Synthesis
without Syntactic Templates

Elizabeth Polgreen
Linacre College
University of Oxford

A dissertation submitted for the degree of
Doctor of Philosophy

Trinity 2019

Abstract

Program synthesis is the mechanised construction of software. One of the main difficulties is the efficient exploration of the very large solution space, and tools often require a user-provided syntactic restriction of the search space. Writing such templates is often challenging, particularly since any constants required in the solution must be given in the template. In this dissertation we explore the research hypothesis that synthesis of programs without provision of such syntactic templates is computationally feasible using methods based on CounterExample Guided Inductive Synthesis (CEGIS).

The key contribution of this dissertation is a new approach to program synthesis that combines the strengths of a counterexample-guided inductive synthesizer with those of a theory solver, exploring the solution space more efficiently without relying on user guidance. We call this approach CEGIS($\mathcal{T}$), where $\mathcal{T}$ is a first-order theory. Notably, since the publication of CEGIS($\mathcal{T}$), a variant of the algorithm has been implemented inside leading synthesis solver CVC4. CEGIS(T) is as complete (or incomplete) as CEGIS, and it does not affect worst-case run-time complexity considerations.

In combination with the development of CEGIS($\mathcal{T}$), we introduced several algorithmic improvements that enhance both the performance of CEGIS($\mathcal{T}$) and the baseline CEGIS algorithm. The first of these improvements is a significant change to the way we encode the synthesis problem for the SAT-based synthesis component of CEGIS in order to produce smaller formulae that can be more efficiently solved by SAT and SMT solvers. The second of these improvements is the use of incremental satisfiability solving in the synthesis component of CEGIS. Incremental satisfiability solving allows the SAT solver to re-use clauses learnt in previous CEGIS iterations, thus reducing the time per synthesis iteration.

We evaluate our contributions on a set of benchmarks taken from the Syntax Guided Synthesis Competition, and note that CEGIS($\mathcal{T}$) is able to solve benchmarks which elude a standard implementation of CEGIS, specifically benchmarks that contain non-trivial constants. We find that the novel proposed encoding provides a substantial speed-up on all the benchmarks that require synthesising a program of length greater than one instruction. The use of incremental first-order solving in CEGIS decreases the solving time in some cases but not all, and we hypothesise that these are the benchmarks on which SAT-solver pre-processing is less important.

Contents

List of Figures

Chapter 1

Introduction

Program synthesis is the task of automatically generating executable code that meets some user-given specification. The promise of program synthesis is to alleviate the burden on the engineer to devise, design and debug software code that meets the desired characteristics. The program synthesis problem has been considered in theoretical computer science since the 1950s. Alonzo Church defined *the synthesis problem* as the problem of synthesising a circuit from mathematical requirements:

> *"Given a requirement $S(t)$ in a logical system which is an extension of recursive arithmetic, to find (if possible) recursion equivalences for a circuit which satisfies the requirement". [51]*

Church used this definition to obtain complexity results [29]. However, the prospect of program synthesis algorithms becoming useful in software engineering is still a vision that is yet to be realised. Nevertheless we have seen measurable improvements in the efficiency and scalability of algorithms for synthesising programs in restricted special cases.

A driver for this algorithmic progress has been the availability of a standardised exchange format for program specifications. This format, referred to as SyGuS-IF, was introduced in 2013 by Rajeev Alur et al. [8] and has enabled an annual synthesis competition to be held. Since the inaugural competition in 2014 the performance of the solvers has improved both in terms of the number of benchmarks that can be solved and the speed of solving. The benchmarks are divided into the following classes:

- A general track, which comprises of benchmarks that require synthesising functions that: manipulate arrays; manipulate bitvectors; solve problems from the book *Hacker's Delight* [76]; use complex branching structures over arithmetic expressions; simplify complex expressions; and control robot movement in motion planning benchmarks.

- An invariant generation track [1, 54], in which solvers must generate loop invariants over bitvectors or integers.
- A programming by example track, in which solvers must synthesise programs that satisfy a set of input-output examples.

Early applications of program synthesis engines identifiable in the literature include data wrangling [46], i.e., transforming, preparing and formatting raw data to produce a more structured data for further use, and string/data transformations, which are often performed by non-programmers. The best example of this application is Flashfill [60], shipped with Microsoft Excel 2013 and later versions; synthesising programs that construct graphical objects [65]; automatic code repair [33, 74, 102]; code suggestions [133]; and code analysis [35].

In full generality program synthesis is too difficult. From a logical point of view, it requires solving a formula in second-order logic, i.e., a formula that quantifies over relations as well as variables. That is, the formula quantifies over a relation P because it asks if there exists a program P such that a logical specification ϕ is satisfied for all program inputs.

In its original form, the program synthesis problem has been defined for a rich range of modal logics to express the formula ϕ, and in full generality, the program synthesis problem is undecidable. The community has therefore made pragmatic simplifications of the problem formulation in order to devise initial algorithms.

In Sytax-Guided Synthesis the problem is constrained as follows: 1) the logical symbols and their interpretation are restricted to a specific background theory; 2) the property ϕ is restricted to a first-order theory in the same background theory; and 3) P is further restricted to syntactic expressions from a user-specified grammar G. The background theories considered in the syntax guided synthesis competition [10] are limited to Linear Integer Arithmetic (LIA) and bitvectors, and the property ϕ is quantifier free. The specification inherently contains quantifiers as the task is to verify whether there *exists* a program that satisfies a specification *for all* possible inputs. The synthesis problem is thus reduced to a second-order formula with a single quantifier alternation, defined as:

> *Given a background theory T, a typed function symbol P, a formula ϕ over the vocabulary of T along with P, and a set G of expressions over the vocabulary of T and of the same type as P, find an expression $e \in G$ such that the formula $\phi[P/e]$ is valid modulo T, where $\phi[P/e]$ denotes the result of replacing each occurrence of P in ϕ with e [8].*

Richer specification logics, including temporal modalities, have been discussed in the literature but are out of scope of SyGuS-IF. The key simplification made in SyGuS-IF, however, is the expectation that the problem is accompanied by the grammar G, which is referred to as a *syntactic template* in the SyGuS literature. The syntactic template is in the form of a grammar given as a set of production rules. The rules used in many benchmarks are frequently very restrictive and significantly simplify the search for a solution. Early SyGuS-IF solvers have in essence been able to find a solution by enumerating possible formulae permitted by the grammar and checking a (usually quantifier free) first-order formula for each candidate. This simplification of the problem setting has had an impact on today's synthesis algorithms. An instance of this is that the expectation that any constant literal required for the solution is given as part of the problem has resulted in a lack of algorithms that can solve problems that require non-trivial constant literals, i.e., constants that do not feature in the logical specification or in the grammar and are not easily constructed by addition or subtraction of 0 or 1 and the maximum value that the bitvector can represent.

While this pragmatic restriction of the problem has indeed enabled a good number of tools to be designed, it has become apparent that the required grammar is a barrier to adoption of this technology in a range of applications. As an exemplar of why this is a problem, loop invariants that are non-trivial for a human to solve frequently require constant literals that are not present in the program under analysis.

For illustration, consider the very simple program below. It increments a variable x from 0 to 100000 and asserts that its value is less than 100005 on exit from the loop.

```
1 int x=0;
2 while (x<=100000) x++;
3 assert(x<100005);
```

Proving the safety of such a program, i.e., that the assertion at line 3 is not violated in any execution of the program, is a task well-suited for synthesis (the Syntax Guided Synthesis Competition [10] has a track dedicated to synthesising safety invariants). For this example, a safety invariant is $x < 100002$, which holds on entrance to the loop, is inductive with respect to the loop's body, and implies the assertion on exit from the loop.

While it is very easy for a human to deduce this invariant, the need for a non-trivial constant makes it surprisingly difficult for state-of-the-art synthesisers: both CVC4 (version 1.5) [120] and EUSolver (version 2017-06-15) [9], given the above specification without a syntactic template, fail to find a solution in an hour on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM.

Current solvers have gone beyond enumeration of formulae permitted by the grammar and it may now be possible to consider problems in the more general unrestricted form that consist solely of a specification, i.e., omitting the requirement 3) of SyGuS-IF, that the program P is restricted to syntactic expressions from a restricted grammar G. In particular, CVC4 leverages counterexample-guided quantifier instantiation techniques for benchmarks where the program to be synthesised is only invoked once. EUSolver leverages enumeration techniques more effectively than standard enumeration, by separately enumerating smaller expressions that are correct on subsets of inputs, as well as predicates that differentiate between these subsets of inputs. Whilst these techniques have provided a significant speed up on the available benchmarks in the SyGuS-IF framework, they are still left struggling when it comes to synthesising solutions that contain non-trivial constant literals not contained within the syntactic template provided.

In this dissertation we explore the research hypothesis that synthesis of programs without provision of such syntactic templates is computationally feasible using methods based on CounterExample Guided Inductive Synthesis.

In support of this hypothesis we present a novel algorithm for program synthesis that allows the synthesis of programs containing non-trivial literal constants. We name this algorithm CEGIS($\mathcal{T}$), where $\mathcal{T}$ is a first-order theory. CEGIS($\mathcal{T}$) combines the strengths of counterexample-guided inductive synthesis with those of a first-order theory solver, in order to perform a more efficient exploration of the solution space without relying on user guidance. One inspiration for this new method is DPLL($\mathcal{T}$), which has boosted the performance of solvers for many fragments of quantifier-free first-order logic [103]. DPLL($\mathcal{T}$) combines reasoning about the Boolean structure of a formula with reasoning about theory facts to decide satisfiability of a given formula. In a similar way, CEGIS($\mathcal{T}$) combines reasoning about generalized candidate solutions with reasoning about theory facts to provide counterexamples that give guidance on the general structure of candidate solutions. The algorithm is complete in all cases where the CEGIS algorithm would be complete, i.e., where the space if possible programs is finite.

Supporting our hypothesis, our experimental evaluation finds that CEGIS($\mathcal{T}$) is able to synthesise programs that elude conventional solvers. The worst case complexity of CEGIS($\mathcal{T}$) is the same as CEGIS, but in practice the run-time is often much smaller if non-trivial constants are required. Furthermore, the CEGIS($\mathcal{T}$) algorithm has been

subsequently integrated into CVC4 [18] by the CVC4 team, and the results there are even more promising.

In combination with the development of CEGIS($\mathcal{T}$), we introduced several algorithmic improvements that enhance both the performance of CEGIS($\mathcal{T}$) and the baseline CEGIS algorithm. The first of these improvements is a significant change to the way we encode the synthesis problem for the SAT-based synthesis component of CEGIS in order to produce smaller formulae that can be more efficient solved by SAT and SMT solvers. This improvement applies across all benchmarks that require the synthesis of programs longer than a single instruction. The second of these improvements is the use of incremental satisfiability solving in the synthesis component of CEGIS. Incremental satisfiability solving allows the SAT solver to re-use clauses learnt in previous CEGIS iterations, thus reducing the time per synthesis iteration. This incremental solving reduces the synthesis time on some benchmarks but not on others, which is likely because it is necessary to disable or reduce the SAT solver pre-processing when using incremental solving.

Research Hypothesis

The research hypothesis explored by this dissertation is that synthesis of programs containing arbitrary constants is computationally feasible without provision of syntactic templates. We explore the above new algorithmic developments for program synthesis and evaluate these on a broad set of benchmarks without syntactic templates. The performance observed, against state-of-the-art program synthesis engines, suggest that program synthesis without syntactic templates is possible.

Structure of this dissertation

The structure of this dissertation is as follows:

- A full discussion of the related work is found in Chapter 2
- The basic preliminaries required for this dissertation are outlined in Chapter 1.3,
- Chapter 4 presents CEGIS($\mathcal{T}$)
- Chapter 5 details the improvements to the synthesis encoding
- Chapter 6 presents incremental CEGIS
- Finally, conclusions and a summary of the results obtained in the dissertation is found in Chapter 7.

