-called external atoms), most notably the dlvhex system [11] which has been successfully applied to problems related to the Semantic Web [12].

3 Preliminaries

In this section, we have a look at the foundations of alternation by recalling alternating Turing machines (ATMs) and some basic results on complexity classes defined via ATMs [5].

Deterministic and non-deterministic TMs. Let $k \geq 1$ be an arbitrary integer. A *deterministic k-string Turing machine (DTM)* is a quadruple $M = (K, \Sigma, \delta, s)$ where K is a finite set of states, Σ is the alphabet of M (i.e., a finite set of symbols including the blank symbol $\sqcup$ and the start symbol $\triangleright$), $s \in K$ is the initial state, and δ: $K \times \Sigma^k \to (K \cup \{\text{“yes”}, \text{“no”}\}) \times (\Sigma \times \{\rightarrow, \leftarrow, -\})^k$ is the *transition function*. A *non-deterministic k-string Turing machine (NTM)* is a quadruple $M = (K, \Sigma, \Delta, s)$ with K, Σ, and s as before, and $\Delta \subseteq K \times \Sigma^k \times (K \cup \{\text{“yes”}, \text{“no”}\}) \times (\Sigma \times \{\rightarrow, \leftarrow, -\})^k$ is the *transition relation*. DTMs can be considered as a special case of NTMs where, for every $(q, \bar{a}) \in K \times \Sigma^k$, the relation Δ contains only one tuple $(q, \bar{a}, q', \bar{a}', \bar{d})$.

Configurations and computations. The computation of a Turing machine M on some input x can be described as a sequence of *configurations*, indicating the state of M, the tape contents of the k tapes of M, and the cursor position on each tape. The initial configuration of M on input x is defined as follows: M is in the initial state s, the cursor on each tape is on the first position, the contents of tape 1 is $\triangleright x \sqcup \sqcup \ldots$ (i.e., start symbol $\triangleright$ plus input x plus blanks), and the contents of the remaining tapes is $\triangleright \sqcup \sqcup \ldots$. Now suppose that M is in some configuration c, s.t. M is in state q and the cursors on the k tapes are currently reading the symbols $\bar{a} = (a_1, \ldots, a_k)$. Then the computation of M is allowed to continue in configuration c' if Δ contains a tuple $(q, \bar{a}, q', \bar{a}', \bar{d})$ and c' is obtained from c as follows: (1) The state in c' is q', (2) the tape cells pointed at by the k cursors in c are overwritten by the symbols $\bar{a}' = (a'_1, \ldots, a'_k)$, (3) all other tape cells are left unchanged, and (4) the k cursors of M are moved according to $\bar{d} = (d_1, \ldots, d_k)$, i.e., if d_i is given by $\rightarrow$ (resp. $\leftarrow$ or $-$) then the i-th cursor is moved one cell to the right (resp. is moved one cell to the left or stays where it is). M halts if it reaches a configuration with state “yes” or “no”.

If M has a legal transition from configuration c to c', then we write $c \overset{M}{\to} c'$. All possible computations of an NTM M on input x can be represented in a *computation tree*, whose vertices are the configurations of M on input x: the root node is the initial configuration and a node c' is the child of a node c if $c \overset{M}{\to} c'$. For a DTM M, this tree degenerates to a single path, i.e., every node has at most one child. An NTM M accepts an input x if there exists a branch in the computation tree, s.t. the leaf node is a configuration with state “yes”. A language (or, equivalently, a decision problem) is *decided by a machine* M if $L = \{x \in \Sigma^* \mid M \text{ accepts input } x\}$.

Alternating Turing machines (ATMs). An *alternating k-string Turing machine (ATM)* is a quintuple $M = (K, \Sigma, \Delta, s, g)$ where K, Σ, Δ, and s are as in case of NTMs and g: $K \to \{\wedge, \vee, \textit{accept}, \textit{reject}\}$ assigns labels to the states. Only the “yes” state is labelled with *accept* and only the “no” state is labelled with *reject*. We call a state “universal” (resp. “existential”) if it is labelled with $\wedge$ (resp. $\vee$). NTMs can be considered as ATMs where all states other than “yes” and “no” are “existential”. Similarly, if some problem $L \subseteq \Sigma^*$ is decided by an NTM, then its co-problem co-$L = \Sigma^* \setminus L$ is decided by an ATM where all states other than “yes” and “no” are “universal”. In Figure 1, a typical computation tree of an arbitrary ATM is juxtaposed with the special case of a problem L decidable by an NTM. For the co-problem of L, the existential states

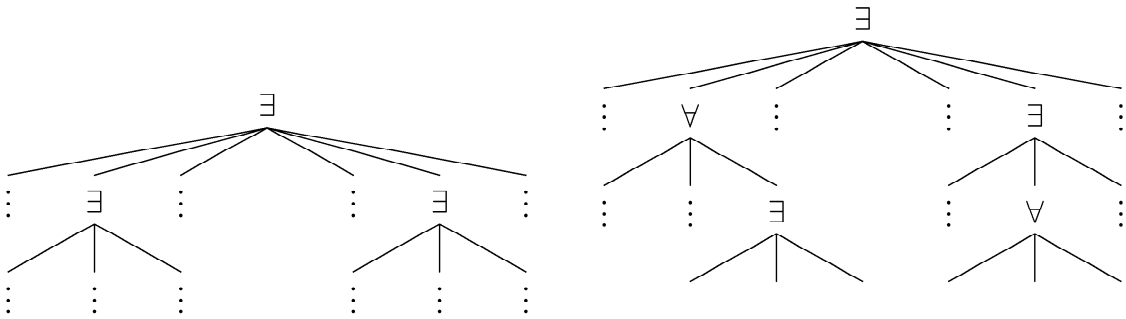


Figure 1: Computation tree of NTMs vs. ATMs.

simply have to be replaced by universal ones.

The intuition of the labelling function g is captured by defining if a configuration *leads to acceptance* or not. This definition works bottom-up in the computation tree of an ATM M on input x. (1) A configuration with state "yes" leads to acceptance; a configuration with state "no" does not lead to acceptance. (2) A configuration with a universal state leads to acceptance if and only if all successor configurations (i.e., all child nodes in the computation tree) lead to acceptance. (3) A configuration with an existential state leads to acceptance if at least one successor leads to acceptance.

