Bachelorarbeit

Program Sketching Using Craig Interpolants

Thomas Eickhoff

Technische Universität Kaiserslautern
Fachbereich Informatik

Betreuer

Prof. Dr. Klaus Schneider

Dr.-Ing. Andreas Morgenstern

Abgabedatum

Kaiserslautern, den ___.05.2012

"Beware of bugs in the above code; I have only proved it correct, not tried it."
– Donald Knuth

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 14. Mai 2012

Thomas Eickhoff

Contents

Abstract **1**

1 Introduction **3**

2 Motivating Example **5**
2.1 Binary Search 5

3 Preliminaries **9**
3.1 Kripke Structures and Specifications 9
3.2 Kripke Structures with Oracles and Implementations 11
3.3 Interpolation Algorithm 12
3.4 The Programming Language Quartz 13

4 Sketching Using Interpolation **15**
4.1 Sketching as a Model Checking Problem 15
4.2 Model Checking Based on Interpolation 15
4.3 Sketching Workflow 19

5 Implementation **21**
5.1 General Structure 21
5.2 Translating a Program to SMT Formulas 22
5.3 Modifications To McMillan's Algorithm 24
5.4 Limitations 26
5.5 Examples and Performance 26

6 Conclusion **31**

Bibliography **33**

Abstract

English

Program sketching is an approach to software synthesis and verification where the programmer leaves certain details of his program for a synthesizer to decide. The synthesizer chooses from a set of possibilities such that a given specification for the resulting program is correctly verified. This has been reduced to a model checking problem by Morgenstern and Schneider. The focus of this work is to adapt an efficient algorithm for model checking described by McMillan to make it usable for sketching problems.

A combined algorithm is developed and a rudimentary implementation is described and analyzed for performance.

Deutsch

Program-Sketching ist ein Verfahren zur Software-Synthese und -Verifikation, bei dem der Programmierer gewisse Details der Implementierung dem Synthesizer überlässt. Der Synthesizer wählt aus allen möglichen Implementierungen eine aus, die die Spezifikation des ursprünglichen Programms erfüllt. Dieses Verfahren wurde von Morgenstern und Schneider auf ein Model-Checking-Problem reduziert. Das Ziel dieser Arbeit ist es, ein von McMillan vorgestelltes effizientes Model-Checking-Verfahren so anzupassen, dass es zum Sketchen verwendet werden kann.

Ein neuer Algorithmus, der die Verfahren von Morgenstern/Schneider und McMillan verbindet, wird entwickelt und eine rudimentäre Implementierung desselben wird beschrieben sowie im Hinblick auf ihre Performance analysiert.

1 Introduction

> "Sketching is an approach to automated software synthesis where the programmer develops a partial implementation called a sketch and a separate specification of the desired functionality. A synthesizer tool then automatically completes the sketch to a complete program that satisfies the specification."
> – Morgenstern and Schneider in [MS11]

To put it slightly more simple, sketching simplifies the work of the programmer by enabling him to omit certain details of a program (e.g., if he is uncertain whether `i<1` or `i<=1` is the correct loop condition) and leave the choice up to the synthesizer. In 2011, [MS11] presented an approach to automatically complete such a partial implementation of a program (called a "sketch") such that the resulting program satisfies a given specification. This was achieved by reducing the sketching problem to CTL* model checking.

This model checking problem can then be solved by using a (BDD-based) symbolic model checker. SAT-based bounded model checkers, which are generally faster, cannot be used directly as there is no upper bound for the program's complexity. However, an efficient algorithm for unbounded SAT-based model checking can be found in [McM03].

The aim of this work is to combine both approaches into one efficient sketching algorithm[1]. In particular, a program was developed that completes program sketches written in the programming language Quartz accompanied by a specification written in a subset of the specification logic LTL [Sch03].

The outline of this paper is as follows: chapter 2 contains an example use case for sketching. chapter 3 summarizes the theoretical foundations necessary to understand the algorithm as well as its implementation. In chapter 4 the actual algorithm is described. chapter 5 highlights several details concerning the implementation of the algorithm and compares its performance to the implementation used in [MS11]. Finally, the results are summarized in the Conclusion.

[1] Because of this, many sections of this work are based on one of the two papers. I have tried to mark all citations accordingly.

2 Motivating Example

> “Errare humanum est.”
> – Seneca

This chapter gives a short example of a program that could make use of sketching. The example is taken from [MS11]. A formulation of a corresponding incomplete algorithm (or “program sketch”) as a sketching problem and an evaluation of its performance can likewise be found in the paper. This example should be sufficient to show that automatic verification is necessary for safety-critical software. And if the programmer, when given the specification of a program, can’t be trusted to get all the details right, then why should these details not not be figured out by the synthesizer instead?

2.1 Binary Search

Programmers are humans, and thus they make mistakes. A prominent example of this is the fact that although the algorithm for binary search was published in 1946, the first correct implementation was given much later (as late as 1960 according to Knuth [Knu98]).

An implementation of binary search is given in Figure 2.1. It was written using the synchronous programming language Quartz [Sch09], but it would not look much different if it were implemented in C. Binary search searches a sorted list for a given value by looking at the middle element of the array and comparing it to the searched value. If the element is larger, binary search continues to look for the value in the left half of the array, if the element is smaller, binary search iterates along the right half. If the value is equal to the element, binary search returns the elements index.

After you have looked at the impelementation and convinced yourself that this implementation is correct (and in this context I’d like to highlight that, according to [MS11], this code is “contained in many textbooks on algorithms, and is even

```
macro M = 10;
macro N = 2;

module BinarySearch([N]int{M} ?a, int{M} ?v, nat{N+1} i) {
  //input:
  //  a -- the array
  //  v -- the value we are looking for
  //output:
  //  i -- the index where v is found
  nat{N} left,right,mid;
  left = 0;
  right = N - 1;
  i = N;
  while(left<right & i==N) {
    mid = (left + right) / 2;
    if (a[mid]<v)
      next(left) = mid + 1;
    else if (a[mid]>v)
      next(right) = mid - 1;
    else                        // v has been found
      next(i) = mid;
    pause;
  }
}
```

Figure 2.1: Almost Binary Search

taught (and verified!) each year in courses in computer science”), let me tell you that it is, in fact, incorrect[1].

The program’s idea seems simple enough. `left` and `right` mark the borders of the sub-array that remains to be searched for the value `v`. Before the first iteration they point to (i.e. contain the index of) the first and last element of the array `a` respectively. In each iteration, v is compared to the middle element `mid` of the remaining sub-array (which is at position `(left + right) / 2`). According to the result of this comparison either the left or the right border is moved in such a way that only the values left (if `mid` was larger than `v`) or right (if `mid` was smaller than `v`) of `mid` remain in the array. If `v` is equal to `mid`, `i` is set to `mid`, thus ending the loop and “returning” the index at which `v` was found. The computation also aborts if `left < right` no longer holds, with `i == N` being returned as an error value (which means that `v` was not found in the list).