1.1 Contributions

In summary, the contributions made in this dissertation are as follows:

- CEGIS($\mathcal{T}$): the key contribution of this dissertation is a novel algorithm for program synthesis, based on combining SAT-based counterexample-guided inductive synthesis with a first-order theory solver. We present CEGIS($\mathcal{T}$), which is an extension to CEGIS, in the same way that DPLL($\mathcal{T}$) is an extension of DPLL. CEGIS($\mathcal{T}$) abstracts the synthesis problem to a higher level, in the same way that DPLL($\mathcal{T}$) initially constructs a Boolean template from a first-order problem; and then leverage a first-order theory solver to provide more detailed counterexamples to the synthesis block of CEGIS. The counterexamples detail whether the abstracted template is the correct one, in a similar way to the use of first-order solver in DPLL($\mathcal{T}$). This work is described in Chapter 4.

- A further contribution of this dissertation is a new efficient encoding for the synthesis problem. This is shown to reduce the size of the SAT formulae being solved by the synthesis block by X and to speed up total synthesis time by X in standard CEGIS. The synthesis encoding is described in Chapter 5.

- The introduction of the efficient encoding also enables a further contribution; incremental CEGIS. Incremental sat soving is a technique whereby, in order to solve a sequence of similar problems, a SAT solver can re-use clauses learnt whilst solving previous problems to speed up solving of subsequent problems. The nature of the CEGIS algorithm is such that the synthesis block solves multiple similar SAT problems, e.g., at each iteration it is searching for a candidate program that satisfies one more input than the previous candidate program. As such, use of incremental SAT is a natural fit to explore in this scenario. Our use of incremental SAT and the evaluation of it is described in Chapter 6.

- In addition, we contribute a set of benchmarks based on using program synthesis to synthesise safe digital controllers for Linear Time Invariant systems. We use these benchmarks, in addition to benchmarks taken from the Syntax-Guided Synthesis competition [11] and benchmarks taken from program analysis [35], to evaluate the benefit of our algorithmic contributions. Full detail of this work is given in Chapter 3.

1.2 Publications

The majority of the work presented in this dissertation is contained in the following papers. Discussion of my contributions to the papers below is included at the beginning of the chapters in this dissertation in which the material is included and in Appendix B:

- Counterexample Guided Inductive Synthesis Modulo Theories [5], CAV 2018 – with Alessandro Abate, Cristina David, Pascal Kesseli and Daniel Kroening. See Chapter 4.

The following papers encompass use of program synthesis to synthesise safe digital controllers, and the controller synthesis benchmarks we use for evaluation in this dissertation are taken from these papers:

- Automated Formal Synthesis of Digital Controllers for State-Space Physical Plants [4], CAV 2017 – with Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli and Daniel Kroening. See Chapter 3.

- DSSynth: an automated digital controller synthesis tool for physical plants [2], ASE 2017 – Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli and Daniel Kroening. See Chapter 3.

Finally, during my PhD I worked on the following papers outside of the scope of this dissertation. In these papers I was the lead author, and lead the development of the algorithms, the experimental work and the writing:

- Automated Experiment Design for Data-Efficient Verification of Parametric Markov Decision Processes [113], QEST 2017 – with Viraj Wijesuriya, Sofie Haesaert and Alessandro Abate

- Data-Efficient Bayesian Verification of Parametric Markov Chains [112], QEST 2016 – with Viraj Wijesuriya, Sofie Haesaert and Alessandro Abate

1.3 Preliminaries

In this section we introduce preliminaries and background knowledge used throughout this dissertation. We formally define the program synthesis problem, and give the definition of Syntax Guided Synthesis, a common program synthesis paradigm which uses restrictions on the program synthesis problem in the form of a syntactic template

that restricts the language the program can be constructed from. We introduce Counterexample Guided Inductive Synthesis, a family of algorithms used to solve the program synthesis problem, and on which we build the algorithmic developments in this dissertation.

1.3.1 Program synthesis

Program synthesis is, informally, the task of automatically generating programs that satisfy a given logical specification. A program synthesiser can be viewed as a solver for existential second-order logic. An existential second-order logic formula allows quantification over functions as well as ground terms [124].

The existential second-order logic formula to be solved by a program synthesiser is of the form

$$\exists P. \forall \vec{x}. \sigma(\vec{x}, P) \tag{1.1}$$

where P ranges over functions (where a function is represented by the program computing it), $\vec{x}$ ranges over ground terms, and σ is a quantifier-free formula. We interpret the ground terms over some finite domain, and we use $\mathcal{I}$ to denote the set of all possible inputs to the function, and so in the formula above $\vec{x} \in \mathcal{I}$. In the context of bitvectors, $\mathcal{I}$ is the full range of numbers that can be represented by the bitvector.

P is a typed function symbol. In this dissertation, we consider programs that operate over the first-order theory of bitvectors or linear integer arithmetic. In the case of bitvectors, P may be a function that takes as input a finite number of bitvector variables and returns a bitvector, or it may be a predicate that takes as input a finite number of bitvector variables and returns a Boolean.

1.3.2 Syntax-guided synthesis

Syntax-guided synthesis is a formulation of the classic program synthesis program described previously, where the solver is, in addition to the logical specification, given a grammar, known as a syntactic template, from which to construct the program. This context-free grammar G defines a set of functions L. Each function in L has the same type as that of P, the program to be specified.

Given the specification σ and the set of L candidates, the program synthesis now becomes to find an expression $e \in L$ such that if we use e as the implementation of P, the specification σ is satisfied. Denote the result of replacing each occurrence of P in σ with $\sigma[P/e]$.

More formally, given a theory T, a typed function symbol P, a specifiation formula σ over the vocabulary of T along with P, and a set of L expressions over the vocabulary of T and of the same type as P, find an expression $e \in L$ such that the formula $\sigma[P/e]$ is valid modulo T.

1.3.2.1 Theories

Program synthesis competitions such as the SyGuS competition provide benchmarks over two theories: the theory of fixed size bitvectors and the theory of linear integer arithmetic. The theory of bitvectors is a convenient theory to focus on, as it aligns very well with the semantics of the C programming language. Computers have finite memory, and so they are not able to perform calculations on true mathematical integers, and programming languages use fixed-width bitvectors as a replacement for true integers. The techniques presented in this dissertation are not, however, limited to bitvectors.

Theory of fixed-size bitvectors A bitvector b is a vector of bits with a fixed length l. Each bit takes the value 0 or 1. Bitvectors are used for encoding information in computer systems, and integers in C programs are encoded as bitvectors. Reasoning about bitvectors is thus highly relevant for program synthesis and program analysis, where the semantics of operations like addition and multiplication will no longer match that of true integers, owing to concepts like arithmetic overflow. A bitvector has a finite width, and a bitvector that is l-bits wide is able to take one of 2^l bit patterns, which represent one of 2^l possible integer values.

$$b : 0, \ldots, l-1 \rightarrow 0, 1.$$

In this dissertation we consider a subset of bitvector arithmetic, defined by the following grammar:

```
formula :: formula ∧ formula | ¬formula | (formula) | atom
   atom :: term rel term | Boolean-Identifier
    rel :: < | = | ≤
   term :: term op term | ∼term | constant |
                        atom?term:term | identifier
     op :: + | - | . | / | << | >> | & | | | ⊗
```

Further operators not included in the above list can be derived from the operators included, for example $\geq$ can be made from $\neg$ and $<$.

The semantics of fixed-size bitvectors differs from the semantics of true mathematical integers in a couple of key ways:

- Integer Overflow: bitvectors are bounded. If an operation is performed on a bitvector that results in a value greater than or smaller than that which can be stored in the bitvector, the result will "wrap around". That is, the least significant bits of the result will be stored. This is identical to the behaviour of integer overflow in C programs and verifying these semantics accurately is often critical to proving safety of C code.

- Bit shifts: bitvectors can be shifted; if the bitvector is shifted by a value greater than the width of the bitvector, the result is zero.

- Division by zero: if a bitvector is divided by zero, then SMT-lib defines the result to be the maximum value that can be represented by the bitvector, e.g., for a bitvector of width 8 the maximum value is 11111111 or 255. Division by zero is undefined in the C standard.

- Bitvectors can be interpreted as signed or unsigned values, where the first bit indicates if the value is signed or unsigned, for example the bitvector 11110000 would be interpreted as -16 if the bitvector is signed, or 240 if the bitvector is unsigned. This effects the result of mathematical operations performed on the bitvectors. for example

Theory of Linear Integer Arithmetic Similarly, the syntax of a formula in linear integer arithmetic is defined by the following rules:

```
formula :: formula ∧ formula | ¬formula | (formula) | atom
   atom :: sum rel sum
    rel :: < | = | ≤
    sum :: term | sum + term
   term :: identifier |  constant
```

Linear arithmetic disallows multiplication and division operators. The minus operator is unnecessary as integers can take negative values and so $x - y$ is equivalent to $x + -1y$. By similar argument, $\geq$ and $>$ are also unnecessary.

1.3.3 The CEGIS framework

The CEGIS framework is illustrated in Figure 1.1. It consists of two phases; the synthesis phase and the verification phase: Given the previously mentioned specification of the desired program, σ, the inductive synthesis procedure produces a candidate program P that satisfies the specification $\sigma(x, P)$ for all x in a set of inputs $\mathcal{I}_\mathcal{G}$ (which is a subset of $\mathcal{I}$). The candidate program P is passed to the verification phase, which

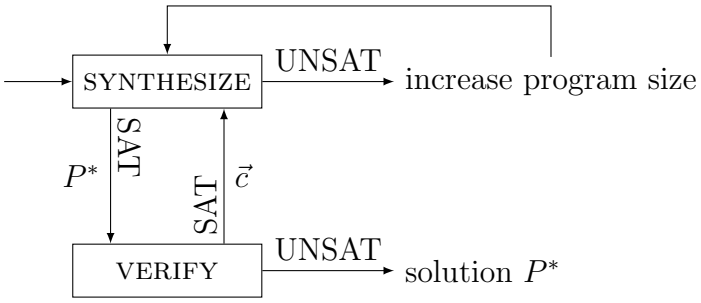


Figure 1.1: The CEGIS framework

checks whether P is a full solution, i.e., it satisfies the specification $\sigma(x, P)$ for all $x \in \mathcal{I}$. If so, we have successfully synthesised a full solution and the algorithm terminates. Otherwise, the verifier produces a counterexample that is added to the set of inputs $\mathcal{I}_\mathcal{G}$ passed to the synthesiser, and the loop repeats. The length of the program the synthesiser is searching for is restricted to some program size L, which is the number of instructions or operators in the program. If the synthesiser is unable to find a solution, there is no candidate program with size $\leq L$ that satisfies the current set of inputs, and we increase the program size. The default behaviour of our implementation is that the algorithm is initialized with a program size of 1 and the size is increased by one instruction any time the synthesiser is unable to find a solution. This means that we will automatically find the shortest program that satisfies the specification. There may be more than one program of a given length that satisfies the specification, in which case we are guaranteed to find one of these shortest programs.

The CEGIS framework encompasses a family of algorithms, as the method used by the synthesis and verification blocks may vary in different CEGIS implementations, effectively varying the way the space of candidate programs is being searched. Popular strategies include the following (and implementations may run different strategies in parallel):

- Exhaustive search – whilst potentially time consuming, the simplest method for finding solutions is to enumerate them all.
- SAT solver based methods – build a Boolean formula that is satisfied if there exists a candidate solution
- Genetic programming (GP) [84] – a population of random programs is generated, and then evolved by mutating programs and combining them together. Programs are selected according to some fitness function that gives a value for how "good"

a program is. The simplest fitness function is counting the number of inputs for which the program satsfies the specification.

In this dissertation we use a SAT or SMT solver based verification and synthesis phase for our basic CEGIS implementation.

1.3.4 Satisfiability (SAT) solvers

Our implementation of CEGIS uses a SAT solver in its verification and synthesis phases. Given a Boolean formula F, a SAT solver decides whether F is satisfiable or not, and if it is returns a satisfying assignment. Satisfiability checking of Boolean formula is NP-complete, i.e., there is no known algorithm which efficiently determines satisfiability of every possible Boolean formula. Nevertheless, modern SAT solvers can check satisfiability of Boolean formulae with millions of variables, which is sufficient for many practical SAT problems, and the performance of SAT solvers has made huge improvements over the last two decades [82].