Clearly, in order to recognize if a configuration c leads to acceptance, it is not always necessary to inspect all successor configurations of c. Indeed, in case of an existential configuration, it suffices to detect one successor configuration which leads to acceptance; we do not care about the other successors. Likewise, if a universal configuration has a successor that does not lead to acceptance, then the other successors are irrelevant. Therefore, we can decide the acceptance of quantified configurations with non-halting successors as long we have a witness for the "leading-to-acceptance" value. This observation leads to the definition of a labelling function $\ell\colon C_{(M,x)} \to \{1, 0, \bot\}$, where $C_{(M,x)}$ is the set of all configurations in the computation tree of M on input x and $\ell(c)$ indicates if a configuration c leads to acceptance ($\ell(c) = 1$) does not lead to acceptance ($\ell(c) = 0$), or we don't care ($\ell(c) = \bot$). The labelling $\ell(c)$ of any configuration c in the computation tree has to be determined by the labellings $\ell(c')$ of all successor configurations according to the standard 3-valued semantics of logical "and" and "or" operators, e.g., $(0 \wedge \bot = 0)$ and $(1 \wedge \bot = \bot)$, while $(0 \vee \bot = \bot)$ and $(1 \wedge \bot = 1)$. An ATM M accepts an input x if there exists a labelling ℓ of the computation tree of M on input x, s.t. the $\ell(c_0) = 1$ for the initial configuration c_0. A language (or, equivalently, a decision problem) is *decided by an ATM* M if $L = \{x \in \Sigma^* \mid M \text{ accepts input } x\}$.

Complexity classes. Suppose that some language L is decided by a k-string DTM or NTM M. We say that M *decides* L *in time* $f(n)$ if, for every input x, the length of every computation path is restricted by $f(|x|)$. Similarly, we say that M *decides* L *in space* $f(n)$ if for every input x and on every computation path, the total number of cells ever visited by any of the cursors on tape $2, \ldots, k$ is restricted by $f(|x|)$. Note that, in order to allow sub-linear space (in particular, logarithmic space), the first tape of a k-string TM is considered as an input tape, i.e., it may be read

in any direction but its content may never be altered.

By P and EXPTIME we denote the class of problems decidable by a DTM in polynomial resp. exponential time. By L, PSPACE, EXPSPACE, we denote the class of problems decidable by a DTM in logarithmic resp. polynomial resp. exponential space. By NP, NEXPTIME, NL, we denote the class of problems decidable by an NTM in polynomial time, exponential time, or logarithmic space, respectively.

Now suppose that some language L is decided by an ATM M. We say that M *decides* L *in time* $f(n)$ if, for every input x, there exists a labelling function $\ell\colon C_{(M,x)} \to \{1, 0, \bot\}$, s.t. the length of every computation path whose configurations have a label different from $\bot$ is restricted by $f(|x|)$. Similarly, we say that M *decides* L *in space* $f(n)$ if for every input x and on every computation path whose configurations have a label different from $\bot$, the total number of cells ever visited by any of the cursors on tape $2, \ldots, k$ is restricted by $f(|x|)$. Note that alternating complexity classes (like deterministic complexity classes) are closed under complement, i.e., if a language L is decided by some ATM M in time resp. space $f(n)$, then the complementary language co-L is decided by the ATM M', which is obtained from M by swapping "yes" and "no" states and by swapping "universal" and "existential" states.

We thus get the alternating complexity classes ALOGSPACE, AP, APSPACE, and AEXPTIME as the classes of problems decidable by ATMs in alternating logarithmic space, polynomial time, polynomial space, and exponential time, respectively. The following relationships between alternating and "ordinary" complexity classes are shown in [5]:

$$\begin{aligned} \mathsf{ALOGSPACE} &= \mathsf{P} \\ \mathsf{AP} &= \mathsf{PSPACE} \\ \mathsf{APSPACE} &= \mathsf{EXPTIME} \\ \mathsf{AEXPTIME} &= \mathsf{EXPSPACE} \end{aligned}$$

4 Problem Descriptions Using Alternation

P is usually considered as the class of efficiently computable problems, whereas the problems hard for NP or any class above (like PSPACE, EXPTIME, EXPSPACE, etc.) are considered as intractable. By the relationship between alternating and "ordinary" complexity classes (see Section 3), we can thus think of ALOGSPACE as the class of efficiently computable alternating problems. Note that in logarithmic space, we are allowed to store pointers or counters but not larger parts of intermediate results.

The class P contains many well-known problems, and a good deal of these problems are in fact P-complete. Hence, in principle, all of these problems are also solvable in ALOGSPACE. Of course, alternating algorithms are not meant to replace deterministic ones in every situation, i.e., alternation is not the best solution for all problems in P. However, there is a surprisingly large portion of problems in the class P which have a natural and intuitive specification in terms of alternation. In this section, we present a few examples. All of these problems are taken from the collections of P-complete problems in [18] and [22].

Alternating Algorithms for Horn Problems.

Definite Horn Minimal Model.
Atom p is in the minimal model if there EXISTS a clause $C \in \varphi$ containing p as literal such that FORALL negative literals $\neg q_i \in C$ it holds that q_i is in the minimal model. Checking that q_i is in the minimal model is done in the same way.

co-Definite Horn Minimal Model.
p is not in the minimal model if FORALL clauses $C \in \varphi$ containing p as literal there EXISTS a negative literal $\neg q_i \in C$ such that q_i is not in the minimal model. Checking that q_i is not in the minimal model is done in the same way.

Horn Satisfiability (HORNSAT).
φ is satisfiable if FORALL goals $C \in \varphi$ there EXISTS a negative literal $\neg q_i \in C$ such that q_i is not in the minimal model. For checking that q_i is not in the minimal model we use the alternating approach for the co-problem of Definite Horn Minimal Model.

Figure 2: Alternating algorithms for Horn problems.

4.1 Horn problems

The first group of problems we present here stems from logic programming, which has been characterized via alternation by Shapiro [32] back in 1984. In that paper, programs with variables are considered. For the matter of presentation, we restrict ourselves here to (propositional) logic programming, i.e. problems associated to propositional *Horn formulas*. Recall that a propositional formula φ in conjunctive normal form is Horn (resp. definite Horn) if each clause in φ contains at most one (resp. exactly one) unnegated atom. A clause consisting of positive and negative literals (resp. only a positive literal resp. only negative literals) is called a rule (resp. fact resp. goal). Now consider the following basic problem:

Definite Horn Minimal Model
Instance: A definite Horn formula φ and an atom p.
Question: Is p in the minimal model of φ?

An atom p is in the minimal model if p is a fact in φ or p is the positive literal in a clause and all other atoms in the same clause are in the minimal model. The case that p is a fact is just a special case of the second one where p is the only literal in the clause.

An alternating algorithm for this problem is given in Figure 2. A computation path of this algorithm ends when arriving at a clause C that is a fact, i.e., C consists of a single (positive) literal. Such a C has no negative literals and therefore no successors. As usual, universal quantifiers over the empty set are interpreted as accept and existential quantifiers over the empty set as reject. A problem in this algorithm is that we may get some cycles. For example, if φ includes the rules $q \rightarrow p, p \rightarrow q$, then asking for p leads to an infinite computation path alternating these two rules. We shall come back to this point in Section 7, when we present some optimizations built into our

alternation framework.