[1]It should be noted that the errors are not specific to Quartz, but could appear in any C-like language.

In spite of its simplicity, the program contains not only one, but several errors. Morgenstern and Schneider give the example where an array containing the elements `0` and `1` is searched for `v=1` [MS11]. A nice property of this array is that it is not too hard to think about whether I am talking about an index or the value stored at this index, as both the index and the value of the first element are 0, and both the index and the value of the right field are 1. For narrative purposes I assume that Ben Bitdiddle [AS96] has written the exact code in Figure 2.1 and wants to test it with this example.

Initially, the algorithm has to search the whole array, thus `left=0` and `right=1`. This leads to `mid=(1+0)/2`, which is `0`. Thus, `a[mid] == 0` and `v== 1`, i.e. `a[mid] < v`. Accordingly, left is assigned the value `mid+1`, or `1`. The loop terminates, because `left` is no longer smaller than `right`, although `i` is till set to `N`, which means that `v` was not found, even if it was obviously in the array.

Ben, who is now a little embarrassed, decides to change the loop condition to `left<=right & i == N`. However, the program is still incorrect. Ben decides to test his algorithm with an array containing the two values `2` and `3` (which is the second example used by Morgenstern and Schneider). He is still searching for `v=1`.

The algorithm starts out the same way as it did in the previous example, with `left=0` and `right=1` leading to mid $= 0$. This time `a[mid]` is `2`, and (with `v` still being `1`) `a[mid] > v`, so right is assigned the value `mid-1`, i.e. `-1`. Ben gasps in horror as he realizes that this will lead to a nasty index error.

According to [Sha09], almost all binary search implementations are broken. While the errors might not be as "obvious" as in this example, arithmetic over- and underflows may occur in the computation of "`mid`".

This is not only theoretical nitpicking, but has practical implications. For 9 years, the Java library contained a faulty implementation of binary search. Of course, these problems could be found by model checking. But with sketching these problems could be avoided, assuming the programmer was uncertain whether she should chose one statement or another. For example, the programmer might be uncertain about the loop condition, i.e., whether it should contain `left<right` or `left<=right`. Using sketching, the programmer could then use `(sel ? left<right : left<=right)`, marking `sel` as a new boolean oracle variale. While more complex choices might require additional syntax in order to remain practical, this example uses the standard ternary operator found in many programming languages like C [KR88] to show that `left<right` should be chosen if `sel` is true and `left<=right` should be chosen otherwise.

3 Preliminaries

“ ’Begin at the beginning,’ the King said gravely, ’and go on till you come to the end: then stop.’ ”
– Lewis Carroll

This chapter gives an overview of the necessary background knowledge. It covers the basics of sketching as well as interpolation-based model checking. section 3.1 describes Kripke structures and defines a rather primitive subset of LTL, which is used as a specification language. section 3.2 shows how Kripke strucutres can be extended to model sketching problems. Whereas the former two sections were based mostly on the work of Schneider and Morgenstern [MS11], section 3.3 summarizes the interpolation algorithm used by McMillan [McM03]. Finally, in section 3.4, a short introduction to the syntax of Quartz is given (which explains only as much of the language as is absolutely necessary to understand the code examples in this thesis).

3.1 Kripke Structures and Specifications

As mentioned before, I will use Kripke structures as an abstract model of programs, which can be thought of as roughly equivalent to finite automata.

Definition 1 (Kripke Structure)**.** [Sch03] A Kripke structure $K = (State, Init, Trans, Label)$ for a finite set of variables V is given by a finite set of states $State$, a set of initial states $Init \subseteq State$, a transition relation $Trans \subseteq State \times State$, and a label function $Label : State \to 2^V$ that maps each state to the set of variables that hold in this state[1].

The states of a Kripke structure can be thought of as roughly equivalent to points in the corresponding program’s control flow. I will define (infinite) paths through a Kripke structure, which correspond to control paths:

[1]In order to make the following sections easier to read (and to avoid naming conflicts in Section section 5.3), the names of the tuple’s elements have been changed, otherwise this definition is taken directly from [Sch03].

Definition 2 (Path)**.** [Sch03] Given a set of atomic propositions V, an infinite path is a function $\pi : \mathbb{N} \rightarrow 2^V$. We denote the i-th state of the path π as π^i for $i \in \mathbb{N}$. Using this notation, paths are often given in the form $\pi^0\pi^1\dots$. The path starting at t is moreover written as $(\pi, t) := \pi^t\pi^{t+1}\dots$

In order to solve model checking problems over Kripke structures using a SAT solver, $Init$ and $Trans$ **can be expressed as logical formulas over** V**.** These formulas can naturally be derived from the variables that hold in each state (i.e., the states' *Labels*). Whenever $Init$ or $Trans$ is used in a context where a logical formula is expected, the equivalent formula is meant. In particular, I will generate different instances of these formulas for each step k of a path, wich will be represented as $Init_k$ resp. $Trans_{k,k+1}$ (the transition from π^k to π^{k+1}). For each state π^k we will define a fresh set of Variables V_{π^k}, which is obtained by appending the suffix "___step___k" to all variable names in V. $Init_k$ will contain only variables form V_{π^k}, whereas $Trans_{k,k+1}$ will contain variables from V_{π^k} and $V_{\pi^{k+1}}$.

In order to solve sketching problems, another formula is needed: The program's specification (hereafter called *Spec*). Just like Init and Trans, Spec can be instantiated as $Spec_k$, using only variables from the set V_{π^k}. In principle any temporal logic could be used to define the spec (McMillan uses LTL, Morgenstern and Schneider use the more expressive CTL*). However, since I wanted to finish my program on time, I use a rather primitive subset of LTL to define **safety conditions** (conditions that must hold in any state of the Kripke structure). These will be written as $AG\Phi$, where Φ is a logical formula using the following[2]:

- boolean constants and
- operators ($\neg$, $\vee$, $\wedge$, ...) and relations ($=$, $\neq$)
- integer constants, operators, relations
- variables defined in the Kripke structure
- references to variables in the successor state ($next(v)$)

[2]This corresponds to an Averest BoolExpr, see section 5.2

3.2 Kripke Structures with Oracles and Implementations

Definition 3 (Kripke Structure with Oracles)**.** [MS11] A Kripke structure with oracles is a Kripke structure $K = (State, Init, Trans, Label)$ as before where $State = Q \times O$ is the product of a state set S with a oracle set O. An *implementation* $K = (State_o, Init_o, Trans_o, Label)$ for some $o \in O$ is a restriction of K given by

- $S_o = S \cap \{o\}$
- $Init_o = Init \cap \{o\}$
- $Trans_o\left((q, \hat{o}), (q', \hat{o}')\right) \leftrightarrow Trans\left((q, \hat{o}), (q', \hat{o}')\right) \wedge \hat{o}' = \hat{o} = o$

According to [MS11], the sketching problem can be formulated in terms of these extended Kripke structures: Given a program sketch (i.e. a Kripke structure K with oracles) and a specification *Spec*, is there an implementation K_o such that $K_o \models Spec$?