SAT solvers require a formula F to be in **Conjunctive Normal Form (CNF)**. A formula is in CNF if it is a conjunction of clauses, where each clause, c, is a disjunction of literals (Boolean variables), i.e., it has the form

$$\bigwedge_i (\bigvee_j l_{ij})$$

Any Boolean formulae can be converted into an equivalent CNF formula but potentially increasing the size of the formulae exponentially. We can transform a formula into an *equisatisfiable* formula in CNF in linear time and with only a linear increase in the size of the formula using Tseitin's transformation [146], but with the cost of introducing a new variable for every logic gate in the formula. Two formulae are *equisatisfiable* if, for the same assignment of variables, they are both satisfiable or both unsatisfiable.

Chapter 2

Literature review

This chapter gives a brief overview of existing work in the areas of program synthesis relevant to our work. Some of the text in this chapter is taken from papers published at CAV in 2017 [4] and 2018 [5]. The novelty of our work in comparison to the related literature is covered at the beginning of each chapter describing our work, and is not discussed in this literature review chapter.

2.1 Program Synthesis

Program synthesis techniques can be grouped by three aspects of their implementation [59]:

- Input language – the logic and syntax by which the user gives the specification that the synthesised program must satisfy
- Search space – the space of possible solutions should also be in some way specified by the user
- Synthesis algorithm – the method by which we find the solution that satisfies the specification from within the search space.

Input language options are usually either formal specifications (e.g., expressed in temporal logics) or simple input-output pairs (programming by example). In the latter, examples must be provided by the user either in advance or through user interaction. Effectively the user is teaching the computer to write the program in a similar way to the way an owner may train a dog. Program synthesis tools that use natural language as the CEGIS.user input [39, 66] exist, and are potentially more popular among users but natural language lacks the precision of formal specifications [153].

Traditionally methods for synthesis using input-output examples and synthesis using formal specifications are disjoint areas of research. In this dissertation we focus our algorithmic developments on CounterExample Guided Inductive Synthesis (CEGIS). CEGIS can be thought of as using a logical specification and converting that to examples; a formal specification is given, candidate solutions are generated for that formal specification that hold for a subset of possible inputs, and then verified across the entire set of inputs. If the verification fails, it produces a counterexample in the form of an input for which the specification fails. This counterexample is effectively an input-output pair, where the output is "specification failed", and is returned to the learning block and the cycle repeats. The learning block is in effect performing program synthesis using input-output pairs. For this reason, in this literature review we will also cover program synthesis techniques that use input-output examples.

The search space can be specified in a variety of ways, but ultimately the program synthesiser must be given some set of possible program structures. This can be done by giving a list of operators the program can use, the number of instructions in the program, the control structure (synthesised programs are usually required to be loop-free). The original application of program synthesis via CEGIS is program sketching [134], where a nearly complete program is given with missing parameters. There is also much work based on learning various different grammars, such as automata or regular expressions.

In the following sections of this literature review, we group program synthesis techniques by their search technique. We identify the following categories of search techniques:

- Enumerative search
- Version space algebra
- Machine learning, or stochastic search
- Constraint based search

Program synthesis is a broad area and this overview is not by any means exhaustive, but we have selected those areas of program synthesis that are appropriate to compare with CEGIS.

2.2 Enumerative search synthesis

One of the most obvious ways of finding a program that satisfies the specification would be to go through all of the possible solutions in the solution space in some order and check whether each one satisfies the property, until a satisfying solution is found. We call this exhaustive search synthesis. In general, basic enumerative search synthesis is not successful as it may involve enumerating all of a possibly massive search space. This technique has, however, been successfully used in some applications of program synthesis, mostly where the search space is small and the verification procedure (checking whether a candidate solution satisfies the specification) is quick.

Brute-force exhaustive synthesis can be used to discover mutual exclusion algorithms [17], i.e., algorithms that are deadlock free and where no two processes are in their critical state at the same time. The search space in this paper [17] was limited to a restricted programming language, and user-specified parameters such as the number of lines of code, the number of processes, the number, type and size of shared variables and the type of conditions (e.g., *if*, *while*). The examples in the paper are short programs with up to 6 shared bits, but the approach still requires the mechanical verification of hundreds of millions of incorrect algorithms before a correct algorithm is found, even when these are limited to very short algorithms.

Brute-force search has also been used successfully for optimisation of bitvector programs [58, 93]. Systematic search methods are also used to find small functional programs in the form of lambda expressions [78]. This is an extension of [77], which uses combinatory expressions instead of lambda functions. The authors note that exhaustive search methods are unlikely to be applicable to synthesis of large programs, though the implementation compares favourably to PolyGP [151] for small examples. QuickSpec [30] guesses algebraic specifications for sets of functions. This paper uses testing to determine whether a specification is correct or not, instead of SAT solver based verification, and so the verification time is quick if the number of possible inputs is small.

In conclusion, using brute-force search over the search space of all possible artefacts works in some cases, where the search space is relatively small, and the verification procedure is very fast. In addition, most brute-force search methods use some optimisation techniques to prune the search space.

Whilst enumerative search by itself may be slow and not scalable, it can be combined in parallel with symbolic bounded model checking and genetic programming to make up the synthesis block of CEGIS [35]. Work on improving the scalability of

enumerative techniques by divide and conquer [12] has had promising results in the Syntax Guided Synthesis competition.

2.3 Version space algebra

In version space algebra, a hypothesis h is a function that takes an input and produces an output. A hypothesis is consistent with a training example input $i \in \mathcal{I}$ and output o if $h(i) = o$. A version space consists of a set of hypotheses that are consistent with the input and output examples observed. When new examples are observed, the version space is updated to maintain the consistency. Version spaces were originally introduced in [96] for binary classification, i.e., learning Boolean functions.

Version space algebra allows composition of version spaces, e.g., unions and transforms, to build more complex classifications. It was proposed by [85, 86] for learning programs from traces. The input used is a user demonstration of correct behaviour, similar to the macro recorders in many applications today. However, typical macro recorders merely capture key strokes and play them back. In this work they use version space algebra to build up a search space of functions composed of smaller, simpler version spaces, e.g., moving the cursor to a new row and column position can be decomposed into a function that moves to a new row and a function that moves to a new column. This allows generalisation from the recorded key strokes.

The most notable use of this technique is FlashFill [60], a tool for string processing in spreadsheets, which was first shipped with Microsoft Excel 2013. This technique is also used in combination with type information, for synthesis of recursive functions [105]. Version space algebra based synthesis learns from examples, and so is a potential candidate for the learning block of CEGIS, which, whilst it works with logical specifications, does use a set of input counterexamples in the synthesis phase. Version space algebra by itself synthesises programs from examples, rather than from formal specifications.

2.4 Machine learning

2.4.1 Probabilistic Inference

Probabilistic inference decides the probability of random variable(s) taking a specific value, or set of values. Probabilistic inference algorithms developed in the machine learning community are commonly used for reasoning over graphical models. For example, in inference-based synthesis, it is common to model a program as a graph of

instructions and states connected by constraint nodes and to perform the inference over this graph.

Probabilistic inference is used by [64] to synthesise proofs of correctness of programs, and by [75] to synthesise small programs for given input-output pairs.

Another way of representing programs is to use program trees, which are similar to decision trees where each node represents a variable. Bayesian program synthesis [90] applies Bayesian optimisation over program trees.

Probabilistic inference is used for solving discrete and continuous state Markov Decision Processes [145], i.e., synthesising satisfying policies, and for planning tasks in partially observable Markov Decision Processes [125], i.e., for online synthesis of strategies.

It also has applications in control, where probabilistic inference is used as a data-efficient model-based search method for obtaining policies for control [38].

Probabilistic inference is useful for synthesis in cases where we wish to synthesise a program from data, rather than specifications, in particular if the data is in some way noisy. However, it cannot synthesise artefacts *guaranteed* to satisfy specified properties. Probabilistic inference is a candidate for being used as the learning block of CEGIS.

2.4.2 Genetic algorithms

In genetic algorithms, computer programs are encoded as a set of genes that are then modified using a genetic algorithm. The algorithm maintains a population of individual programs, which it evolves through mutations and crossovers, producing random changes. The programs are evaluated against some kind of user-defined fitness function.

Genetic programming has been applied to program synthesis, where it is used to fix bugs in imperative programs [27], to discover mutual exclusion algorithms [79], to synthesise feature models [88], and for automated program repair [149, 155]. Genetic algorithms have also been applied to the synthesis of strategies for Markov Decision Processes [56].

Synthesis of digital circuits [68, 98, 128] and digital hardware [144] is also achieved using Genetic Algorithms.

Evolutionary algorithms have even been used to tackle the SAT problem [57], and results at the time found genetic programming to be competitive with heuristic algorithms for SAT like WSAT [129], with some adaptations.

CEGIS has been implemented using genetic programming as a learning method within the CEGIS loop [35]. Genetic algorithms in general cannot synthesise artefacts that formally satisfy a specification without this loop.

2.5 Constraint-based synthesis

SAT-based, or logical, synthesis takes advantage of the success of Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solving. The synthesis problem is broken down into two stages: first logical constraints must be generated from the synthesis problem; and then the constraints must be converted into something that can be solved with an off-the-shelf SAT/SMT solver.

Constraints can be generated by various methods. Firstly invariant-based constraint generation generates a synthesis constraint that asserts that a program should behave correctly for all inputs. It amounts to specifying an inductive invariant that implies the specification. Then a program is synthesised for which the inductive invariant holds, and therefore for which the specification holds. This method is based on proof-based deductive synthesis [92], and has been applied to synthesis of Bresenham's line drawing algorithm [139]. Invariant-based constraints guarantee that the program behaves correctly for all inputs and so do not need an external verifier. However, invariant-based constraints are usually more complex and require quantifiers.

An alternative is to use path-based constraints, which assert that the program should behave correctly on all inputs that take a given set of paths. The constraints are simpler than invariant-based constraints but do not guarantee that the generated program always meets the desired specification and so requires an external verifier. This approach has been used for semi-automated program inversion [138].

The final constraint generation method we will mention is input-based constraints. These assert that an unknown program should behave correctly on a given set of inputs. This is used when the input language is a set of input and output examples. The constraints are simpler to solve than path-based constraints, but do not guarantee that the program behaves correctly for all the inputs and so an external verifier is required.

Constraints are solved using standard SAT/SMT solvers. However, a certain amount of preprocessing is required first as the constraints generated above are second-order logical formulae with universal quantification. We can either eliminate the universal quantification or the second-order unknowns.

Second-order unknowns are objects such as invariants, conditional guards, or assignment statements. In order to reduce the second-order unknowns to first-order unknowns we use some template structure, which specifies the structure for the second-order unknowns and leaves only first-order gaps to be filled in. A disadvantage of this method is the need to know a fairly detailed template for the program we are synthesising.

Alternatively, we may eliminate the universal quantifiers. We can translate universal quantifiers into existential quantifiers for some specific domains, e.g., Farkas lemma can be used to transform universal quantification to existential quantification for linear arithmetic [67]. However, a more general approach is to use sampling based-methods. Instead of trying to synthesise a program that satisfies the specification for all inputs in one go, we synthesise a program that satisfies the specification for only a subset of the possible inputs and then ask a verification procedure whether it satisfies the specification for all of the inputs.

This is the technique used in counterexample guided synthesis (CEGIS). CEGIS was first proposed in the seminal paper on program sketching [134]. In program sketching a template for the program is given by the user, which is effectively a program with holes that must be filled in a way such that the program meets the specification. A program is generated that satisfies the specification for a subset of inputs to the program; then a verifier is asked if the specification holds over all inputs; if the answer is no, a counterexample is returned in the form of an input for which it does not hold; the input is added to the subset of inputs given to the synthesiser and the loop repeats. This technique has become increasingly popular in program synthesis [8, 35, 61, 62, 71]. The advantages of this approach are that we can use simple input/output based constraints, and yet still synthesise a program that satisfies the full set of inputs, and that the use of an external verifier means that the synthesis engine does not need to be sound, and can work on some abstraction or approximation.

2.6 Synthesis of digital controllers

In Chapter 3, we present a new set of benchmarks which use program synthesis to synthesised digital controllers for Linear Time Invariant systems. These benchmarks are interesting for evaluation of the algorithms introduced in this thesis because they require synthesis of non-trivial constants, but do not require synthesis of complex expressions.