As mentioned in Section 3, alternating complexity classes are closed under complement. Hence, in Figure 2, we also give an alternating algorithm for the co-problem of Definite Horn Minimal Model. This algorithm can then be used to solve the satisfiability problem for Horn clauses, which is a well-known P-complete problem [10].

Horn Satisfiability (HORNSAT)
Instance: A Horn formula φ.
Question: Is φ satisfiable?

A straightforward deterministic algorithm would compute the minimal model for the definite Horn part and then check all goals with this model. Our alternating algorithm in Figure 2 uses this idea, but in reversed order: In the first step, we branch over the goals and then we check whether they are consistent with the rules and facts of φ.

4.2 Games

A very common class of natural alternating problems are two-player games, more precisely the problem of finding a winning strategy in such a game. For most strategy games, this problem is much harder than P. For instance, Go (with Japanese rules) is EXPTIME complete [13, 29] and even the simplified version in [24], which forbids exponentially long games, is PSPACE-complete. In fact, even the simple game Tic-Tac-Toe (when played on an $m \times n$ board and players need k cells in a row) is PSPACE-complete [28]. Below, we present two tractable games on graphs in conjunction with alternating algorithms. Note however that our alternating programming framework is, in principle, also applicable to hard games – of course, with the expected exponential time and space consumption. We start our exposition by describing the general pattern of two-player games [20, 18].

Two-Player Game (GAME)
Instance: A two-player game G.
This is a tuple $G = (P_1, P_2, W_1, s, M)$ with $P_1 \cap P_2 = \emptyset$, $W_1 \subseteq P_1 \cup P_2$, $s \in P_1$ and $M \subseteq P_1 \times P_2 \cup P_2 \times P_1$.

- P_1 is the set of game position where it is Player 1's turn, and P_2 is defined analogously.
- W_1 the set of immediate winning positions for player 1.
- s the starting position, i.e. player 1 starts the game.
- M is the set of allowed moves. So if a player is in position p and $(p, q) \in M$, then the player may move to position q. The definition of M ensures that the turns alternate between the two players.

We define a winning position with a kind of recursion. A position p is called a winning position if and only if one of the following conditions holds:

- $p \in W_1$ (i.e., p is an immediate winning position).

Alternating Algorithms for Games.

Two-Player Game.
Player 1 has a winning strategy in the Game G if there EXISTS a move for player 1 such that FORALL possible moves of player 2, player 1 has a winning strategy.

Cat and Mouse.
The mouse has a winning strategy if there EXISTS a move for the mouse such that FORALL possible moves of the cat, the mouse has a winning strategy.

Acyclic Geography Game.
Player 1 has a winning strategy starting in s if there EXISTS an edge (s, v) such that FORALL edges (v, v') there is winning strategy for player 1 starting in v'.

Figure 3: Alternating algorithms for games.

- $p \in P_1$ and there exists a move $(p, w) \in M$ such that w is a winning position.
- $p \in P_2$ and for every move $(p, w) \in M$, w is a winning position

Question: Is s a winning position for Player 1?

The definition of GAME captures almost all two-player games, but for the most games the encoding as GAME would lead to an exponential or higher blow up of the input. Note that also n-player games can be easily transformed into a two-player game. We just encode each possible combination of moves from player $2, 3, \ldots, n$ as move for player 2 in the two-player game. A straightforward alternating algorithm for the GAME problem is displayed in Figure 3.

We now turn our attention to two tractable games. In fact, the tractability of these games is easiest to prove by an argument using alternation: We shall see below that these problems have alternating algorithms that need to store only logarithmically big information on each position (i.e., pointers to vertices in a graph). Hence, these problems are easily seen to be in ALOGSPACE and, thus, in P.

Cat and Mouse
Instance: A directed graph (V, E). Vertices $c, m, g \in V$
The game is played as follows: The cat starts on vertex c, the mouse starts on vertex m, and vertex g is the goal (the mouse hole). The mouse has the first move and then moves alternate between cat and mouse. In each move, the player (cat or mouse) has the choice to follow an edge in the graph or stay at the current vertex. The mouse loses if it gets caught by the cat, which means that mouse and cat occupy the same vertex. The mouse wins if it reaches the mouse hole without being caught.
Question: Does the mouse have a winning strategy?

Acyclic Geography Game (AGG)
Instance: An acyclic directed graph $G = (V, E)$ and a vertex s. The game is played as follows: We start with a token on the vertex s. Player 1 has the first move and then the players alternate

Alternating Algorithms for Boolean Circuit Problems.

Monotone Circuit Value Problem.
An AND gate is true if FORALL predecessor gates it holds that they are true, it is false otherwise. An OR gate is true if there EXISTS a predecessor gate which is true, it is false otherwise. An input gate has the truth value of its corresponding input. To compute the value of y, we just use these rules.

co-Monotone Circuit Value Problem.
An AND gate is true if there EXISTS a predecessor gate that is true, it is false otherwise. An OR gate is true if FORALL predecessor gates it holds that they are true, it is false otherwise. An input gate has the negated truth value of its corresponding input. To compute the negated value of y we just use these rules.

Circuit Value Problem.
To compute the truth value of y, we start using the rules from the Monotone Circuit Value Problem above to evaluate AND and OR gates. When evaluating a NOT gate, we skip to the rules from the co-Monotone Circuit Value Problem above to evaluate AND and OR gates. When we again encounter a NOT gate, then we switch back to the rules from the Monotone Circuit Value Problem above to evaluate AND and OR gates.

Figure 4: Alternating algorithms for Boolean circuit problems.

moving. In each move the active player can move the token along one edge. The first player with no possible move loses.
Question: Does Player 1 have a winning strategy?

Cat and Mouse was studied in [6], while the Acyclic Geography game is from [7]. Straightforward alternating algorithms for these two games are shown in Figure 3. The membership of these two games in ALOGSPACE and, hence, in P is immediate. Indeed, in the Cat and Mouse game, each position of the game is determined by the current vertex of the mouse and of the cat. Similarly, in the Acyclic Geography Game, it suffices to store the current vertex of the token. In order to store at most two vertices of a given graph, we do not require more than logarithmic space.

4.3 Circuits

Also the evaluation of Boolean circuits allows for intuitive solutions via alternation. We start with a restricted form, which is nonetheless P-complete [18].

Monotone Circuit Value Problem (MCVP)
Instance: A monotone Boolean circuit α (a circuit built only of AND and OR gates) with inputs $x_1, \ldots, x_n$ and output y
Question: Is output y true on the inputs $x_1, \ldots, x_n$?

In Figure 4, we give an alternating algorithm for the Monotone Circuit Value Problem and also for its co-problem, i.e., the problem of checking if some Boolean circuit *does not* evaluate to true. These two algorithms can then be combined to solve the Circuit Value Problem [21] for arbitrary Boolean circuits (including NOT-gates).

Circuit Value Problem (CVP)
Instance: A Boolean circuit α with inputs $x_1, \ldots, x_n$ and output y
Question: Is output y true on the inputs $x_1, \ldots, x_n$?