This question was answered by [MS11]. In their paper, they give the following theorem:

Theorem 4. *Let $K = (Q \times O, Init, Trans, Label)$ be a Kripke structure with oracle set O and a CTL* formula Φ. There exists a Kripke structure $K' = (State', Init', Trans', Label')$ whose size is polynomial in the size of K such that the following holds: there exists an implementation K_o of K satisfying Φ if and only if $K' \models EXAX\Phi$.*

Proof. We define $K' = (State', Init', Trans', Label')$ by adding new states q_0 and q^o_{sel} for each value of the oracle set $o \in O$ as follows: □

- $State' = Q \times O \cup \{q_0\} \cup \{q^o_{sel} \mid o \in O\}$
- $Init' = \{q_0\}$
- $R' = \begin{pmatrix} \{(q_0, q^o_{sel}) \mid o \in O\} \cup \\ \{((q^o_{sel}, (q, o)) \mid q \in Init \wedge o \in O)\} \cup \\ \{((q, o), (q', o)) \mid ((q, o), (q', o)) \in Trans \wedge o \in O\} \end{pmatrix}$

"The intuitive idea behind the construction of K' is as follows: The only initial state is q_0, and q_0 has transitions to every state q^o_{sel} for each value o of the oracle set O. In each state q^o_{sel}, a choice is made for the oracle set that is kept invariant on all transitions of K': The transition relation $Trans'$ of K' is defined such that from q^o_{sel} there are only transitions to states (q, o) where q is an initial state of K and o is the value chosen by q^o_{sel}. Moreover, the transitions of K are pruned so that a transition leads from a state (q, o) to a state (q', o') only if $o = o'$ holds (and there was already this transition in K). Hence the sub-structure starting at q^o_{sel} is bisimulation equivalent to the implementation K_o.

Now remember that a Kripke structure K satisfies a CTL* specification Φ if and only if all initial states of K satisfy Φ. Hence there does exist a implementation K_o if and only if the sub-structure rooted at $q^o_{sel} \models EXAX\Phi$ and accordingly if and only if $K' \models EXAX\Phi$. □"[MS11]

3.3 Interpolation Algorithm

In addition to the previous definitions and construction concerning sketching, the following sections of this thesis will use an interpolation-based algorithm to efficiently solve sketching problems. Interpolation is the method of computing interpolants as defined in [Cra57]. Basically, an interpolant is a formula that can be computed from two logical formulas A and B (with $A \wedge B$ unsatisfiable), such that

- A implies P
- $P \wedge B$ is unsatisfiable
- P refers only to the common variables of A and B

The intuition of the first property is that A is a more general formula than P, while the second property states that P retains A's property of being satisfiable in conjunction with B. In this thesis, I will use logical formulas to describe sets of states of a program. Accordingly, the interpolant P describes more states than the initial formula A.

Since I left most of the hard work in computing the required interpolants, I will only give a general idea of the procedure described in [McM03]. The exact procedure used by my MathSAT-based implementation can be found in [Gri12].

The first step to compute an interpolant of two formulas is to prove that the conjunction of the two formulas is unsatisfiable.

It is usually taught in introductory courses on propositional logic that every logical formula can be transformed to Conjunctive Normal Form (CNF), i.e., a conjunction of disjunctions of variables or negated variables[3]. The disjunctions are called clauses. A formula in CNF can be interpreted as a set of clauses, the conjunction of two such formulas can be seen as the union of the corresponding clause sets.

The unsatisfiability of such a clause set can be proven using the Davis-Putnam algorithm. Basically, the idea of this algorithm is to construct a directed acyclic graph, where each root is a clause from the clause set, and each inner node is the

[3]Keep in mind that every logical formula can be transformed to CNF and thus the following algorithm works even for the formulas in this thesis that are not in CNF.

For readability reasons, logical formulas in this thesis will not always be in CNF. They are assumed to be implicitly transformed as necessary (in fact, in the actual implementation this transformation was left to the SAT solver).

resolvent[4] of its two parents. If the empty clause can be rached as a leaf, the clause set is unsatisfiable. In this case, the graph is called a proof of unsatisfiability for the clause set.

Given two formulas A and B and a proof of unsatisfiability Π, an interpolant of A and B can be computed by defining a logical formula $p(c)$ for each node c in Π such that:

- if c is a root, then
  - if $c \in A$ then $p(c)$ is the disjunction of all literals in c that appear in both A and B
  - else $p(c)$ is *true*
- else, let c_1 and c_2 be the predecessors and let v their pivot variable
  - if v only appears in A, then $p(c) = p(c_1) \vee p(c_2)$
  - else $p(c) = p(c_1) \wedge p(c_2)$

The interpolant of A and B is the formula assigned to the leaf node (the empty clause).

3.4 The Programming Language Quartz

All example program sketches in this thesis are written in the synchronous programming language Quartz [Sch09]. This is by no means a necessity, you could use any programming language, assuming you have a compiler that outputs Averest Kripke structures (or that you are willing to adapt my program). While most of Quartz's syntax looks familiar to anyone who has programmed in a syntactically C-like language (e.g., C, C++, Java, C#, and, to a lesser extend, Javascript or PHP), some features are different enough to require explanation.

Please note that this section is not a comprehensive introduction to Quartz, its only purpose is to enable a reader who is familiar with at least one C-like programming language to understand the code samples in this thesis.

Quartz is a synchronous programming language. As such, all statements are executed concurrently and in zero time, unless separated by a `pause;` statement. Because of this, pause statemets are the only meaningful points in the program's control flow, which roughly correspond to the states in the equivalent Kripke structure. Everything that happens between two pause statements is summarized into one **macro step**.

[4] A resolvent of two clauses $c_1 = v \vee A$ and $c_2 = \neg v \vee B$ is $A \vee B$, if $A \vee B$ is not tautological. v is called the pivot variable of c_1 and c_2. Obviously the resolvent of $c_1$and c_2 is unique (if it exists). Also, it is implied by $c_1 \wedge c_2$.

Assignment works mostly as expected, the statement `x = y` evaluates `y` and stores the result in the variable `x`. While these assignments work immidiately, a delayed assignment `next(x) = y` waits until the following macro step.

Quartz supports several data types. In this thesis, the types **`bool`** and **`nat{N}`** are used. While a `bool` value can either be true or false, a `nat{N}` value can be any integer in the interval $[0, N-1]$.