Program synthesisers are an ideal fit for the synthesis of digital controllers, because the semantics of programs allow us to precisely capture the behaviour of finite precision arithmetic. Control literature has, in general, not exploited this connection. An exception is [117,118], who employ CEGIS for the synthesis of switching controllers for stabilizing continuous-time plants with polynomial dynamics. The work synthesises controllers that satisfy "reach while stay" properties, that is, the system states must reach a given safe set and then remain within that set. The method presented in the paper uses CEGIS to synthesise Lyapunov functions, a function which bounds the trajectory of the system. This method is limited by the capacity of the state-of-the-art SMT solvers for solving non-linear arithmetic. Since this approach uses switching actions for the digital controller, it avoids problems related to finite precision arithmetic, but potentially suffers from state-space explosion. The methods used in these papers are unsuitable for the kind of controllers that we wish to synthesise, which do not use switching.

The work in [3] synthesises stabilizing controllers for continuous plants given as transfer functions by exploiting bit-precise verification of software-implemented digital controllers [22]. While this work also uses CEGIS, the approach is restricted to digital controllers for stable closed-loop systems expressed as transfer function models. Verifying that a controller stabilises such a system can be done with a simple check on the controller coefficients. In contrast, the benchmarks we have produced require synthesis of a controller expressed using a state-space representation of the physical system, with a specification on the safety of the states. This safety specification must be verified to hold over time, and so a static check is not sufficient.

A state-space model has well known advantages over the transfer function representation [50], as it generalizes well to multivariate systems (i.e., with multiple inputs and outputs); and it allows synthesis of controllers with guarantees on the internal dynamics, e.g., *safety*. Our benchmarks specify that controllers must ensure the safety of internal states, which is by and large overlooked in the standard (digital) control literature, by default focussed on stability/regulation/tracking properties.

Beyond CEGIS-based architectures, there is an important line of research on provably correct control synthesis for dynamical models, which leverages formal abstractions. The tool Pessoa [94] synthesises correct-by-design embedded control software in a Matlab toolbox. Pessoa constructs a discrete symbolic abstraction of the system that is ϵ-bisimilar to the original system. Specifications are given in Linear Temporal Logic over this abstraction. The controller is the result of solving automata theoretic games on this abstraction. The embedded controller software can

be more complicated than the state-feedback control we synthesise, and the properties available cover more detail. However, relying on state-space discretization Pessoa is likely to incur in scalability limitations, and this method cannot account for the errors introduced by fixed-point arithmetic. Also along this research line, [13, 89] studies the synthesis of digital controllers for continuous dynamics, and [152] extends the approach to the recent setup of Network Control Systems.

Discretization Effects in Control Design – Recent results in digital control have focused on separate aspects of discretization, e.g. delayed response [40] and finite precision arithmetic, with the goal either to verify [36] the correctness of the implementation or to optimize [106] its design.

There are two different problems that arise from finite precision arithmetic in digital controllers. The first is the error caused by the inability to represent the exact state of the physical system, while the second relates to rounding and saturation errors during mathematical operations. In [49], a stability measure based on the error of the digital dynamics ensures that the deviation introduced by finite precision arithmetic does not lead to instability. [150] uses μ-calculus to synthesise directly a digital controller, so that selected parameters result in stable model dynamics. The analyses in [126, 148] rely on an invariant computation on the discrete dynamics using Semi-Definite Programming (SDP): while the former contribution a uses bounded-input and bounded-output (BIBO) notion of stability, the latter employs Lyapunov-based quadratic invariants. In both cases, the SDP solver uses floating-point arithmetic and soundness is checked by bounding the obtained error. An alternative is [108], where the verification of given control code is performed against a known model by extracting an LTI model of the code via symbolic execution: to account for rounding errors, upper bounds of their values are introduced in the verification phase. The work in [111] introduces invariant sets as a mechanism to bound the quantization error effect on stabilization. Similarly, [87] evaluates the quantization error dynamics and calculates an upper and lower bound for the possible trajectory of the system, up to a finite time. The last two approaches can be placed within the research area known as "hybrid systems theory."

Chapter 3

Benchmarks and Modelling

Our research hypothesis is that synthesis of programs without provision of syntactic templates is possible using methods based on CounterExample Guided Inductive Synthesis. We make four algorithmic contributions towards this hypothesis: a new algorithm titled counterexample guided inductive synthesis modulo theories; a novel encoding of the synthesis problem; a variation of CEGIS using incremental SAT solving; and a counterexample guided neural synthesis algorithm. The hypothesis is experimental in nature and, in this chapter, we present the benchmarks on which we test our hypothesis against these contributions. The chapter is split into two sections: in Section 3.1 we describe the existing benchmarks; and in Section 3.2 we present new benchmarks in the area of controller synthesis. Examples of benchmarks and solutions that we synthesise for them are found in Appendix A.

In the first half of this chapter, Section 3.1, we describe the existing benchmarks available, which we take from the Syntax Guided Synthesis competition in 2018. The syntax guided synthesis competition provides benchmarks with a syntactic template, which is used to render the synthesis problems tractable and provides any arbitrary constants needed by the synthesise program. Thus, these benchmarks with the syntactic template removed, provide us with a useful set of benchmarks against which to test our hypothesis. These benchmarks are split into several categories, across a variety of applications. The benchmarks are available in SyGuS-IF format. We translated some of the Linear Integer Arithmetic benchmarks into bitvectors to expand the collection of bitvector benchmarks. We also use benchmarks from the application area of program analysis [35]. These benchmarks involve finding invariants that prove or refute the safety of C programs, and are naturally originally written in C. We translate these benchmarks into SyGuS-IF in order to allow comparison of our tools with CVC4. The benchmarks from this section are used to evaluate the techniques

presented dissertation and to compare them to state of the art SyGuS solvers. The new work in this section is my translation of the Syntax Guided Synthesis competition benchmarks into C programs and the Linear Integer Arithmetic benchmarks into bitvectors.

In the second half of this chapter, Section 3.2, we introduce a new set of benchmarks, which show how program synthesis can be used to synthesise controllers for Linear Time Invariant systems. These benchmarks require simpler solutions than the SyGuS competition benchmarks, requiring a matrix of constants instead of expressions, but the benchmarks and properties analysed are considerably more complex, as elaborated on further in this chapter. We use these benchmarks as a comparison point for incremental CEGIS and the novel synthesis encoding to enable us to hypothesise about the effect of complexity of the solution required on our algorithmic contributions. Furthermore, given our interest in synthesising expressions containing non-trivial constants, i.e., constants which do not appear in the benchmark, these are benchmarks which are guaranteed to require arbitrary constants in their solution which cannot be read directly from the specification. This work is published at CAV in 2017 [4], and the bulk of the text in this chapter is taken from that paper and from a journal version of this paper submitted to Automatica. A further tool paper was presented at ASE [2]. These benchmarks are available in the C input format only, and we do not translate them into SyGuS-IF format due to the complexity of the benchmarks: namely that they must be configurable for different precisions of bitvector arithmetic, different control systems and different completeness thresholds, and we achieve this via header files in C. We use the benchmarks to evaluate the benefits of incremental satisfiability solving in CEGIS and the effect of the novel encoding.

Contributions My contributions to the paper published at CAV 2017 [4] are as follows:

- Conversion of the general controller synthesis problem into a program synthesis problem for the multi-stage back end.
- Integration of the translated control problem into the multi-stage back end, in close collaboration with Pascal Kesseli.
- Running of experiments for both the multi-stage and acceleration based back end, in close collaboration with Pascal Kesseli.

- Write up of the multi-stage back end in collaboration with Cristina David.
- Formalisation of the correctness argument behind the multi-stage back end, including the proof of stability of closed-loop models with fixed-point controller error.

My contributions to the tool paper presented at ASE are limited to my work on the implementation and experimentation for the CAV 2017 paper, as described above.

In the final section of this chapter, we summarise the features of our set of benchmarks and describe the experimental set up used throughout the dissertation.

3.1 Existing Benchmarks

3.1.1 SyGuS Competition

The first set of benchmarks we use are taken from the Syntax Guided Synthesis competition. Recall that the benchmarks in the competition are divided into the following tracks:

- A general track, which comprises of benchmarks that require synthesising functions that: manipulate arrays; manipulate bitvectors; solve problems from the Hackers Delight book [76]; use complex branching structures over arithmetic expressions; simplify complex expressions; and control robot movement in motion planning benchmarks.
- An invariant generation track [1, 54], in which solvers must generate loop invariants over bitvectors or integers.
- A programming by example track, in which solvers must synthesise programs that satisfy a set of input-output examples.

The invariant generation and general tracks are composed primarily of benchmarks using Linear Integer Arithmetic, but for this dissertation we also convert these benchmarks into bitvectors to expand the collection of bitvector benchmarks (this changes the semantics of the benchmarks subtly). We are motivated to do this because a key application for program synthesis is synthesising and verifying code, and typically programming languages such as C represent integers as bitvectors, and do not have a way of representing true integers. The programming by example track is split into bitvectors and string manipulation programs. We do not consider the string manipulation benchmarks in this dissertation.

3.1.1.1 A typical SyGuS-IF problem

Recall that the program synthesis problem is expressed as $\exists P. \forall \vec{x} \in \mathcal{I}. \sigma(\vec{x}, P)$. A typical SyGuS benchmark from the general track, expressed in SyGuS-IF [115], is given below. It is comprised of the following:

- a statement which sets the logic being used in the benchmark, in this case Linear Integer Arithmetic;
- the signature of the function to be synthesised, `max`, which corresponds to P in the program synthesis formula;
- the syntactic template from which it should be constructed;
- a declaration of the variables that will be used in the logical constraints on the program, `x` and `y`, which correspond to the vector of input variables $\vec{x}$ in the program synthesis formula; and
- a series of constraints that specify the behaviour of the program to be synthesised, and correspond to σ in the program synthesis formula.

```
(set-logic LIA)

(synth-fun max ((x Int) (y Int)) Int
    ((Start Int (x
                 y
                 (ite StartBool Start Start)))
     (StartBool Bool ((<=  Start Start)
                      (>=  Start Start)))

(declare-var x Int)
(declare-var y Int)

(constraint (>= (max x y) x))
(constraint (>= (max x y) y))
(constraint (or (= x (max x y))
                (= y (max x y))))


(check-synth)
```

The constraints, expressed in classical logic, are as follows:

$$\begin{aligned} & \max(x, y) \geq x \\ \wedge & \max(x, y) \geq y \\ \wedge & (\max(x, y) = x \vee \max(x, y) = y) \end{aligned}$$

A program that satisfies the specification will be a program that returns the maximum number passed to it. The type of is $P : Int \times Int \rightarrow Int$.

Recall that the syntax guided synthesis paradigm uses a syntactic template to restrict the program synthesis task. Our research hypothesis is that program synthesis is computationally tractable without provision of this template. A restrictive syntactic template like the example above allows the synthesis problem to be solved very effectively by enumerative methods such as EUSolver [12]. Consider the syntactic template in the exemplar below:

```
    ((Start Int (x
                 y
                 (ite StartBool Start Start)))
     (StartBool Bool ((<=  Start Start)
                      (>=  Start Start))
```

That is:

```
  formula :: (formula) | atom
     atom :: term rel term
      rel :: ≥ | ≤
     term :: atom?term:term | x | y
```

We can enumerate the set of expressions given by the grammar:

```
x
y
ite(x <= y, x, y)
ite(x <= y, y, x)
ite(y <= x, x, y)
ite(y <= x, y, x)
ite(x >= y, x, y)
ite(x <=y, ite(y<=x, x, y), x)
...
```

We can intuitively see that an enumerative approach would do well on this problem. However, in cases where the template is less restrictive, or the solution requires arbitrary constants not given in the template, SAT- and SMT-based approaches often fare better. In our experiments we remove the syntactic templates from the benchmarks in order to compare solvers based on their ability to solve problems without a syntactic template.

3.1.1.2 Loop Invariants

There are many problems that can be expressed as synthesis problems of the form above. A class of benchmarks in the syntax guided synthesis competition is the problem of synthesising *loop invariants* [10,55]. A loop invariant is a formal statement

about the relationship between variables in a program which is true before the loop is entered and after each iteration of the loop. Loop invariants can be viewed as an abstraction of the loop, and can be used in program analysis to verify the safety of loops where otherwise the verification would need to explicitly analyse an unknown or large number of iterations of the loop. To prove a loop is safe, we must find a valid loop invariant that if true implies that the safety property is also true.