5 Computing a Solution

Strictly speaking, the alternating algorithms in the previous section (like alternating Turing machines, in general) only give "yes" and "no" answers telling us *if a solution exists.* In case of games, the solution is a winning strategy, in case of the Horn problems, the solution is a proof tree, etc. In the first place, these algorithms do not tell us *what a solution may look like.* However, as we execute the alternating algorithms, little extra effort suffices to store the required information which allows us to reconstruct the actual solutions in a post-processing step. Essentially, the information needed is a representation of the computation tree of the alternating algorithm. In fact, as we have seen in Section 3, not the entire computation tree is needed as a witness for the "yes" answer. Recall that we may use a labelling function $\ell\colon C_{(M,x)} \to \{1, 0, \bot\}$, where $C_{(M,x)}$ is the set of all configuration in the computation tree on input x and $\ell(c)$ indicates if a configuration c leads to acceptance ($\ell(c) = 1$), does not lead to acceptance ($\ell(c) = 0$), or we don't care ($\ell(c) = \bot$). Only the configurations with label different from $\bot$ are needed to derive a solution. Hence, if we store the configurations in the computation tree plus the edges between them, we can afterwards compute a solution in case the algorithms ends with a "yes" answer. In this section, we look how the (relevant part of the) computation tree can be encoded for the alternating algorithms from the previous section.

As far as the complexity is concerned, it should be noted that even tractable problems (i.e., in $\mathsf{P} = \mathsf{ALOGSPACE}$) have, in general, exponentially big computation trees. In Section 7 we shall see how tabling can be used to ensure efficient execution of our alternating algorithms. The tables used to store the already encountered configurations of the alternating algorithm can be extended to a succinct (i.e., polynomially big) representation of the possibly exponentially big solution.

5.1 Proof Trees in Horn Problems

When computing that p is in the minimal model of a definite Horn formula φ we may also be interested in proving this. We can use the computation tree as proof tree. To prove that p is in the minimal model we have to find a rule $q_1, \ldots, q_n \to p,\ n \geq 0$ and proofs for $q_1, \ldots, q_n$. There are two types of configurations in the computation tree. First the existentially quantified configurations with literals to be proven and then the universal configurations with rules to be satisfied.

To get a proof that p is in the minimal model, we start with the configuration representing p (the initial configuration) and then follow the (first) leading-to-acceptance successor configuration.

This successor configuration represents a rule of the form $q_1, \ldots, q_n \rightarrow p,\ n \geq 0$ and has the successor configurations representing $q_1, \ldots, q_n$ and all of them lead to acceptance. Assuming we have a proof that $q_1, \ldots, q_n$ are in the minimal model, we can construct the proof for p by simply adding $q_1, \ldots, q_n \rightarrow p$ to the proof. The proofs for the q_i's can be computed recursively from the computation tree by the same idea. This can be formally proven by induction on the tree size and the fact that every accepting computation path is finite.

Note that, in general, it is not possible to compute the minimal model from the computation tree. The reason for this is that there may be facts and rules in φ which the alternating algorithm has not used when checking that p is in the minimal model (i.e., these facts and rules are encoded in configurations labelled with $\perp$ in the computation tree).

5.2 Winning Strategy in Games

The computation tree of a two player game saves the game configurations (e.g., the position of cat and mouse in case of the Cat and Mouse game and the position of the token in case of AGG) and the connections between two configurations. The difference between two connected configurations is the move made by one player. Configurations of a winning strategy can be divided into two sets. The existentially quantified moves where player 1 has to make a move and the universally quantified moves where it is player 2's turn.

We can identify the first move of a winning strategy by choosing a leading-to-acceptance configuration A among the successors of the initial configuration. Now the opponent can choose among several moves which all lead to successors $E_1, \ldots, E_k$ of A. A was universally quantified and therefore every E_i is a leading-to-acceptance configuration. Therefore there is a leading-to-acceptance successor for E_i which gives us the next move for the winning strategy. We do this until there is an immediate winning position. For a formal proof once more we can use induction and the fact that an accepting computation path is finite.

5.3 Circuit Value

An alternating algorithm for evaluating Boolean circuits would save the current gate of the evaluation in the configuration. So here we can use the computation tree to get the truth values of some internal gates or find all gates that were evaluated to compute the truth value of the output gate. A gate is true if it is encoded in a leading-to-acceptance configuration (it is labeled with 1) and false if the configuration is labeled with 0. We cannot make a decision about gates not encoded in a configuration or encoded in a configuration labelled with $\perp$.

6 Alter-Java

We now present Alter-Java – an extension of the Java language that supports alternating programs. Due to space limitations, we only highlight this language here. The complete ANTLR grammar [25] of Alter-Java together with a compiler that transforms Alter-Java programs into

standard Java and a Java package for executing alternating programs is available at `http://www.dbai.tuwien.ac.at/proj/alternation`.

Variables. The main idea of alternating programming is to define states which compute successor states and combine the successors' results to a return value for the original state. Following the idea of alternating Turing machines, we have to distinguish between input variables (corresponding to the input tape of an ATM) and work variables (corresponding to the work tapes of an ATM). Recall that this distinction is crucial for problems in ALOGSPACE and, hence, in P. Input variables are variables that the alternating algorithm cannot modify. Work variables are variables that describe the configurations of an alternating program. For problems in ALOGSPACE, the work variables may only consume logarithmic space. Hence, in this case, the work variables are either of primitive data type (or possibly fixed-size arrays thereof) or variables of object types used as "cursors", i.e.: they may only be assigned to objects from the input (for reading access) rather than created with *new*.

Overall structure. An alternating program is structured as follows. At the beginning, there is the place for header information. The program starts with the *atm* keyword and the name of the program. Following the name there is the place to define the input variables for the alternating algorithm. Then there is a block with the definition of our work variables and the code for the states – the central part of an alternating program. A state definition begins with the *state* keyword followed by the name of the state and a code block. In this code block, the programmer can use most of the Java constructs. An important restriction is that the programmer is not allowed to define local variables (since variables have to be either input or work variables) except in for-loops. Moreover, there are some restrictions on object-oriented features, e.g., as has already been mentioned, the *new* command is not allowed.

States. In the code block of a state, some additional constructs are allowed in order to support alternation. First there are the commands *accept* and *reject* which immediately decide if a configuration leads to acceptance or not. The alternation in the algorithms comes from the *forall* and *exists* statement. The *forall* command says that the configuration has a set of successors (maybe the empty set) and it only leads to acceptance if all successors lead to acceptance. The *exists* statement has the dual semantics, i.e., the configuration leads to acceptance if one successor leads to acceptance. As the semantics of the *forall* and *exists* commands needs the successor configurations they are always followed by a block defining these successors. For a configuration we need a state and work variables. So a *forall* block resp. an *exists* block is basically a list of (state, work variables) pairs. Each such pair is the name of a state and a block defining the work variables. If the number of successors depends on the input, then for-loops may be used to compute the successors.