Finally, specifiactions can be added at the end of a Quartz program using a pair of curly braces preceded by the keyword **`satisfies`**. Both specifications and pause statements are sometimes preceeded by a label
(`some_label : pause;` or `whatever : assert A G (x == true);`) which can be used to identify the corresponding point in the program's control flow.

4 Sketching Using Interpolation

In this chapter, I present an algorithm for the sketching problem using interpolation based model checking. Conceptually, this algorithm is a combination of Morgenstern/Schneider's sketching algorithm [MS11] and McMillan's model checking procedure [McM03]. Because of this, these two algorithms are described in section 4.1 and section 4.2. section 4.3 shows how these two algorithms can be combined.

4.1 Sketching as a Model Checking Problem

After understanding section 3.2, reducing sketching to a model checking problem is not difficult. When translating a program sketch to a Kripke structure, the synthesizer introduces a new variable for each implementation detail that the programmer left undecided. Each value of this variable corresponds to one possible choice for this detail. These variables are the oracle variables mentioned in section 3.2. The translation then results in a Kripke structure as described in 4, where a single initial state q_o is connected to several states q_0^{sel}, which are the initial states of different implementations of the sketch.

Now, model checking can be used to find out which implementations satisfy the specification and choose any of those as the answer to the sketching problem.

4.2 Model Checking Based on Interpolation

After reducing the sketching problem to a model checking problem, McMillan's interpolation-based model checking algorithm can be used to solve it. This section presents a variant of said algorithm, which uses Kripke structures (instead of McMillan's finite automata) as its model of computation.

McMillan uses bounded model checking as a starting point. In bounded model checking, k transitions through the Kripke structure are computed and it is checked if a bad state (a state violating the program's specification) can be reached. Using the formula sets defined in section 3.1, this can be formulated as follows: Starting with the initial constraints ($Init$) in state s_0, the transition relation ($Trans$) is applied k times, and check if *Spec* can be violated in any state up to s_k (which is equivalent to asserting the negated spec for all states up to k). The formula that is used by the SAT solver is thus:

$$Init_0 \wedge \left(\bigwedge_{0 \leq i < k} Trans_{i,i+1} \right) \wedge \left(\bigvee_{0 \leq i \leq k} \neg Spec_i \right)$$

Figure 4.1 is a visualization of this approach. The whole image is the state space, bounded model checking can be seen as checking if s_k intersects with Bad. For further detail about the formulas used in BMC, see section 5.2.

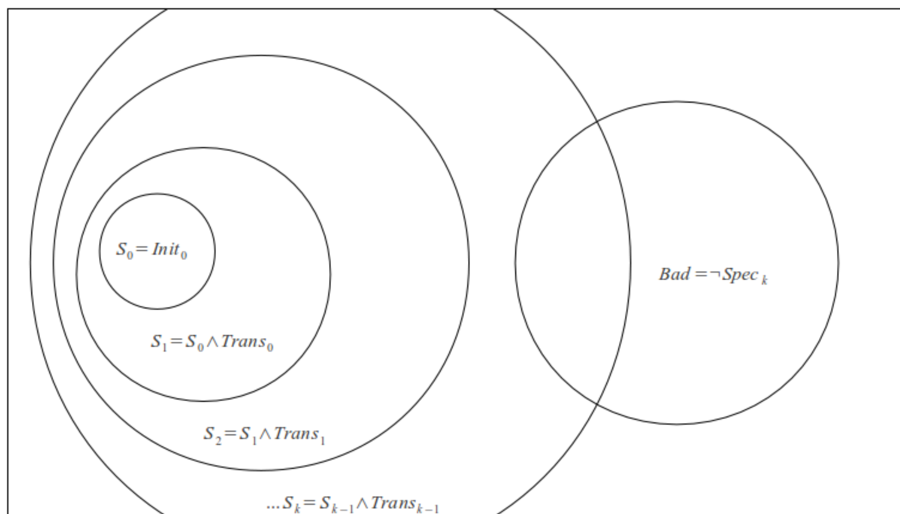

Figure 4.1: Bounded Model Checking

Using this approach, it can be checked if a bad state can be reached in k steps. It can not, however, be verified that a bad state cannot be reached at all (i.e., prove that the program is correct). In order to use SAT-based model checking for sketching purposes, an unbounded model checking procedure is needed, such as the algorithm proposed by McMillan.

In order to find arbitraryly long paths to bad states, McMillan uses interpolation to check if a bad state is potentially reachable, and (if that is the case) increases k. More precisely, he divides the formula for bounded model checking into two parts: $Init$ and the first transition are used as one interpolation group A, whereas the remaining $k-1$ transitions and the negated specs are used as interpolation group B. If a bad state is proven unreachable in k steps, the interpolant P of both groups is computed from the proof of unsatisfiability of $A \cup B$. By definition, this interpolant contains only the common variables of both groups (which are exactly the variables representing state 1[1]). By definition, A implies P, i.e., P is an over-approximation of the states reachable in one transition. This over-approximation is then iteratively

[1] McMillan mentions this fact (and it is rather obvious, as the clauses in group A contain the variables of s_0 and s_1, whereas the clauses in group B contain variables of s_1 to s_k), but it is never used in a proof.

used instead of A to find out if a bad state is potentially reachable. If that is the case, k is increased accordingly. Otherwise, the program was successfully proven correct.

McMillan states that this procedure must terminate for “sufficiently large values of k”. He gives an upper bound of the “*reverse depth* of M”, which in turn is bounded by $2^{|V|}$, but points out that “in most practical cases [it] is much smaller”[McM03, Non12].

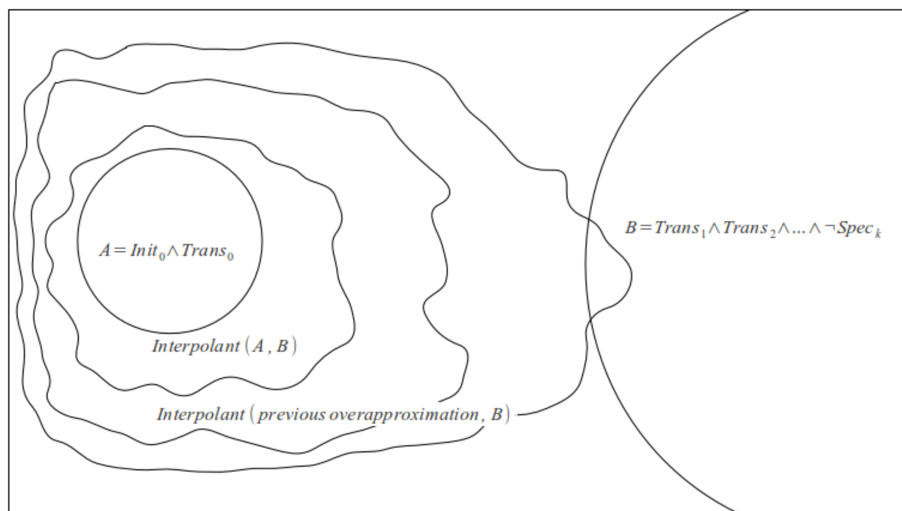

Figure 4.2: Using Interpolants to generate successive over-approximations of $Init_0 \wedge Trans_0$

A visualisation for this approach is given in Figure 4.2. Note that this visualizes the successive over-approximations of the first transition from the initial states rather than the several transitions of bounded model checking (which are all included in interpolation group B).