The general form of a loop is:

```
1 assume(I);
2 while (G) T;
3 assert (A);
```

We are interested in invariants that prove a given assertion holds in a program. A formal definition of such a loop invariant is as follows: Let $\vec{x}$ denote a program state, i.e., the values for a set of program variables at a given point in the program, I denote the predicate for the initial condition, A an assertion, G the loop guard and T a transition relation.
A predicate P is such an invariant if it is a model for the following formula:

$$\begin{aligned} \exists P\ \forall \vec{x}, \vec{x'}.(I(\vec{x}) \Rightarrow P(\vec{x})) \wedge \\ (P(\vec{x}) \wedge G(\vec{x}) \wedge T(\vec{x}, \vec{x'}) \Rightarrow P(\vec{x'})) \wedge \\ (P(\vec{x}) \wedge \neg G(\vec{x}) \Rightarrow A(\vec{x})). \end{aligned}$$

Consider the following exemplar program:

```
1 int x=0;
2 While(x < 100000)
3 {
4   x++;
5 }
6 assert(x<100005);
```

A loop invariant that proves the safety of this program will prove that the assertion is not violated in any execution of the program. A correct invariant for this program would be $x < 100003$, which holds before the loop is entered, and after every iteration of the loop, and implies that the assertion is true.

We illustrate loop invariants by presenting the loop itself in C code, but the actual format of these benchmarks in the SyGuS competition is SyGuS-IF. The exemplar program above would be represented as follows. The full specification of the SyGuS-IF invariant syntax is given in [115]

```
; we use bitvectors to represent C integers
(set-logic BV)
```

```
; the invariant to be synthesised
(synth-inv inv-f ((x (BitVec 32))) Bool)

(declare-primed-var x (BitVec 32))

; initial conditions
(define-fun pre-f ((x (BitVec 32))) Bool
  (= x #x00000000 ))

; if the loop guard is true, x is incremented
(define-fun trans-f ((x (BitVec 32))(x! (BitVec 32))) Bool
    (and (bvult x #x000186A0) (= x! (+ x #x00000001)))
)

; the assertion: x is less than 100005
(define-fun post-f ((x (BitVec 32))) Bool
  (bvult x #x000186A5 ))

(inv-constraint inv-f pre-f trans-f post-f)

(check-synth)
```

3.1.2 Program analysis

In addition to benchmarks from the syntax-guided synthesis competition, we also consider benchmarks taken from program analysis [34, 35]. The benchmarks fall into one of two categories: synthesising safety invariants; and synthesising danger invariants [34]. The safety invariants are exactly as described in the preceding section. Danger invariants are the dual of safety invariants; they summarise a set of traces that are guaranteed to reach an error state. Both the danger invariant and safety invariant benchmarks are translated into SyGuS-IF syntax.

Danger Invariants A danger invariant is the complement of a safety invariant, we are looking for an invariant which, if true on exit of the loop, implies the assertion will fail. A trace that starts from the initial state and contains an error trace exists *iff* a predicate D can be found such that:

$$\begin{aligned} \exists \vec{x_0}.I(\vec{x}) &\Rightarrow D(\vec{x}) \wedge \\ \forall \vec{x} D(\vec{x}) \wedge G(\vec{x}) &\Rightarrow \wedge \\ \forall \vec{x}.P'(\vec{x'}) \wedge \neg G(\vec{x}) &\Rightarrow \neg A(\vec{x}) \end{aligned}$$

The above predicate potentially captures infinite traces, so in order to capture only finite traces, we must find a ranking function and a predicate.

A function $R : X \to Y$ is a ranking function for the transition relation T if Y is a well-founded set with order $>$ and R is injective and monotonically decreasing w.r.t. T, i.e.,

$$\forall \vec{x}, \vec{x'} \in X.T(\vec{x}, \vec{x'}) \Rightarrow R(\vec{x}) > R(\vec{x'}).$$

A danger invariant is thus defined as a pair D, R such that:

$$\begin{aligned} \exists \vec{x_0}.I(\vec{x}) &\Rightarrow D(\vec{x}) \wedge \\ \forall \vec{x} D(\vec{x}) \wedge G(\vec{x}) &\Rightarrow R(\vec{x}) > 0 \wedge \exists \vec{x'}.T(\vec{x}, \vec{x'}) \wedge D(\vec{x'}) \wedge R(\vec{x'}) < R(\vec{x}) \wedge \\ \forall \vec{x}.P'(\vec{x'}) \wedge \neg G(\vec{x}) &\Rightarrow \neg A(\vec{x}) \end{aligned}$$

Consider the following exemplar program, where "$*$" denotes a non-deterministic choice:

```
1 int x=0, y=1;
2 While(x < 100)
3 {
4   x++;
5   if(*) y++;
6 }
7 assert(x!=y);
```

A trace that violates the assertion is one where the nondeterministic choice is *TRUE* in 99 iterations of the loop. A danger invariant D would be $y = (x < 1\,?\,1 : x)$ and ranking function $R = 10 - x$, i.e., y is not incremented on the first iteration of the loop but is incremented on all other iterations.

3.2 New Benchmarks: Synthesising controllers for LTI systems

3.2.1 Rational

In this section we introduce a new set of benchmarks based on synthesising digital controllers for Linear Time Invariant systems. We use these benchmarks to evaluate the contribution of the incremental satisfiability solving in CEGIS, as described in Chapter 6, and the novel encoding of the synthesis problem, as described in Chapter 5, on a set of benchmarks that do not require synthesis of expressions. This allows us to hypothesise about the relative contributions of the three algorithmic developments we present when used in combination.

Modern implementations of embedded control systems have proliferated with the availability of low-cost devices that can perform highly non-trivial control tasks,

with significant impact in numerous application areas such as process and industrial engineering, high-precision control, automotive and robotics [16,50]. However, provably correct synthesis of control software for such platforms, needed if certification is in order, is non-trivial even in cases with unsophisticated dynamics.

We examine the case of physical systems (a.k.a. "plants" in control literature) described mathematically as Linear Time Invariant (LTI) models, for which the classical synthesis of controllers is well understood. However, the use of digital control architectures adds new challenges caused by artefacts specific to digital control, such as the effects of finite-precision arithmetic and quantization errors introduced by Analogue to Digital (A/D) and Digital to Analogue (D/A) conversion. Given an LTI model, we use program synthesis to generate correct-by-design digital controllers that address these challenges. Specifically, we automatically synthesise safe, software-implemented embedded controllers for physical plants. Our approach evaluates the effects of the A/D and D/A conversions, as well as representation errors introduced by the controller working in a finite precision domain.

We use a multi-staged program synthesis based technique that starts by devising a digital controller that stabilizes the plant model while remaining safe for a pre-selected time horizon and a single initial state; then, it verifies unbounded-time safety by unfolding the model dynamics, considering the full set of initial states, and checking a *completeness threshold* [81], i.e., the number of stages required to sufficiently unwind the closed-loop model, such that the safety boundaries are not violated for any larger number of iterations.

This new set of benchmarks requires synthesising safe controllers for a set of intricate physical plant models taken from the digital control literature.

Related work Existing work makes use of CEGIS for similar problems, for instance [117, 118] use CEGIS for synthesis of switching controllers, and [3] use CEGIS to synthesise stabilizing controllers for systems represented with transfer functions. Our work differs from [117, 118], which does not take into account the errors due to fixed-point arithmetic, and [3] which synthesises controllers only for stability and not safety properties.

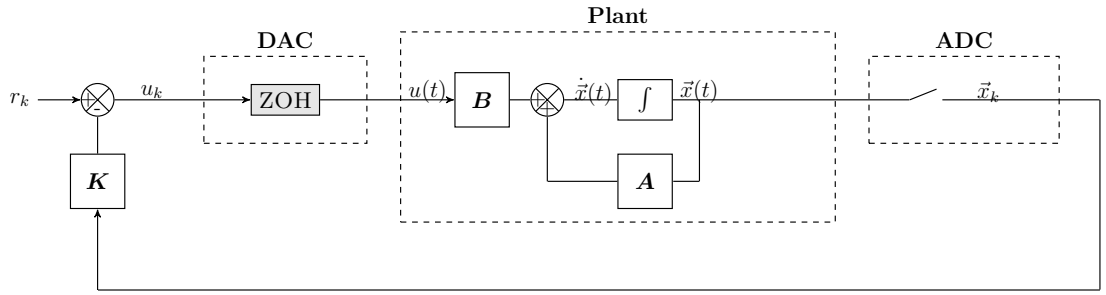


Figure 3.1: Closed-loop digital control setup, comprising an analogue model of the underlying real system, alongside a digital controller.

3.2.2 Preliminaries

3.2.2.1 State-space representation of physical systems

We consider models of physical plants expressed as ordinary differential equations, which we assume are controllable [14]:

$$\dot{x}(t) = \boldsymbol{A}\vec{x}(t) + \boldsymbol{B}\vec{u}(t), \tag{3.1}$$

where $\vec{x} \in \mathbb{R}^n$, $\vec{u} \in \mathbb{R}^p$, $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times p}$, and $t \in \mathbb{R}_0^+$ denotes continuous time. We denote with $x(0)$ the model initial condition, which can be non-deterministic.

Equation (3.1) is discretized in time [95, 147] with constant sampling intervals, each of duration T_s (the sample time), into the difference equation

$$\vec{x}_{k+1} = \boldsymbol{A}_d\vec{x}_k + \boldsymbol{B}_d\vec{u}_k, \tag{3.2}$$

where $\boldsymbol{A}_d = e^{\boldsymbol{A}T_s}$ and $\boldsymbol{B}_d = \int_{t=0}^{T_s} e^{\boldsymbol{A}t}dt\ \boldsymbol{B}$, $k \in \mathbb{N}$ is a discrete counter and where $\vec{x}_0 = \vec{x}(0)$ denotes the initial state. We assume that specifications on the model concern (3.2), and plan to devise controllers $\vec{u}_k$ to meet them. (The more general problem of synthesising controllers $u(t)$ for (3.1) falls outside the scope of the present work.)

3.2.2.2 Digital control synthesis

Models (3.1) and (3.2) depend on external non-determinism in the form of input signals $u(t)$ and u_k, respectively. Feedback architectures can be employed to manipulate properties and behaviors of the plant: we are interested in the synthesis of digital feedback controllers u_k, as in Fig. 3.1, as practically implemented on Field-Programmable Gate Arrays or Digital Signal Processors and as classically studied in [16].

We consider state feedback control architectures, where u_k (notice we work with the time discretized signal) is $u_k = r_k - Kx_k$. Here $K \in \mathbb{R}^{p\times n}$ is a state-feedback

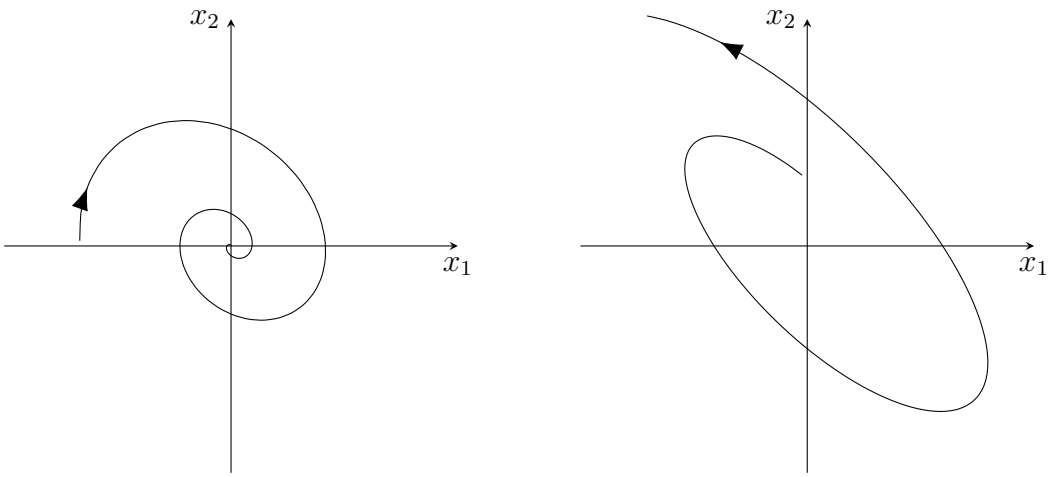
gain matrix, and r_k is a reference signal (again digital). We will assume $r_k = 0$, thus obtaining the closed-loop model $\vec{x}_{k+1} = (\boldsymbol{A}_d - \boldsymbol{B}_d\boldsymbol{K})\vec{x}_k$.