Execution. An alternating program works as follows: After initialization of the work variables, the computation starts in the first state. A state makes some computation using input and work variables and then either immediately accepts / rejects the input or computes a quantifier and a set of successor configurations. A configuration in an alternating program is defined analogously to alternating Turing machines, so a configuration is simply a state with an instance of the work variables. A configuration that does not immediately accept or reject has a set of successors and a quantifier. Such a configuration accepts the input if one successor accepts (in case of an existential

```
package at.ac.tuwien.dbai.alternation.examples;
import java.util.Map;
import java.util.Set;

atm DefiniteHorn(int goal,
                 Map<Integer, Set<Set<Integer>>> rules)
{
   int head=goal;
   Set<Integer> body;

   state head{
      exists{
         for(Set<Integer> rule: rules.get(head)){
            body{
               body=rule;
            }
         }
      }
   }

   state body {
      forall{
         for(Integer literal: body){
            head{
               head=literal;
            }
         }
      }
   }
}
```

Figure 5: Alter-Java program for Definite Horn Minimal Model.

quantifier) or if all successors accept (in case of a universal quantifier).

Examples. In the following subsections, we put Alter-Java to work by presenting code portions to implement some of the algorithms from Section 4.

6.1 Definite Horn Minimal Model

Recall the Definite Horn Minimal Model problem from Section 4.1. In Figure 5 we give an Alter-Java program for this problem. We have decided to encode the propositional variables as integer values but, of course, other encodings are conceivable. A problem instance is encoded as a pair consisting of **int** goal (the variable for which we test whether it is in the minimal model) and the object Map<Integer, Set<Set<Integer>>> rules, that is our set of Horn clauses represented by mapping each atom h to bodies of rules which have h as head. Bodies of rules themselves are sets of atoms. Thus we map each atom to a set of sets of atoms. In particular, a fact f is encoded as rule with f as head and the emtpy set as body — so, f is mapped to a set that contains the empty set. Recall here that existential quantification over the empty set yields rejection, while universal quantification over the the empty set yields acceptance.

The program has two states: the state head that tests if the currently processed variable head is in the minimal model and the state body that tests if the currently processed body of a rule is in the

```
package at.ac.tuwien.dbai.alternation.examples;

atm CatAndMouse(Graph<Integer> gameboard,
                Graph.Node<Integer> catStart,
                Graph.Node<Integer> mouseStart,
                Graph.Node<Integer> goal)
{
   Graph.Node<Integer> catPosition=catStart;
   Graph.Node<Integer> mousePosition=mouseStart;

   state mouseTurn {
      if (mousePosition==catPosition){
         reject;
      }
      exists{
      // mouse move
         for(Graph.Node n: mousePosition.getChildren()){
            catTurn{
               mousePosition=n;
            }
         }
      }
   }

   state catTurn {
      if (mousePosition==goal){
         accept;
      }
      if (mousePosition==catPosition){
         reject;
      }
      forall{
         // cat doesn't move
         mouseTurn{
         }
         // cat move
         for(Graph.Node n: catPosition.getChildren()){
            if(n!=goal){
               mouseTurn{
                  catPosition=n;
               }
            }
         }
      }
   }
}
```

Figure 6: Alter-Java program for Cat and Mouse.

minimal model. A configuration is thus characterized either by an atom or by a set of atoms. In the head state, we loop over all clauses which have the atom head in their head. If no such rule exists, we thus reject; otherwise we move to the state body for each such rule. Likewise, in the body state we loop over all atoms in the currently processed body. If no such atom exists (i.e., we arrived at a fact), we return accept, thanks to the universal quantification over the empty set. Otherwise, we call the head state again, for each atom in the body.

6.2 Cat and Mouse

Now consider the Cat and Mouse game from Section 4.2. The Alter-Java program in Figure 6 again defines two states: One for the mouse to move next and one for the cat. Recall that the mouse (i.e., Player 1) only wins when it reaches the goal. Both, the cat and the mouse always have the choice to pass moving and stay at their node.

As input, the program takes the gameboard Graph<Integer> gameboard (where Graph<Integer> is a simple class for storing directed graphs with integer nodes), the start position of the cat Graph.Node<Integer> catStart and of the mouse Graph.Node<Integer> mouseStart, as well as the goal (i.e., the mouse hole) Graph.Node<Integer> goal. In the work variables, we store the current position of the cat in Graph.Node<Integer> catPosition, and likewise, of the mouse in Graph.Node<Integer> mousePosition. These two variables represent the current game configuration.

The states contain the straightforward implementation of the alternating algorithm from Section 4.2; the realization essentially uses the same ideas as used in the implementation for the Horn Minimal Model problem; just note that the getChildren () method yields the set of nodes accessible from the current node and the loops range over these (possibly empty) sets.

We have implemented one simple optimization: We observe that, if the mouse passes, this leads to a cycle in the strategy when the cat passes. As the choices of the cat are universally quantified, we would always get such a cycle in a winning strategy and therefore there is a smaller winning strategy without passing. So it is dispensable to follow such a passing successor and we can remove this possibility for the mouse player.

6.3 Acyclic Geographic Game

Here we deal with another game from Section 4.2, the Acyclic Geographic Game. In Figure 7, we show an Alter-Java program which decides if there is a winning strategy in the Acyclic Geographic Game.

An instance of the Acyclic Geographic Game consists of the directed graph Graph<Position > gameBoard which is the board the game is played on and the startNode Graph.Node<Position> startNode where the token is placed at the beginning.

To describe the current game configuration we just need a cursor to the current node with the token, the program saves this in the work variable Graph.Node<Position> position.

As the Acyclic Geographic Game is a typical two-player game, the program uses two states, one for each player. These states simply implement the alternating algorithm from Figure 3. As before for the Definite Horn Minimal Model problem, our alter-Java algorithm has no explicit **accept** or **reject** statements. Instead, we use the fact that universal quantification over the empty set evaluates to an accept and existential quantification over the empty set evaluates to a reject.

6.4 Monotone Circuit Value Problem

We conclude our collection of example implementations with the Alter-Java program for the Monotone Circuit Value Problem from Section 4.3, as depicted in Figure 8. Notably, this program works with a single state but still has a quantifier alternation in the computation. A problem instance here

```
package at.ac.tuwien.dbai.alternation.examples;
import at.ac.tuwien.dbai.alternation.examples.GraphGame.Position;

atm Geographic (Graph<Position> gameBoard,
                Graph.Node<Position> startNode)
{
  Graph.Node<Position> position=startNode;

  state ExistsState {
    exists{
      for (Graph.Node<Position> node : position.getChildren()){
        ForallState{
          position=node;
        }
      }
    }
  }

  state ForallState {
    forall {
      for (Graph.Node<Position> node : position.getChildren()){
        ExistsState{
          position=node;
        }
      }
    }
  }
}
```

Figure 7: Alter-Java program for Acyclic Geographic Game.

consists of the circuit Circuit circ with an id for each gate, a series of input values **boolean**[] input and the id of output gate **int** outIndex. A configuration of the program is described by the gate Circuit.Gate<Circuit.GateTypes> gate that is currently evaluated. The state evaluate chooses the quantifier to use according to the type of the gate. If it is an OR gate, then the program quantifies existentially over the predecessor gates and universally for AND gates.