The two interpolation groups can be formalized using the following two formulas $pref\,(K)$ (for group A) and $suff_k\,(K)$ (for group B), where K is a Kripke structure:

$$Pref\,(K) = Init_{-1} \wedge Trans_{-1,0}$$

$$Suff_k\,(K) = \left(\bigwedge_{0 \leq i < k} Trans_{i,i+1} \right) \wedge \left(\bigvee_{0 \leq i \leq k} \neg Spec_i \right)$$

Using these definitions, an algorithm for interpolation based model checking can be given as the following pseudo code (Algorithm 4.1).

Algorithm 4.1 A modified version of McMillan's procedure FINITERUN

```
def finite_run((State,Init,Trans,Label) as Kripke, Spec, k > 1):
  if (Init ∧ ¬Spec) is SATISFIABLE:
    return TRUE                          # (1)
  R = Init
  while true:
    Kripke' = (State, R, Trans, Label)
    A = pref(Kripke')    #Init and first transition
    B = suff(Kripke', k) #remaining transitions and ¬Spec
    if (A ∪ B) is SATISFIABLE:
      if R = I: return TRUE              # (2)
      else:     return ABORT             # (3)
    else:
      P = interpolant(A,B)
      R' = instantiate(P,"___step___0")
      if R' implies R:
        return FALSE                     # (4)
      R = R or R'
```

In order to understand the pseudo-code[2], I found it enlightening to analyze the return behaviour of `finite_run`. The procedure returns `TRUE` if a bad state is reachable in k steps, `FALSE` if a bad state is not reachable and `ABORT` if k is not sufficiently large to decide, whether or not a bad state is reachable. There are return statements at four points in the control flow, marked by the comments `# (1)` to `# (4)`.

The first two are trivial cases: If $Init \wedge \neg Spec$ are satisfiable, an initial state is also a bad state `(1)`. Analogously, if $A \cup B$ is satisfiable with $R = I$, a counterexample was found during the BMC phase, ergo the corresponding model represents a real path (of length k) to a bad state `(2)`. These are also the cases which plain bounded model checking could decide. `(3)` is the first non-trivial case. $A \cup B$ being satisfiable means that a bad state is reachable from the over-approximation of state s_1 (in $k-1$ further steps), but not necessarily from the real state s_1. In this case, the procedure is run again for $k+1$ steps. In case `(4)`, the freshly computed interpolant implies the

[2]This pseudo code is essentially the same as the procedure FINITERUN in [McM03], it was however adapted in the following ways:

- This variant of the procedure always returns a value, in particular, the value `ABORT` is returned whenever McMillan's procedure aborts
- The syntax has been slightly altered for aesthetic reasons
- There are slight differences in the data structures; also, some redundant computation is avoided in the actual implementation, see section 5.3

previous group A. By definition, the previous group A also implies the interpolant. Thus, both formula sets are equivalent, i.e., a fixed point of the computation has been reached, i.e., we are not getting closer to reaching a bad state, i.e., no bad states will be reachable, no matter how long we iterate or how much we increase k. When such a fixed point is reached, it can be shown that a bad state is not reachable in any number of steps.

4.3 Sketching Workflow

In the previous section I defined the core of my algorithm. In order to actually solve sketching problems, the function must be run iteratively with the right parameters. This behaviour is mirrored in the pseudo code, which consists of a loop in which the outcome of `finite_run` determines what has to be done next.

Algorithm 4.2 Using McMillan's algorithm for sketching problems

```
def complete_sketch((State,Init,Trans,Label) as Kripke, Spec):
  k = 2
  while true:
    result = finite_run(Kripke, Spec, k)
    if(result == TRUE):
      remove the assignments to the oracle variables
      that correspond to the generated counterexample
      if no possibilities remain
        return "All possibilities are incorrect. Try again."
    if(result == FALSE):
      get a correct assignment to the oracle-vars
        (any of the remaining assignments)
      return the completed program
    if(result == ABORT):
      k = k + 1
```

Initially, k is assigned the value 2, as `finite_run` checks the trivial cases of $k = 0$ and $k = 1$ (in which interpolation group B would be empty).

If `finite_run` returns `TRUE`, a bad state can be reached and the SAT solver gives us a counterexample in form of a model which contains assignments to all relevant variables. A corresponding assignment to the oracle variables can be extracted from the counterexample. As this assignment to the oracle variables is provably incorrect, the corresponding choice is removed from the Kripke structure and the algorithm is run again.

As was hinted in the previous section, a return value of `ABORT` indicates that `finite_run` must be called again with a larger k. Thus, k is incremented before the

next iteration.

Finally, if finite_run returns false, the specification cannot be violated anymore. In other words, all remaining assignments for the oracle variables are correct. Because of this, any of the remaining assignments can be chosen to complete the original program sketch according to it and return the resulting program.

5 Implementation

This section contains information about the implementation of the algorithm presented in the previous chapter. section 5.1 and section 5.2 explain how the algorithm works in practice for a simple example program and highlight which tools were used for each individual step. In the former section, the general structure of my program and its environment is explained. In the latter, the specific data structures of each step are shown using example code. In section 5.3, the differences between McMillan's original algorithm and the variant used in my thesis are pointed out. I list the limitations of the current version of my implementation in section 5.4. Finally, the example from section 5.2 is extended to a complete sketching problem and supplemented by a second, more complex example and the implementation's performance is compared to the earlier work by Morgenstern and Schneider in section 5.5.

5.1 General Structure

The actual sketching algorithm was described in section 4.3, this section shows how the algorithm is embedded into a complete path from a program sketch to an assignment to the oracle variables, which can be translated to a correct implementation. Figure 5.1 shows a visualisation of all components that are used by my program. The parts highlighted in gray were (partially) developed by me, the white components are called by my code.

Starting with a program sketch written in Quartz, the Averest framework's qrz2aif compiler is used to translate the program sketch to Averest's intermediate format AIF. This representation is then translated to Averest's Kripke data type[1], using an F# module provided by Andreas Morgenstern. After that, the resulting Kripke structure is transformed by my program (shown in Figure 5.1 as "`F# Code`"), for the actual algorithm, both MathSAT terms and a modified version of Averest's data structure are used.