The gain matrix K can be set so that the closed-loop discrete dynamics are shaped as desired, for instance according to a specific stability goal or around a dynamical behavior of interest [16]. As argued later in this work, we will target a less standard objective, namely a quantitative safety requirement, which opens up to more complex specifications [21, 141]. This is not typical in the digital control literature. We will further precisely account for the digital nature of the controller, which manipulates quantized signals as discrete quantities represented with finite precision. The new benchmarks allow us to leverage a fully automated approach based on CEGIS.

3.2.2.3 Stability of closed-loop models

In this work we employ the notion of asymptotic stability, as a means for reducing the search space of possible safe controllers, where the notion of a safe controller is defined in the following section. As discussed later, specifically for linear models a safe controller is necessarily asymptotically stable, although the reverse is not true. Qualitatively, (local) asymptotic stability is a property denoting the convergence of the model executions towards an equilibrium point, starting from any states in a neighbourhood of the point. In the case of linear systems considered with a zero reference signal, the equilibrium point of interest is the origin. Figure 3.2a illustrates an example trajectory of an asymptotically stable execution, converging to the origin, and Figure 3.2b illustrations an example trajectory of an asymptotically unstable execution which diverges away from the origin.

It can be shown that a discrete-time LTI model is asymptotically stable if all the roots of its characteristic polynomial (i.e., the eigenvalues of the closed-loop matrix $A_d - B_dK$) are inside the unity circle of the complex plane, i.e., if their absolute values are strictly less than one [16]. Whilst this simple sufficient condition can be either generalised or strengthen to be necessary, this is not necessary in the context of this work. What is technically key is that in this , we shall express this asymptotic stability as a specification $\phi_{stability}$, and encode it in terms of a check known as *Jury's criterion* [43]: this is an easy algebraic formula to check the entries of matrix K, so that the closed-loop dynamics are shaped as desired. Jury's Criterion is a standard result in control literature.



(a) A stable execution (b) An unstable execution

Figure 3.2: Executions of a 2 dimensional Linear Time Invariant system shown in the state-space domain

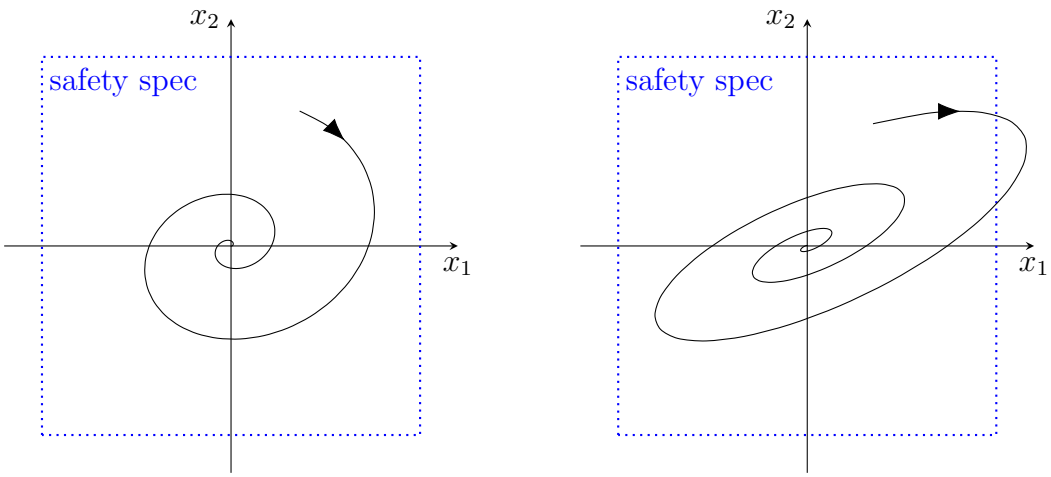
3.2.2.4 Safety of closed-loop models

As previously stated, we are not limited to the synthesis of digital stabilizing controllers – a well known task in the literature on digital control systems – but target safety requirements with an overall approach that is sound and automated. More specifically, we require that the closed-loop model meets a given safety specification. A safety specification gives rise to a requirement on the states of the model, namely that they remain within the safe set at all times (that is, over an infinite number of time steps). So the feedback controller (namely the choice of the gain matrix K) must ensure that the state never violates the requirement. Note that an asymptotically stable, closed-loop system is not necessarily a safe system: indeed, the state values may leave the safe part of the state space while they converge to the equilibrium, which is typical in the case of oscillatory dynamics. Figure 3.3a shows a trajectory of a stable system that is also safe, whereas Figure 3.3b shows a trajectory of a system that is stable and yet violates the safety specification. In this work, the safety property is expressed as:

$$\phi_{safety} = \left\{ \forall k \geq 0. \bigwedge_{i=1}^{n} \underline{x_i} \leq x_{i,k} \leq \overline{x_i} \right\}, \tag{3.3}$$

where $\underline{x_i}$ and $\overline{x_i}$ are lower and upper bounds for the i-th coordinate x_i of state $x \in \mathbb{R}^n$ at the k-th instant, respectively. This requires that the states will always be within an n-dimensional hyper-box.

Beyond the main requirement on safety, it is practically relevant to consider the constraints ϕ_{input} on the input signal u_k and ϕ_{init} on the initial states x_0, which



(a) A stable and safe execution

(b) A stable yet un unsafe execution

Figure 3.3: Executions of a 2 dimensional Linear Time Invariant system shown in the state-space domain. The blue dotted line represents the safety specification

we assume have given bounds: $\phi_{input} = \{\forall k \geq 0. \bigwedge_{i=1}^{p} \underline{u} \leq u_i \leq \overline{u}\}$, and $\phi_{init} = \{\bigwedge_{i=1}^{n} \underline{x_{i,0}} \leq x_{i,0} \leq \overline{x_{i,0}}\}$. The former constraint expresses that the control inputs, shaped via state-feedback, might saturate in view of physical constraints.

3.2.2.5 Semantics of finite-precision arithmetic

A key contribution of this work is that, when synthesising controllers, it precisely replicates the finite-precision arithmetic within the digital controller, thus guaranteeing that controllers implemented with finite-precision arithmetic are safe. The specific components of the model, shown in Figure 3.1, we are concerned with are the ADC, digital controller and DAC. Details of how we model the behaviour of the finite-precision arithmetic in these components are in Section 3.2.4. We must model the semantics precisely, in order to capture the full behaviour of the model. More specifically, we encompass the following features:

- The ADC converts the analog signal $x(t)$ into a digital signal x_k, which is then fed into the controller, converting a real value to a finite-precision value.
- The controller block performs arithmetic at finite precision. We assume the ADC represents numbers with at least the same precision as the controller, and thus focus on the precision as limited by the controller. This is a reasonable assumption based on commonly available hardware.

- The DAC converts finite-precision values back to real values. We assume that the input to the DAC has the same precision as the output of the controller. It would, however, be straightforward to account for a DAC or ADC of different precision than the controller in our algorithm, if necessary.

3.2.2.6 Soundness of modelling

In addition to precisely replicating the finite-precision arithmetic of the digital controller, we must consider that our *model* itself in (3.2) employs finite-precision arithmetic to represent the behaviour of the real system. In order to guarantee soundness, we must therefore encompass the error that is due to modelling (as opposed to the nature of the digital controller): the representations used in the plant model and its arithmetic operations are carried out at finite precision.

- We account for the error introduced by finite precision arithmetic applied over the model variables x_k and u_k, which are actually real values. We guarantee that the precision we use to represent the model variables is at least as precise as the precision used in the digital controller, and we use interval arithmetic to bound the incurred errors, as further detailed in Section 3.2.4.

3.2.2.7 Notation for fixed- and floating-point precision

In this section we will use $\mathcal{F}_{\langle I,F\rangle}(x)$ to denote a real number x expressed at a fixed-point precision, using I bits to represent the integer part of the number and F bits to represent its decimal part. In particular, $\mathcal{F}_{\langle I_c,F_c\rangle}(x)$ denotes a real number x represented at the fixed-point precision of the controller, and $\mathcal{F}_{\langle I_p,F_p\rangle}(x)$ denotes a real number x represented at the fixed-point precision of the plant model. I_c and F_c are determined by the controller. We pick I_p and F_p for our synthesis such that $I_p \geq I_c$ and $F_p \geq F_c$, so that our controller can be represented accurately in the model domain. Thus any mathematical operations in our modelled digital controller will be in the range of $\mathcal{F}_{\langle I_c,F_c\rangle}$, and all other calculations in our model will be carried out in the range of $\mathcal{F}_{\langle I_p,F_p\rangle}$.

We further employ $\mathcal{F}_{\langle E,M\rangle}(x)$ to denote a real number x represented in a floating-point domain, with E bits representing the exponent part and M bits representing the mantissa part. In particular, we use $\mathcal{F}_{\langle E_c,M_c\rangle}(x)$ to denote a real number represented at the floating-point precision of the controller, whereas $\mathcal{F}_{\langle E_p,M_p\rangle}(x)$ denotes a real number represented at the floating-point precision of the plant model.

3.2.3 Formal specification of properties on a model

We are now equipped to apply the general CEGIS framework to the synthesis of digital controllers. We start by describing the property that we pass to the synthesiser as the specification ϕ. Namely, as we are interested in capturing safety (as discussed in Section 3.2.2.4), we use a stability specification to narrow the search space of possible controllers, as detailed in Section 3.2.3.1.

3.2.3.1 Jury's stability criterion

There are a number of well known procedures to perform stability analysis of dynamical models [14]. Here we select the Jury stability criterion [16], in view of its efficiency and ease of integration within our implementation. This method checks the stability working in the complex domain of the characteristic polynomial $S(z)$, considered in its general form as

$$S(z) = a_0 z^N + a_1 z^{N-1} + \cdots + a_{N-1} z + a_N,\ a_0 \neq 0.$$

This polynomial is obtained as a function of the state-space matrices A_d, B_d [14], and in particular its order N corresponds to the dimensions of the state variables (above, n). A sufficient condition for asymptotic stability of the closed-loop LTI model [16] is when all the roots of its characteristic polynomial $S(z)$ (which correspond to the eigenvalues of the matrix $A_d - B_d K$) are inside the unit circle in the complex plane, i.e., if their absolute values are less than one.

The following matrix M with dimension $(2N-2) \times N$ and elements $m_{(\cdot),(\cdot)}$ is built from the coefficients of $S(z)$ as:

$$M = \begin{pmatrix} V^{(0)} \\ V^{(1)} \\ \vdots \\ V^{(N-2)} \end{pmatrix},$$

where $V^{(k)} = [v_{ij}^{(k)}]_{2\times N}$ is such that:

$$v_{ij}^{(0)} = \begin{cases} a_{j-1}, & \text{if } i = 1 \\ v_{(1)(N-j+1)}^{0}, & \text{if } i = 2 \end{cases}$$

$$v_{ij}^{(k)} = \begin{cases} 0, & \text{if } j > n-k \\ v_{1j}^{(k-1)} - v_{2j}^{(k-1)} . \frac{v_{11}^{(k-1)}}{v_{21}^{(k-1)}}, & \text{if } j \leq n-k \text{ and } i = 1 \\ v_{(1)(N-j+1)}^{k}, & \text{if } j \leq n-k \text{ and } i = 2 \end{cases}$$

and where $k \in \mathbb{Z}$ is such that $0 < k < N-2$.