7 Implementation and Optimization

We have provided a framework for actually running Alter-Java programs. This framework consists of (1) an ANTLR Grammar [25] for Alter-Java, (2) a compiler that translates alternating programs into Java classes, and (3) a Java package for executing alternating programs. All resources are available at `http://www.dbai.tuwien.ac.at/proj/alternation/`.

Overall goal of the execution mechanism. The primary goal of executing an alternating program is to compute the leading-to-acceptance value of the initial configuration (consisting of initial state and initial values of the work variables). Our program execution mechanism explores the computation tree by means of a depth-first search. As we have already mentioned in Section 1, care has to be taken so as not to compute the leading-to-acceptance value of a configuration multiple times. For this purpose, we apply tabling [8, 31], i.e., all configurations already encountered are stored

```
package at.ac.tuwien.dbai.alternation.examples;

atm MCVP(Circuit circ,
         boolean[] input,
         int outIndex)
{
   Circuit.Gate<Circuit.GateTypes> gate=
      circ.getOutGate(outIndex);

   state evaluate {
      switch(gate.type) {
         case INPUT:
            if (input[circ.getInputIndex(gate)]==true){
               accept;
            } else {
               reject;
            }
         case AND:
            forall{
               for(Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()){
                  evaluate{
                     gate=tempGate;
                  }
               }
             }
         case OR:
            exists{
               for(Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()){
                  evaluate{
                     gate=tempGate;
                  }
               }
            }
         case OUTPUT:
            exists{
               for(Circuit.Gate<Circuit.GateTypes> tempGate: gate.getIn()){
                  evaluate{
                     gate=tempGate;
                  }
               }
            }
         default: reject;
      }
   }
}
```

Figure 8: Alter-Java program for MCVP.

in a table together with their leading-to-acceptance value. Hence, when the same configuration occurs again in some other node of the computation tree, then its leading-to-acceptance value is simply read from this table rather than computed again.

Infinite paths in the computation tree. But then the handling of cycles (and, hence, possibly infinite paths in the computation tree) has to be taken care of. One possibility to prevent cycles would be to request the programmer to provide a counter or similar techniques. Such counters are usually initialized by a polynomial w.r.t. the size of the input. For the Horn minimal model problem, we know that there can be no derivation whose length exceeds the number of rules. For

```
public class DefiniteHorn {
   private ATM <Inputtape , Worktape> atm;
   public class Inputtape {
      [...]
   }
   public class Worktape implements
                InterfaceWorktape<Inputtape> {
       [...]
   }
   private class head extends State<Inputtape , Worktape>{
       public ResultTuple<Worktape> compute (Worktape atmWorktape) {
          [...]
       }
    }
   private class body State<Inputtape , Worktape>  {
      public ResultTuple<Worktape> compute(Worktape atmWorktape) {
         [...]
      }
   }
}
```

Figure 9: The skeleton of class DefiniteHorn.

the Cat and Mouse game, the number of possible game configurations explored along each path cannot exceed n^2, where n is the number of vertices in the graph. For theoretical considerations, the counter technique is very useful in that it guarantees certain upper bounds on the length of computation paths. However, for practical computation, they are far from optimal. First, counters do not exclude cycles completely; they just guarantee a certain upper bound on the the worst-case effect of cycles. Second, counters would have to be introduced and maintained on the *program level*; instead we want to solve the problem of cycles *on the level of the execution mechanism*.

In fact, there is a surprisingly simple solution to this problem guided by the following observation: If a cycle (i.e., the repetition of a configuration) occurs in the execution of the alternating program, then the last edge of this computation path should return a "reject" answer to its parent node, since this cycle would never lead to a minimal solution. However, this "reject" answer should be considered as *local* in the sense that it has thus not been generally proved that the corresponding configuration really has a negative leading-to-acceptance value. It has only been detected that *on this computation path* we may assume a negative leading-to-acceptance value without loosing minimal solutions. Our execution mechanism therefore saves the current computation path in a list with some stack functionality. Whenever a computation revisits a configuration c saved in this list, the algorithm simple determines this edge as false – without modifying c and its leading-to-acceptance value in the table. Note that an edge (i.e., the transition from some configuration c to a configuration c') may lead to a cycle in one computation path while it is a sensible choice in another path. We resolve this problem by *marking* rejects caused by cycles. As soon as the first node of the cycle (which we refer to as the cycle leader), is decided to an accept or non-marked reject, the algorithm recomputes the leading-to-acceptance values of the states mapped to marked rejects. If the first cycle leader in the computation path is set to a marked reject, then we save this as an ordinary reject.

Strategy of the execution mechanism. In summary, the execution mechanism for alternating programs works as follows:

1. Check if the configuration is in the table and, if so, return the value from the table.
2. Check if the configuration is in the predecessor list and, if so, return a marked reject.
3. Save a copy of the configuration.
4. Compute the deterministic code of the state until there is an alternating construct.
5. If this construct is an **accept / reject** statement, then the algorithm returns **true / false**.
6. If this construct is a **forall** or **exists** construct, then the algorithm
   (a) computes all successor configurations and stores them until we make a decision about the original configuration;
   (b) evaluates the leading-to-acceptance value of the successors and combines them, according to the quantifier, to a leading-to-acceptance value for the original configuration.
7. Store the copy of the original configuration and the leading-to-acceptance value in the table.
8. If this configuration is a cycle leader, then we update the table according to the leading-to-acceptance value.

Compiler. Our Alter-Java compiler maps an alternating program to several Java classes within the same file, i.e., we handle nested classes. The main class adopts the name of the alternating program and has the following nested classes:

- a class Inputtape which is a container for the input variables of the alternating program
- a container class Worktape to handle the work variables.
- a class for each state in the alternating algorithm. Such a class inherits a method to set the input of the computation and defines a method which, when given a configuration (i.e., actual values of the work variables), computes the result of the state. Such a result is either accept, reject, or a set of successors with a quantifier.

In Figure 9, a schematic presentation of the class DefiniteHorn is given, corresponding to the Alter-Java program for the Definite Horn Minimal Model problem (Figure 5 in Section 6.1).