As was mentioned before, much of the mathematical heavy lifting (transforming logical formulas to CNF, solving SAT problems, computing interpolants) is left to the SAT solver MathSAT [Gri12]. My F# code repeatedly calls the MathSAT API to solve SAT problems which correspond to the model cheking problems that result from the original sketching task. Finally, my program emits an assignment to

[1]This representation already contains a representation of the program's specification

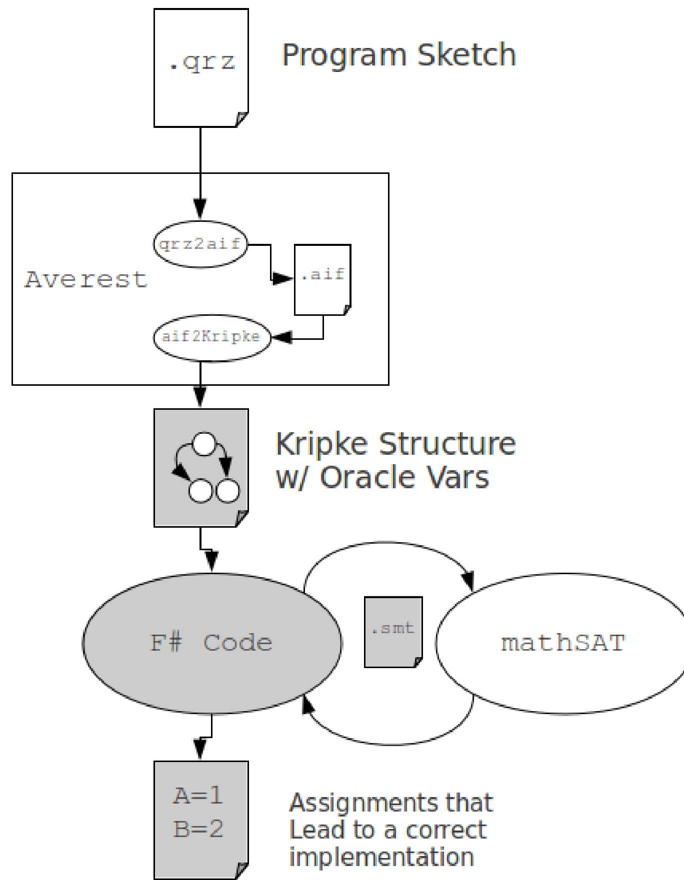

Figure 5.1: The general structure of my program as described in this section

the oracle variables which leads to a correct implementation of the program sketch according to the specification. Actual examples of the different representations used in the program are shown in section 5.2.

5.2 Translating a Program to SMT Formulas

The usage of an external SAT solver required many translations between different data formats. Although these formats are not important to the algorithm in general, this is where much of the development time was spent. This section presents short examples of some of the formats, while relating each format to a step described in section 5.1. The example that is used here is a part of the "Hello World program of sketching" which will be shown in its entirety in section 5.5. It is important to note that the program shown in this section is not actually a program sketch on its own (as the oracle variable mul is just treated as one of the input variables). I have decided to use only a part of the program sketch because the representations of the complete sketch are much too long to be useful as examples.

```
package helloworld;
macro nat_size = 2;
module helloworld_sketch(nat{3} ? mul , nat{nat_size} ? x,
nat{nat_size+1} ! y) {
    y = x * mul;
}
```

Figure 5.2: helloworld_sketch.qrz from [MS11]

Starting with the program shown in Figure 5.2, the first translation to a Kripke structure (which is shown in Figure 5.3) is done by the Averest framework (the AIF representation is not shown). The nat variables from the Quartz program have been transformed to Boolean variables used in the Kripke structure's formulas. The program in general has been translated to several logical formulas, which correspond to the formulas mentioned in section 3.1. However, in addition to the formulas for the `initial states` (called *Init* in the definition of Kripke structures) and the `transition relation` (*trans*), a new formula set called `invariants` is introduced. This formulas assert facts that hold in every state of the Kripke structure, thus removing redundancy between the other formula sets[2]. Additionally, this representation uses additional `definitions` to further shorten the formulas (e.g. the definition `__bit_invar002` is used as an abbreviation for (`!mul§0§0 & !mul§0§1`)).

These formulas are represented as a tree-like data structure which is directly equivalent to the more human-readable formulas shown here.

In order to use MathSAT to solve SAT problems over these formulas, they have to be translated to MathSAT's internal data structures. This is done by calling the public functions of MathSAT's API. The solutions returned by MathSAT can then be retranslated to Averest/F# data structures by calling more API functions, the only way to obtain a human-readable form from MathSAT directly is to convert the formulas to SMT-LIB expressions [BST10]. The SMT-LIB formulas for `transition relation` $\wedge$ `invariants` are shown in Figure 5.4. These formulas have been cleaned up to improve readability, MathSAT actually returns formulas consisting of consecutive definitions of abbreviations, the last of which is then used in one very short assert statement[3]. In the present form, it is possible to see how these formulas were constructed from the corresponding terms in Figure 5.3. Note that the formulas have been instantiated for `step__1`, as was explained in section 3.1. Because of this, most variables bear the suffix `step__1`, except the variables used in a next()-statement, which use the suffix `step__2` instead.

[2] A consequence of this new formula set is that (for example) the transition relation has to be stated as `transition relation` $\wedge$ `invariants`.

[3] These abbreviations have been resubstituted using a small python script, the resulting expression was then pretty-printed using emacs's Lisp pretty printer (which was possible because SMT-LIB's syntax is "a sublanguage of Common Lisp's S-expressions" [BST10])

```
kripke structure helloworld_sketch_helloworld_sketch:
    declarations:
            mul§0§0: oracle bool
            mul§0§1: oracle bool
            x§0§0: input memorized bool
            y§0§0: output memorized bool
            y§0§1: output memorized bool
            __init000: local memorized bool
            carrier'y§0§0: local memorized bool
            carrier'y§0§1: local memorized bool
    definitions:
        __bit000 : mul§0§1&x§0§0
        __bit001 : mul§0§0&x§0§0
        __bit_invar000 : mul§0§0
        __bit_invar001 : mul§0§1
        __bit_invar002 : !__bit_invar000|!__bit_invar001
        __bit_invar003 : !x§0§0
        __bit_invar004 : y§0§0
        __bit_invar005 : y§0§1
        __bit_invar006 : !__bit_invar004|!__bit_invar005
    initial states:
      !carrier'y§0§0&__init000&!carrier'y§0§1
   transition relation:
      !next(__init000)&
      (y§0§1<->next(carrier'y§0§1))&
      (y§0§0<->next(carrier'y§0§0))
   invariants:
      __bit_invar002
      __bit_invar006
      y§0§0<->case(__init000: __bit001, default: carrier'y§0§0)
      y§0§1<->case(__init000: __bit000, default: carrier'y§0§1)
   fairness constraints:
   specifications:
```

Figure 5.3: Textual representation of the corresponding Averest Kripke structure as printed by `Averest.Analysis.Kripke.ToString()`

5.3 Modifications To McMillan's Algorithm

Although in theory, my program contains a straightforward implementation of McMillan's algorithm, there are several subtle differences to the algorithm in its original form.