We have that $S(z)$ is the characteristic polynomial of an asymptotically stable system if and only if the following four conditions R_i, for $i = 1, 2, 3, 4$ hold [16]:

$$\begin{aligned}
R_1 &: S(1) && > 0 \\
R_2 &: (-1)^N S(-1) && > 0 \\
R_3 &: |a_0| && < a_N \\
R_4 &: m_{11} && > 0 \ \wedge \\
& \quad m_{31} && > 0 \ \wedge \\
& \quad m_{51} && > 0 \ \wedge \ldots \ \wedge \\
& \quad m_{(2N-3)(1)} && > 0
\end{aligned}$$

In conclusion, the asymptotic stability property is finally encoded by a constraint expressed as the following formula:

$$\phi_{stability} = \{R_1 \wedge R_2 \wedge R_3 \wedge R_4\}.$$

3.2.4 Numerical representation and soundness

As discussed in Section 3.2.2.5, the considered models must account for the semantics of finite-precision arithmetic, deriving from several sources: we formally bound the numerical error introduced by the finite precision representation of the plant (and its operations), and precisely model the behaviour introduced by the ADC/DAC conversions and the behaviour of the limited-precision arithmetic used by the controller.

Technically, we employ interval arithmetic to bound the error introduced by the finite-precision plant model, and we use bitvector semantics to precisely model the semantics of finite-precision arithmetic as introduced by the ADC/DAC blocks and by the finite-precision controller.

3.2.4.1 Interval arithmetic for errors in numerical representations

We use finite-precision arithmetic to model the plant. This is an approximation that speeds up each CEGIS iteration, however it necessitates a further stage where we verify that the errors introduced by the approximation have not resulted in a controller that is unsafe when executed on a model expressed over real numbers. In this stage, we represent the plant model using double-precision floating-point numbers and we use the Boost interval arithmetic library [25] to bound the error in this representation. We employ a compositional numerical library to model the fixed point arithmetic for the controller[1] within double-precision floating-point numbers. We check that the

[1] `https://github.com/johnmcfarlane/cnl`

controller is safe starting from each vertex of the set of initial states, and show that this is sufficient to prove safety from any state in this set (see Theorem 1).

We outline here the mathematics behind bounding the errors on the double-precision floating-point numbers. Recall we use $\mathcal{F}_{\langle E,M\rangle}(x)$ denote a real number x represented in a floating-point domain, with E bits representing the exponent part, and M bits representing the mantissa. In general the representation of a real number using the floating-point domain introduces an error, for which an upper bound can be given [24]. For each number x represented in the floating-point domain as $\mathcal{F}_{\langle E,M\rangle}(x)$, we store an interval that encompasses this error. Further mathematical operations performed at the precision $\mathcal{F}_{\langle E,M\rangle}(x)$ will propagate this error, leading to further errors for which bounds can be derived [24].

The fixed-point arithmetic of the digital controller is performed on the upper and lower bound of the intervals from above independently, and the upper and lower bound of the result is taken as the interval result. For example, consider the conversion from the model precision to controller precision performed by the ADC on a single state value. The state value is represented as an interval $\{x.high, x.low\}$, and the result of the conversion is an interval where the upper bound is the conversion of $x.high$ and the lower bound is the conversion of $x.low$. Since the precision of the floating-point domain is greater than the precision of the controller, this is guaranteed to bound the real behaviour of the controller.

3.2.5 Effect of finite precision arithmetic on safety specification and on stability

In this section we will quantify how the finite precision arithmetic in a digital controller affects the safety and stability properties of an LTI model.

3.2.5.1 Safety of closed-loop models with finite precision controller error

Let us first consider the effect of the quantization errors on safety. Within the controller, state values are manipulated at low precision, by means of the vector multiplication Kx. The inputs are thus computed using the following equation:

$$u_k = -(\mathcal{F}_{\langle I_c,F_c\rangle}(K) \cdot \mathcal{F}_{\langle I_c,F_c\rangle}(x_k)).$$

This induces two types of errors, as detailed above: first, the truncation error due to the representation of x_k as $\mathcal{F}_{\langle I_c,F_c\rangle}(x_k)$; and second, the rounding error introduced by the multiplication operation. This error is modelled precisely by bounded model checking.

An additional error is due to the representation of the plant dynamics, namely

$$x_{k+1} = \mathcal{F}_{\langle I_p,F_p\rangle}(A_d)\mathcal{F}_{\langle I_p,F_p\rangle}(x_k) + \mathcal{F}_{\langle I_p,F_p\rangle}(B_d)\mathcal{F}_{\langle I_p,F_p\rangle}(u_k).$$

We encompass this error using interval arithmetic [97] in the precision check shown in Figure 3.4 and detailed in the previous section.

3.2.5.2 Stability of closed-loop models with fixed-point controller error

The validity of Jury's criterion [43] relies on the representation of the closed-loop dynamics $x_{k+1} = (A_d - B_dK)x_k$ at infinite precision. When we employ a fixed-point arithmetic digital controller, the operation above can be expressed as follows, where we use $\mathcal{F}_{\langle I_c,F_c\rangle}$ preceding a variable to indicate that variable is converted into the fixed-point precision given by $\mathcal{F}_{\langle I_c,F_c\rangle}$:

$$x_{k+1} = A_d \cdot x_k - B_d(\mathcal{F}_{\langle I_c,F_c\rangle}(K) \cdot \mathcal{F}_{\langle I_c,F_c\rangle}(x_k)).$$

This translates to

$$x_{k+1} = (A_d - B_dK) \cdot x_k + B_dK\delta,$$

where δ is the maximum error that can be introduced by the digital controller in one step, i.e., by reading the states values once and multiplying by K once. We derive the closed form expression for x_n recursively, as follows:

$$\begin{aligned}
x_1 &= (A_d - B_dK)x_0 + B_dK\delta \\
x_2 &= (A_d - B_dK)^2x_0 + (A_d - B_dK)B_dK\delta + B_dK\delta \\
x_n &= (A_d - B_dK)^nx_0 + (A_d - B_dK)^{n-1}B_dK\delta + \\
&\quad \ldots + (A_d - B_dK)^1B_dK\delta + B_dK\delta \\
&= (A_d - B_dK)^nx_0 + \sum_{i=0}^{n-1}(A_d - B_dK)^iB_dk\delta.
\end{aligned}$$

Recall that a closed-loop asymptotically stable system will converge to the origin. We know that the original system with an infinite precision controller is stable, because we have synthesised it to meet Jury's criterion. Hence, $(A_d - B_dK)^nx_0$ must converge to zero as $n \uparrow \infty$. Furthermore, the power series of a square matrix T converges [70] iff the eigenvalues of the matrix are less than 1, and the limit results in $\sum_{i=0}^{\infty} T^i = (I - T)^{-1}$, where I is the identity matrix. Thus, the closed-loop model converges to the value

$$0 + (I - A_d + B_dK)^{-1}B_dk\delta\,.$$

As a result, if the value $(I - A_d + B_d K)^{-1} B_d k\delta$ is within the safe set of states given by the safety specification then the synthesised fixed-point controller results in a safe closed-loop model. The convergence to a finite value, however, will not make it asymptotically stable. However, since we require stability only as a precursor to safety, it is sufficient to check that the perturbed model converges to a neighborhood of the equilibrium within the safe set.

A similar argument can be made for floating-point arithmetic. We can thus disregard these steady-state errors (caused by finite precision arithmetic) when stability is ensured by synthesis, and then verify its safety accounting for the finite precision errors.

3.2.6 Synthesis of digital controllers with CEGIS

In this section we discuss the way the CEGIS procedure is used to synthesise safe digital controllers via these benchmarks, accounting for the precision issues detailed in the previous sections. We employ a multi-stage approach that unwinds the dynamics of the model up to a completeness threshold, encompassing finite precision arithmetic using bit-precise bounded model checking, and then verifying soundness of the resulting controller using interval arithmetic.

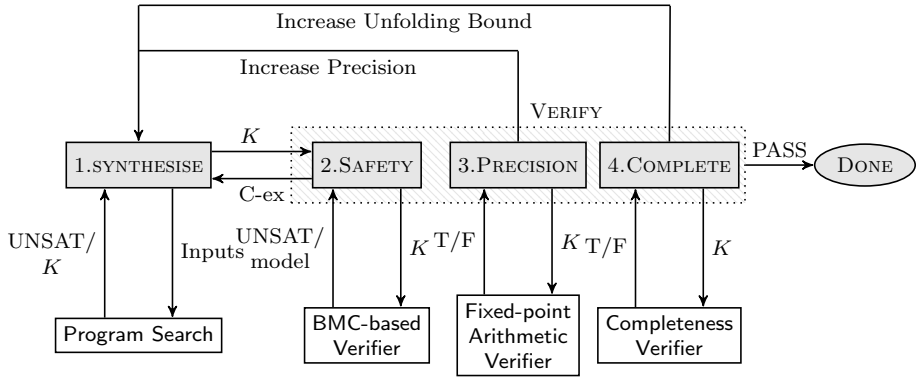


Figure 3.4: CEGIS with multi-staged verification for digital controller synthesis

An overview of the algorithm for controller synthesis is given in Figure 3.4. One important point is that we formally synthesise a controller over a finite number (k) of time steps (i.e., it is multi-stage). We then compute a completeness threshold $\overline{k}$ [81] for this controller, and verify the correct behaviour for $\overline{k}$ time steps. As we will later argue, $\overline{k}$ is the number of iterations required to sufficiently unwind the dynamics of

the closed-loop state-space model, ensuring that the safety boundaries are not violated for any other $k>\overline{k}$.

Next, with reference to the CEGIS scheme in Figure 1.1, we describe in detail the different phases in Fig. 3.4 (shaded blocks 1 to 4).

3.2.6.1 SYNTHESISE Block

The inductive synthesis phase (SYNTHESISE) uses BMC to compute a candidate solution K that satisfies both the stability requirement and the safety specification, within a finite-precision model. In order to synthesise a controller that satisfies the stability requirement, we need the characteristic polynomial of the closed-loop model to satisfy Jury's criterion [43] (see Section 3.2.3.1).

We fix an index k, and we synthesise a safe controller by unfolding the transition system (i.e., the closed-loop model) k steps and by selecting a controller K and a single initial state, such that the states at each step do not violate the safety criteria (see Section 3.2.2.4). That is, we ask the bounded model checker [31] if there exists a K that is safe for at least one x_0 in our set of all possible initial states, and a given fixed-point precision for the controller, and where the input signal remains within the specified bounds. This approach is sound, i.e., the controller produced is guaranteed to be safe, if the current k is greater than the completeness threshold (see later step). We also assume some finite precision for the model and a given time discretisation, as described in section 3.2.2.1. We use a fixed-point precision in this description, given by $\langle I_p, F_p \rangle$, but if we are considering a floating-point controller we will instead model the plant with a floating-point precision, $\langle E_p, M_p \rangle$. The checks that these assumptions hold are performed by the subsequent VERIFY stages.

Algorithm 1 Safety check (SAFETY stage in Fig. 3.4)

```
1:  function safetyCheck()
2:      assert(u ≤ u ≤ ū)
3:      set x_0 to be a vertex state, e.g., [x_0, x_0]
4:      for (c = 0; c < 2^n; c++) do
5:          for (i = 0; i < k; i++) do
6:              u = (plant_typet)((controller_typet)K * (controller_typet)x)
7:              x = A * x + B * u
8:              assert(x ≤ x ≤ x̄ )
9:          end for
10:         set x_0 to be a new vertex state
11:     end for
12: end function
```

3.2.6.2 SAFETY Block

The first VERIFY stage, SAFETY is shown in Alg. 1. The algorithm checks that the candidate solution K, which we have synthesised to be safe for at least one initial state, is safe for *all* possible initial states, i.e., it does not reach the (complement) unsafe set within k steps. After unfolding the transition system corresponding to the previously synthesised controller k steps, we check that the safety specification holds for any initial state.

We use (*controller_typet*) to denote a cast or conversion to the controller precision, and (*plant_typet*) to denote a cast or conversion to the model precision. A *vertex state* is defined as a state where all values are either equal to the upper or lower bound of the states in the initial set. It is sufficient to verify that the controller is safe for all vertex states, as shown next.

Theorem 1. *If a controller is safe for each of the corner cases of the hypercube of allowed initial states, i.e., the vertex states, then it is safe for any initial state in the hypercube.*

Proof. Consider the set of initial states, X_0, which we assume to be convex since it is a hypercube. Name v_i its vertices, where $i = 1, \ldots, 2^n$. Thus any point $x \in X_0$ can be expressed by convexity as $x = \sum_{i=1}^{2^n} \alpha_i v_i$, where $\sum_{i=1}^{2^n} \alpha_i = 1$. Then if $x_0 = x$, we obtain

$$\begin{aligned} x_k &= (A_d - B_d K)^k x = (A_d - B_d K)^k \sum_{i=1}^{2^n} \alpha_i v_i \\ &= \sum_{i=1}^{2^n} \alpha_i (A_d - B_d K)^k v_i = \sum_{i=1}^{2^n} \alpha_i x_k^i, \end{aligned}$$

where x_k^i denotes the trajectories obtained from the single vertex v_i. We conclude that any k-step trajectory is encompassed, within a convex set, by those generated from the vertices. □

In conclusion, we only need to check 2^n initial states, where n is the dimension of the state space (number of continuous variables).