Methods of the main class. The main class itself has two methods. The compute method executes the alternating algorithm for a given problem instance (which has to be provided as input parameters on the method call) and returns whether the algorithm accepts or rejects. Thus, in case of DefiniteHorn, the interface is as follows:

```
public boolean compute(
   int goal,
   Map<Integer, Set<Set<Integer>>> rules)
```

```
public static void drawProofTree(
      ComputationTree<Horn.Worktape> cT){
   drawNode(cT,cT.getRoot());
}

public static void drawNode(
                    ComputationTree<Horn.Worktape> cT,
                    ComputationNode<Horn.Worktape> node){

   List<ComputationNode<Horn.Worktape>> children=
      cT.getChildren(node);

   if (children!=null){
      if (node.isAllQuantified()){
         System.out.println(node.getWorktape().head +
                            " :- "+
                            node.getWorktape().body);
         for (ComputationNode<Horn.Worktape> child : children){
            if (cT.get(child).equals(true)){
               drawNode(cT,child);
            }
         }
      } else {
         for (ComputationNode<Horn.Worktape> child : children){
            drawNode(cT,child);
         }
      }
   }
}
```

Figure 10: Extracting rules from a proof tree.

The getComputationTree() method is the interface to the computation tree of the alternating computation and returns that tree as an ComputationTree<Worktape> object.

As mentioned in Section 5, the computation tree allows us to reconstruct a solution in case the alternating program returns a positive result value. In our program execution based on tabling, we save the whole computation tree in the table. After solving the decision problem with the compute method, we can access the computation tree with the getComputationTree() method. This method returns an object of type alternation.runtime.ComputationTree, which offers a tree functionality to navigate between configurations and access to their work variables. So we can write programs to retrieve a winning strategy for a specific game or to get a proof tree for a Horn problem.

In Figure 10, we show an example program that extracts all rules from the proof tree of a definite Horn minimal model instance. We simply do this by recursive access to the leading-to-acceptance successors starting with the initial configuration (i.e., the root of the computation tree).

8 Experimental Results

In this section we report on first experiences gained with our Alter-Java framework. The primary goal of our tests was to compare our alternating programs with corresponding deterministic programs. Moreover, we wanted to test the performance gain of the sophisticated cylce detection

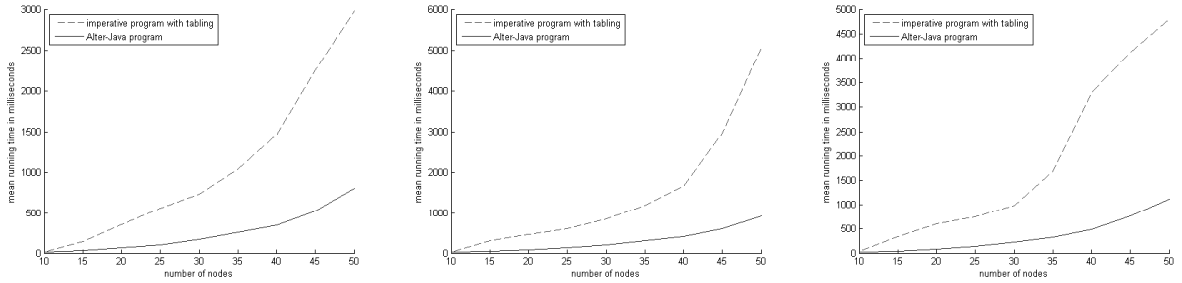
```
package at.ac.tuwien.dbai.alternation.example;

public class CatAndMouse{
  private static Graph<Integer> gameboard;
  private static Graph.Node<Integer> goal;

  public static boolean decideCatAndMouse(Graph<Integer> gameboard,Graph.
      Node<Integer> catStart,Graph.Node<Integer> mouseStart,Graph.Node<
      Integer> goal){
    CatAndMouse.gameboard=gameboard;
    CatAndMouse.goal=goal;
    return computeMouse(catStart,mouseStart,gameboard.size()*gameboard.size
        ());
  }

  public static boolean computeMouse(Graph.Node<Integer> catPosition, Graph
      .Node<Integer> mousePosition, int counter){
    if (mousePosition==catPosition || counter<0) return false;
    for(Graph.Node child: mousePosition.getChildren()){
      if(computeCat(catPosition, child,counter-1))return true;
    }
    return false;
  }

  public static boolean computeCat(Graph.Node<Integer> catPosition, Graph.
      Node<Integer> mousePosition, int counter){
    if (mousePosition==goal) return true;
    if (mousePosition==catPosition) return false;
    if (!computeMouse(catPosition, mousePosition,counter-1)) return false;
    for(Graph.Node child: catPosition.getChildren()){
      if(child!=goal && !computeMouse(child, mousePosition,counter-1))
          return false;
    }
    return true;
  }
}
```

Figure 11: Imperative realization of the Cat and Mouse Game.

mechanism presented in Section 7 compared with a naive counter-based cycle detection. For the comparison of the alternating and deterministic programs, we were interested in a runtime comparison and, moreover, we wanted to quantify the amount of extra code required in case of an efficient deterministic program. Below we summarize the results obtained for the Cat and Mouse problem.

In order to compare the alternating Cat and Mouse program from Section 6.2 with an imperative program, we first have to decide how much effort should be put into the development of the imperative program. In principle, one could start with a straightforward program which implements the exists- and forall-conditions by means of for-loops and recursion. The resulting imperative program would look quite similar to our alternating program from Section 6.2. However, without the embedding into a framework including cycle detection, such an attempt fails. We therefore have to incorporate some kind of cycle detection into our imperative program. The resulting program is depicted in Figure 11.

Note that if we omit the counter-based cycle detection from that program, then termination could not be guaranteed. Indeed, our program corresponds to a depth-first search through the com-



(a) p=0.3 (b) p=0.5 (c) p=0.7

Figure 12: Simple tabled evaluation vs. tabled evaluation with cycle detection, Cat and Mouse programs mean running time (y-axis) according to the number of nodes (x-axis) and the edge density p.

putation tree. Since the computation tree, in general, contains infinite branches (due to cycles in the configurations), the cycle-detection is crucial to guarantee termination. However, the program is still far from satisfactory. Without some improvement in the form of tabling, our imperative program has, in general, exponential runtime behaviour. We do not explicitly give the code for table management here. Our imperative program from Figure 11 increases to approx. 120 lines of Java code if tabling is incorporated.

We carried out tests with the following two programs: on the one hand, the imperative program with counter-based cycle detection and tabling; on the other hand, the alternating program from Section 6.2 when executed in our Alter-Java framework (which contains a more sophisticated cycle detection mechanism, see Section 7). We obtained test cases by varying the graph size from 10 to 50 vertices and randomly generating graphs for these cases. In order to control the number of edges, we fixed a parameter p indicating the probability with which two arbitrary vertices are adjacent or not. We carried out tests for parameter values $p = 0.3$, $p = 0.5$, and $p = 0.7$. For each of the resulting cases (determined by number of vertices and parameter value p) we randomly generated 100 instances of the Cat and Mouse game (i.e., a graph plus start position of mouse, start position of cat, and goal). In Figure 12, the average run times of our alternating program and of the above imperative program are juxtaposed. These run times were obtained on an ordinary desktop PC with the following characteristics: (1) CPU: PE T300 Quad Core Xeon X3323, 2.5GHz, 2x3MB, 1333MHz FSB; (2) RAM: 4GB DDR2 667MHz Memory; (3) OS: SUSE Linux Enterprise Server 10; (4) Java: Java(TM) SE Runtime Environment (build 1.6.0 10-rc-b28). The Java virtual machine was started with the parameters *-Xms512m -Xmx1536m -Xss2024k*. So the programs where restricted to 1.5 GB memory.