The most obvious one is that McMillan does not use Kripke structures and specifications as his model for programs. Instead he uses finite automata, which he defines as a 3-tuple (I,T,F), "where the *initial constraint* I and the *final constraint* F are state predicates, and the *transition constraint* T is a state relation" [McM03]. Fortunately, the translation from these automata to Kripke structures is trivial: I corresponds to $Init$, T to $Trans$ and F to $\neg Spec$[4].

As the MathSAT SMT solver was used for model checking and the computation of interpolants, it was not necessary to transform any logic formula to CNF on my side

[4]Since I am looking for paths to a state where the specification is violated,$\neg Spec$ can be used as a description of the final states.

```
(set-info :source |printed by MathSAT, python and emacs|)
(declare-fun __bit000__step__1 () Bool)
(declare-fun |carrier'y§0§0__step__1| () Bool)
(declare-fun |carrier'y§0§1__step__2| () Bool)
(declare-fun __init000__step__1 () Bool)
(declare-fun __bit_invar002__step__1 () Bool)
(declare-fun |carrier'y§0§0__step__2| () Bool)
(declare-fun __init000__step__2 () Bool)
(declare-fun __bit001__step__1 () Bool)
(declare-fun |carrier'y§0§1__step__1| () Bool)
(declare-fun |y§0§1__step__1| () Bool)
(declare-fun |y§0§0__step__1| () Bool)
(assert
 (and
  (and
   (and __bit_invar006__step__1
        (and __bit_invar002__step__1
             (and (and (not __init000__step__2)
                       (or (and |y§0§1__step__1|
                                |carrier'y§0§1__step__2|)
                           (and (not |y§0§1__step__1|)
                                (not |carrier'y§0§1__step__2|))))
                  (or (and |y§0§0__step__1|
                           |carrier'y§0§0__step__2|)
                      (and (not |y§0§0__step__1|)
                           (not |carrier'y§0§0__step__2|)))))))
   (or (and |y§0§0__step__1|
            (or (and __init000__step__1
                     __bit001__step__1)
                (and |carrier'y§0§0__step__1|
                     (not __init000__step__1))))
       (and (not |y§0§0__step__1|)
            (not (or (and __init000__step__1
                          __bit001__step__1)
                     (and |carrier'y§0§0__step__1|
                          (not __init000__step__1)))))))
  (or (and |y§0§1__step__1|
           (or (and __init000__step__1
                    __bit000__step__1)
               (and |carrier'y§0§1__step__1|
                    (not __init000__step__1))))
      (and (not |y§0§1__step__1|)
           (not (or (and __init000__step__1
                         __bit000__step__1)
                    (and |carrier'y§0§1__step__1|
                         (not __init000__step__1))))))))
```

Figure 5.4: SMT formulas for trans ∧ invar, instantiated for step 0.

of the algorithm[5]. Also, MathSAT's backtrack points as well as references to previously computed MathSAT expressions were used to avoid redundant computation. For example, the interpolation group `B` contains only instances of $Trans$ and $\neg Spec$. Since only $Init$ (or rather, `R`) changes between the iterations of `finite_run`, it is only necessary to construct `B` once.

[5]Although I did translate all logical operators to ∧, ∨, and ¬.

5.4 Limitations

Due to the limited editing time of a Bachelor thesis the implementation of my algorithm has several limitations. First of all, only safety constraints are allowed. In implementation terms this means only formulas starting with `A G`, followed by an Averest `BoolExpr` are allowed.

Additionally, the implementation supports only a subset of Averest's aif syntax. However, Morgenstern's AIF2Kripke module seems to compile most Quartz programs to a similar subset, so it remains unclear whether or not additional suppport is needed.

Also, the starting point for my program are Quartz sketches written by Schneider and Morgenstern in which the oracle variables have to be marked by hand, i.e. my program does not support any syntax for the introduction of oracle variables.

The program's user interface leaves much to be desired as my program does not actually return a completed program. Instead, only a list of assignments to the oracle variables present in the translated Kripke structure is provided, which has to be translated to actual choices in the program. Also, many details concerning the program sketch in question have to be hard-coded in the F# program itself.

Finally, there are several points in the program (especially during the communication between F# and MathSAT) where a more efficient implementation might lead to significant performance improvements.

5.5 Examples and Performance

Since I mentioned that the example in section 5.2 was not an actual program sketch, it remains to be shown how an actual program sketch looks like. For this purpose I will show how the Hello World world program sketch might look to the user and to the synthesizer, respectively. As the Hello World program sketch is not particularly interesting and could probably be solved by brute force, I will give another example (the NIM game) which can be scaled in size.

```
macro N = 2;
module HelloWorld(nat{N} ?x, nat{N*2} !y) {
  y = x * ??;
}
satisfies{
  s1: assert (y == x+x);
}
```

Figure 5.5: The Hello World sketch

Figure 5.5 shows the Hello World program sketch in the syntax also used by Morgenstern and Schneider [MS11]. The `??` token represents the uncertainty of the programmer who in this case was not sure which integer `x` needs to be multiplied with in order to make sure that `y` is always equal to `x+x`.

```
macro N = 2;
module HelloWorldSketch(nat{N} ?x, nat{N*2} !y,nat{3} ?mul,) {
  y = x * mul;
}
module HelloWorldSelector(nat{N} ?x, nat{N*2} !y,nat{3} ?mul,){
  nat{3} m;
  nothing; // here q 0 implicitly holds
  q_sel : pause;
  m = mul;
  q 1:pause;
  HelloWorldSketch(x,y,m);
}
satisfies{
  spec1: assert A G (y == 0 | y == x + x);
}
```

Figure 5.6: The Hello World sketch transformed

After the transformation described in section 3.2, the program has the form shown in Figure 5.6. Here, an oracle variable `mul` was introduced. Additionally, another module was wrapped around the program sketch, which is used to select the value of `mul`. The local variable `m` is used to store the value of `mul`. This is necessary, as the inputs to Quartz modules are more similar to electrical wires than actual variables (which means that their value can always change over time). As a local variable, however, `m` is memorized by default and will store the value that is assigned to it.

Figure 5.7 shows the corresponding Kripke structure. Here it becomes more obvious that the transformation from section 3.2 was actually applied. There is now a common initial state that connects to three sub-structures, each of which symbolizes one choice for mul (manifested in the value of m). In order to keep the Kripke structure small and readable, some states have been combined by replacing "dont-care" variables with a ★.

It becomes clear that only the right sub-structure fulfils the specification (`assert (y == x+x);`). Accordingly, my algorithm finds out that bad states can be reached for `m == 0` and `m == 1`, removes them as possible choices and returns the remaining assignment `m = 2`.

Of course, this example is rather uninteresting. Because of this, one of Schneider/-Morgenstern's other examples was tested: The sketch for a winning strategy for the

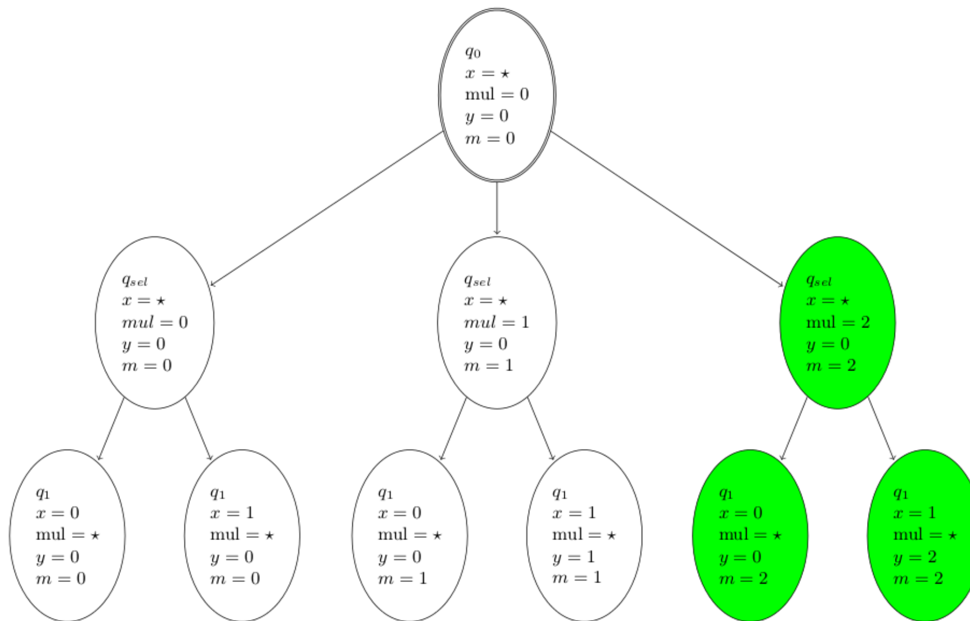

Figure 5.7: Kripke Structure for the Hello World Example[MS11]

single row NIM game, as shown in Figure 5.8. In this game, N matches are placed before two players A and B. The two players then alternate in taking 1, 2 or 3 matches from this pile, starting with player A. The player who takes the last match wins. There exists an optimal strategy: If a player always chooses to take so many matches that the number of remaining matches is a multiple of four, player B will never have the chance to take the last match. This is obviously possible for player A, if the number of initial matches is not a multiple of four. If the initial number of matches is a multiple of four, player B has the winning strategy.

The sketch assumes that the initial number of matches is 21, i.e. A could always win, if the player remembers the strategy. The programmer remembered that the remaining number of matches modulo `x` must always be zero after A's turn. However, he does not remember the exact value of the modulus `x`. Because of this, an oracle variable modulus is introduced. It is obvious that the modulus should be greater than zero, which has been added to the programs specification.

Unfortunately, there was not enough time for proper performance testing. However, for the two examples tested, performance seems to be similar to the results of Schneider and Morgenstern. It is difficult to directly compare the two approaches, since sketching using my current implementation includes the startup time of mono (which luckily seems to be dominating the runtime for the Hello World example) as well as communication between C and F# code.

```
package SingleRowNIM;
macro N = 21;
module SingleRowNIMSketch(
     nat{N} numA, numB,
     nat{N+1} matches,
     nat{5} modulus,
     bool turnA
) {
     turnA = true;
     matches = N;
     while(matches>0) {
          if(turnA) {
               numA = (matches % modulus);
               next(matches) = matches-numA;
               next(turnA) = not turnA;
          } else {
               if(0<numB & numB<=matches) {
                    next(matches) = matches-numB;
                    next(turnA) = not turnA;
               }
          }
          pause;
     }
}
module SingleRowNIM(nat{N} numA, numB,        nat{N+1} matches,
nat{5} modulus,        bool turnA) {
     nat{5} modu;
     pause;
     modu=modulus;
     SingleRowNIMSketch(numA,numB,matches,modu,turnA);
}
satisfies {
   A_always_wins: assert A G (modu > 0 & (matches==0 -> !turnA));
}
```

Figure 5.8: A sketch for the Single Row NIM Game

6 Conclusion

I have shown that McMillan's procedure for model checking and Morgenstern/Schneider's approach to sketching can be combined. I have derived an algorithm that combines the two procedures and I have developed a proof of concept implementation of said algorithm. The performance of the new algorithm seems reasonable, although a more detailed analysis is necessary to determine if it is actually faster than a BDD-based approach. Unsurprisingly, there is still work left to do. My implementation is somewhat incomplete, especially the support of arbitrary specifications instead of only safety constraints should be added, which would also enable a more extensive analysis of the program's performance. However, these limitations stem from a lack of time, rather than exceptional difficulty.

Future work may also include the extended consideration of the ideas of sketching, e.g., the extension of programming language syntax to allow the programmer to conviniently describe choices left to the synthesizer instead of manually adding oracle variables. This may lead to a more declarative style of programming, where the specification gains more and more importance relative to the actual code that needs to be written by human programmers.

Another field for future research is the improvement of the model checking algorithm that was used. McMillan mentions several options in the section "Optimizations" of his paper. It remains to be tested which of these optimizations prove worthwhile for sketching and which other optimizations may be performed for the specialized sub-problem of sketching instead of general model checking. In particular, I assume that my algorithm can be massively parallelized, due to the anatomy of Morgenstern/Schneider's construction of Kripke structures with oracles.

Bibliography

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press, 1996.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[Cra57] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[Gri12] Alberto Griggio. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *JSAT*, 8:1–27, January 2012.

[Knu98] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

[MS11] A. Morgenstern and K. Schneider. Program sketching via CTL* model checking. In A. Groce and M. Musuvathi, editors, *Model Checking Software (SPIN)*, volume 6823 of *LNCS*, pages 126–143, Snowbird, Utah, USA, 2011. Springer.

[Non12] A. Nonymous. Hand wave. *Television Tropes & Idioms*, May 2012.

[Sch03] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[Sch09] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[Sha09] E. Shade. Size matters: lessons from a broken binary search. *Journal of Computing Sciences in Colleges (JCSC)*, 24(5):175–182, May 2009.