The test results suggest that even more effort has to be put into the imperative program – incorporating a more sophisticated cycle detection mechanism, e.g., like the one described in Section 7. But then we ultimately arrive at an imperative program which does all the work provided by our Alter-Java framework without a significant gain in performance. Note that the compilation of our alternating Cat and Mouse program results in Java code which – together with the tabling and cycle detection code of our framework – amounts to approx. 700 lines of code. Hence, the imper-

ative program in Figure 11 has to be increased by a factor of > 10 in order to arrive at a similar performance as our alternating program from Section 6.2. In contrast, the programmer of Alter-Java can fully concentrate on the declarative meaning of the alternating program without worrying about the execution of this program. Avoiding repeated computation of the same configurations and detecting cycles is entirely in the responsibility of the execution mechanism and not of the programmer.

9 Conclusion

In this paper, we have presented alternation as a useful programming paradigm, which we have actually incorporated into Java. We have shown that many problems (also outside the domain of games) admit very succinct and intuitive characterizations via alternation. A framework for this extended language (referred to as Alter-Java) consisting of an ANTLR Grammar for Alter-Java, a compiler that translates alternating programs into Java classes, and a Java package for executing alternating programs are available at `http://www.dbai.tuwien.ac.at/proj/alternation/`. We have also reported on our experience gained from tests with this framework. In particular, it has turned out that a lot of code can be avoided by writing alternating programs and using our framework instead of implementing functionalities like cycle detection separately for every problem.

So far, we have mainly concentrated on problems from P (the class of problems solvable in polynomial time), which coincides with ALOGSPACE. Recall that alternating algorithms for problems in ALOGSPACE have, in general, an exponentially big computation tree. An important subclass of ALOGSPACE and, hence, of P is LOGCFL – the subclass of problems in ALOGSPACE where every successful computation is guaranteed to have a computation tree of polynomial size [30]. An important aspect of LOGCFL is that the problems in this class are highly parallelizable. Our next step of research in this area is therefore devoted to LOGCFL and to optimizations of the execution mechanism for alternating programs, exploiting the possibilities of parallelization. As target application, we primarily envisage constraint satisfaction problems (CSPs). Note that one of the most powerful concepts of identifying tractable fragments of CSPs via structural decomposition are so-called hypertree decompositions [14]. It was shown that deciding if a CSP instance has a hypertree decomposition of some width w (for fixed w) is in LOGCFL. The actual hypertree decomposition is very closely related to (and can be easily retrieved from) the computation tree of a successful computation of the alternating algorithm. Likewise, in case such a hypertree decomposition exists, solving the CSP instance itself is in LOGCFL [15]. We therefore plan to explore the potential of alternating programs for LOGCFL-problems – in particular in the area of CSPs.

Another area where alternation naturally occurs is Computer Aided Verification (CAV). Many logics in this field rely on alternation (see, e.g., [23, 33]). Exploring possible applications of alternating programs to CAV is, therefore, also an important item on the agenda for future work.

References

[1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java constraint kit. *Electr. Notes Theor. Comput. Sci.*, 64, 2002.

[2] K. R. Apt and A. Schaerf. Search and imperative programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 67–79, 1997.

[3] K. R. Apt and A. Schaerf. The Alma project, or how first-order logic can help us in imperative programming. In *Correct System Design*, volume 1710 of *LNCS*, pages 89–113. Springer, 1999.

[4] K. R. Apt and A. Schaerf. Programming in Alma-0, or imperative and declarative programming reconciled. In *Proceedings of Frontiers of Combining Systems*, pages 1–16. Research Studies Press Ltd, 2000.

[5] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[6] A. K. Chandra and L. J. Stockmeyer. Alternation. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS'76)*, pages 98–108. IEEE Computer Society, 1976.

[7] A. K. Chandra and M. Tompa. The complexity of short two-person games. *Discrete Appl. Math.*, 29(1):21–33, 1990.

[8] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[9] J. Cohen. Non-deterministic algorithms. *ACM Comput. Surv.*, 11(2):79–94, 1979.

[10] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[11] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 90–96. Professional Book Center, 2005.

[12] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference (ESWC'06)*, volume 4011 of *LNCS*, pages 273–287. Springer, 2006.

[13] A. S. Fraenkel and D. Lichtenstein. Computing a perfect strategy for n*n chess requires time exponential in n. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 278–293. Springer, 1981.

[14] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[15] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[16] M. Grabmüller. Constraint Imperative Programming. Diploma thesis, Technische Universität Berlin, February 2003.

[17] M. Grabmüller. Multiparadigmen-Programmiersprachen. Research report 2003-15 in *Forschungsberichte Fakultät IV – Elektrotechnik und Informatik*, Technische Universität Berlin, October 2003.

[18] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press Inc., 1995.

[19] D. Harel. And/or programs: A new approach to structured programming. *ACM Trans. Program. Lang. Syst.*, 2(1):1–17, 1980.

[20] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. In *STOC '74: Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 40–46. ACM, 1974.

[21] R. E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[22] S. Miyano, S. Shiraishi, and T. Shoudai. A list of P - complete problems. *RIFIS Technical Report*, 17:1–69, 1989.

[23] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 422–427. IEEE Computer Society, 1988.

[24] C. M. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[25] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[26] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[27] A. Radensky. Toward integration of the imperative and logic programming paradigms: Horn-clause programming in the Pascal environment. *SIGPLAN Not.*, 25(2):25–34, 1990.

[28] S. Reisch. Gobang ist PSPACE - vollständig. *Acta Inf.*, 13:59–66, 1980.

[29] J. M. Robson. The complexity of Go. In *Proceedings IFIP Congress*, pages 413–417, 1983.

[30] W. L. Ruzzo. Tree-size bounded alternation. *J. Comput. Syst. Sci.*, 21(2):218–235, 1980.

[31] K. Sagonas and P. J. Stuckey. Just enough tabling. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 78–89. ACM, 2004.

[32] E. Y. Shapiro. Alternation and the computational complexity of logic programs. *J. Log. Program.*, 1(1):19–33, 1984.

[33] M. Y. Vardi. Alternating automata and program verification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 471–485. Springer, 1995.

[34] N.-F. Zhou, S. Kaneko, and K. Yamauchi. DJ: A Java-based constraint language and system. In *Proceedings of the Annual JSSST Conference*, 1998.