3.2.6.3 PRECISION Block

The second VERIFY stage, PRECISION, restores soundness with respect to the plant model precision by using interval arithmetic [97] to validate the operations performed by the previous stage.

3.2.6.4 COMPLETE Block

The third and last VERIFY stage, COMPLETE, checks that the current k is large enough to ensure safety for any further time steps. Here, we compute the completeness threshold $\overline{k}$ for the current candidate controller K and check that $k \geq \overline{k}$. This is done by computing the number of time steps required for the states to have completed a 360°circle, as illustrated in Fig. 3.5.

Theorem 2. *There exists a finite $\overline{k}$ such that it is sufficient to unwind the closed-loop state-space model up to $\overline{k}$ in order to ensure that ϕ_{safety} holds.*

Proof. An asymptotically stable model is known to have converging dynamics. Assume the closed-loop matrix eigenvalues are not repeated (which is sensible to do, since we select them). The distance of the trajectory from the reference point (origin) decreases over time within subspaces related to real-valued eigenvalues; however, this is not the case in general when dealing with complex eigenvalues. Consider the closed-loop matrix that updates the states in every discrete time step, and select the eigenvalue ϑ with the smallest (non-trivial) imaginary value. Between every pair of consecutive time steps $k\,T_s$ and $(k+1)\,T_s$, the dynamics projected on the corresponding eigenspace rotate ϑT_s radians. Thus, taking $\overline{k}$ as the ceiling of $\frac{2\pi}{\vartheta T_s}$, after $k \geq \overline{k}$ steps we have completed a full rotation, which results in a point closer to the origin. The synthesised $\overline{k}$ is the completeness threshold. □

3.2.7 Description of the control benchmarks

Our benchmark suite consists of 19 case studies extracted from the control literature [28, 50, 52, 52, 80, 104, 137, 142, 143]. These case studies comprise control models, which we represent in state space form as in (3.1). The models are time discretized, with sampling times ranging from 0.0001s to 1s. CEGIS initially starts with the coarsest discretisation (i.e., 1s sampling time), and if it fails to find a controller, reduces the sampling time. The initial states are bounded between 0.5 and -0.5 for all states, and the safety specification requires that the states remain between -1.5 and 1.5. The input bounds are selected individually for each benchmark.

The *acrobot* plant corresponds to the model of a two-link underactuated robot manipulator, named acrobot, linearized by partial feedback linearization [137]. The *antenna* benchmark (#2) describes the dynamics of the azimuth angle of an antenna motion system composed by a gearbox, a DC motor and amplifiers [52].
The *ballmaglev* plant corresponds to the model of a simple eletromagnet-ball system,

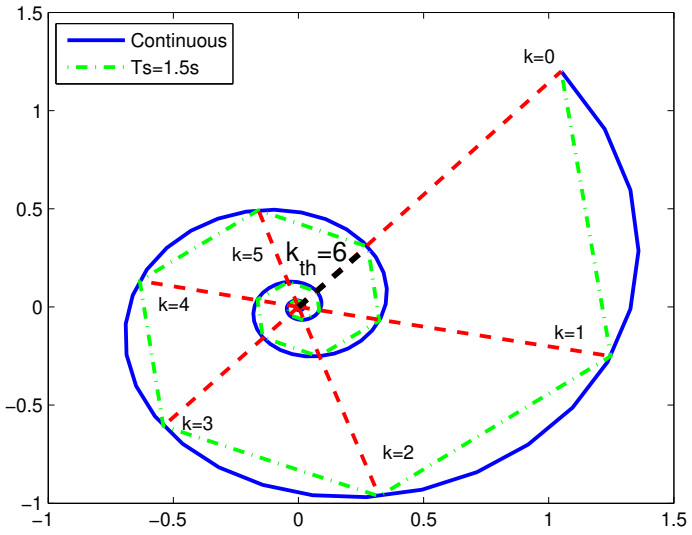


Figure 3.5: Completeness threshold for multi-staged verification (quantity T_s is the time discretization step)

where a steel ball can be levitated by the force generated by an electromagnet [50]. The *bioreactor* benchmark is a linear model of the cell mass concentration controlled through the dilution rate of a bioreactor [114]. The *Chen* benchmark correspond to a higher-order control system model employed as a case study for model-order reduction techniques [28]. The benchmarks *Cruise 1* [50] and *Cruise 2* [14] deal with automotive cruise control models, where the control is used to maintain the speed of an automobile constant, tracking a desired speed reference, and compensating disturbances and uncertainties. *cstr* and *cstrtmp* describe the pH dynamics of a reaction of an aqueous solution of sodium acetate with hydrochloric acid [142] and the temperature of a reaction [15] in a tank reactor. The *DC motor* plant describes the velocity dynamics of a direct-current electrical machine. The *Flexbeam* benchmark is a plant employed in vibration studies and models the physical system consists of a flexible metallic structure with sensors and actuators. The *guidance chen* benchmark is a high-order system model used by Chen et al. [28] to evaluate model-order reduction techniques; it models a guidance control system, which is used to determine the path of an autonomous vehicle.

The *helicopter* benchmark plant describes the transitional and rotational dynamics of a coaxial helicopter. The *inverted pendulum* benchmark describes a model for the cart position and the angle of an inverted pendulum placed on the cart, which moves over a track through a DC motor. The *magnetic pointer* benchmark describes

a magnetic pointer, whose angular position dynamics is controlled by a magnetic field. The *magsuspension* describes the dynamic of a magnetic car suspension system. The *pendulum* plant consists of a swinging point mass suspended from a frictionless pivot by means of a rod with negligible mass. The *Regulator* consists of a linear model of a synchronous electrical machine [80]. The *satellite attitude control system* plant describes the dynamics of a satellite attitude, *i.e.*, the orientation angles. An attitude control system must maintain the desired orientation of a satellite with respect to an inertial frame during all the excursion. The *springer-mass damper system* plant is a standard model for the dynamics of several mechanical systems. The *steam drum* benchmark describes a linear model for the level dynamics of a boiler steam drum [91]. The *suspension* models the single-wheel suspension system of a car, namely the relative motion dynamics of a mass-spring-damper model, which connects the car to one of its wheels. The *tapedriver* benchmark models a computer tape driver, that is a computer storage device used to read and write data on magnetic tapes. The *USCG cutter tampa heading angle* plant describes the heading angle dynamics of a Coast Guard cutter.

3.2.8 Benchmark format

In order to process the control benchmarks, we introduce a front end to our synthesis engine that processes a form of annotated C. This is necessary because the SyGuS-IF format does not allow loops or recursion in the standard benchmarks. It is not straight-forward to encode the control benchmarks as a loop invariant benchmark because a loop invariant, in SyGuS-IF format, must return a function, not a matrix of constants.

3.2.8.1 C front-end

For benchmarks in annotated *C*, recursion free C programs are used as input plus the following annotations:

- All programs to be synthesised must contain `EXPRESSION` in their name
- Any inputs to the program must be initialised with a call to a function containing `nondet` in its name.
- `__CPROVER_assert()` is used to introduce assertions that the program must satisfy
- `__CPROVER_assume()` is used to introduce assumptions about values which the inputs may take.

An example specification is shown below:

```
1  // program to be synthesised
2  int EXPRESSION(int);
3  // return nondeterministic integer
4  int nondet();
5
6  int main()
7  {
8    int in=nondet();
9    int out;
10   out=EXPRESSION(in);
11   __CPROVER_assert(out==in<<3, "assertion 1");
12 }
```

Inputs to the C-front end are converted into Boolean formulae via the same techniques as bit-precise bounded model checking [83]. That is, the code is parsed and converted into an intermediate representation known as a *GOTO program*. In GOTO programs all non-linear control flow, such as if statements and switch statements are translated into guarded goto statements. *Symbolic execution* is then performed on this intermediate representation, unwinding loops up to a fixed bound. Any loops without easily determined bounds may be unwound indefinitely unless a limit is specified by the user.

3.2.9 Example benchmark

The C-front end for the control benchmarks allows us the flexibility to include different headerfiles and sections of code based on variables in the benchmarks such as the number of states, without needing to reproduce a completely new version of the code for each benchmark. The pseudocode for a benchmark is shown here. For each benchmark, the values for A, B and NSTATES would be changed.

```
// Benchmark plant variables
#define NSTATES 2
plant_typet A[2][2]={{0.901,0.001},{0.07,0}};
plant_typet B[2]={128,0};

controller_typet K[NSTATES];
plant_typet states[NSTATES];

int main(void)
{
  K = EXPRESSION();
  for(int i=0; i<NSTATES; i++)
  {
    states[i]=nondet();
    assume(states[i] < initial_state_upper_bound);
    assume(states[i] > initial_state_lower_bound);
  }

  assert(check_stability());
  assert(check_safety());
}
```

```
bool check_stability()
{
  calculate_characteristic_polynomial();
  return(jury_criterion_satisfied());
}
```

```
bool check_safety()
{
  for(int i=0; i<NUMBER_LOOPS; i++)
  {
    input = -K * (controller_typet)states;
    if(input > input_upper_bound ||
      input < input_lower_bound)
        return false;

    states = A * states + B * (plant_typet)input;
    for(int i=0; i<NSTATES; i++)
    {
      if(states[i] > safety_upper_bound ||
          states[i] < safety_lower_bound)
          return false;
    }
  }
  return true;
}
```

A set of results on these benchmarks can be found in subsequent chapters and in

category	total	
invariant generation	47	from SyGuS competition and program analysis
comparisons	7	e.g., find the maximum
hacker's	6	from Hacker's Delight [76]
other SyGuS	23	
controller synthesis	29	6 time discretisations per benchmark

Table 3.1: Categories of the benchmarks

Section 7.

3.3 Summary of benchmarks

There are 112 benchmarks split into the categories shown in Table 3.1. 56 benchmarks require synthesising only one program. The number of program input arguments ranges from one through to twelve. The number of times the program is called ranges from one through to fifty. There are 27 benchmarks that require synthesising more than one function. All invariant benchmarks require one program to be synthesised and have four calls to that program.
For each section of this dissertation, we use the following subsets of benchmarks for the evaluation:

- CEGIS($\mathcal{T}$): we use all the Syntax Guided Synthesis competition benchmarks and the program analysis benchmarks. We omit the control benchmarks, as these do not require synthesising expressions, just constants, and so do not fully evaluate CEGIS($\mathcal{T}$), for reasons elaborated on in Chapter 4.

- Encoding: we evaluate the new synthesis encoding using the control benchmarks, and a subset of the syntax guided synthesis benchmarks.

- Incremental: we evaluate the incremental CEGIS implementation using the control benchmarks, syntax guided synthesis benchmarks and the program analysis benchmarks.

3.4 Experimental setup

In this dissertation we conduct the experimental evaluations on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM and Linux OS. We use the Linux *times*

command to measure CPU time used for each benchmark. We use MiniSat [42] as the SAT solver, and Z3 v4.5.1 [37] as the SMT-solver where we use SMT solvers.

Chapter 4

Synthesis Modulo Theories: CEGIS($\mathcal{T}$)

This chapter is based on work published at CAV 2018 [5], in collaboration with Pascal Kesseli, Daniel Kroening, Cristina David and Alessandro Abate. The work presented in this chapter has subsequently been implemented as part of CVC4 and is available in release 1.7[1]. My contributions to this work are as follows:

- Implementation of the SyGuS front-end, including parsing, for our experimental evaluation.
- Implementation of the use of incremental satisfiability within CEGIS, detailed in Chapter 6.
- Collaboration on development of the general CEGIS($\mathcal{T}$) algorithm with Daniel Kroening.
- Collaboration on the CEGIS($\mathcal{T}$)-FM algorithm with Daniel Kroening.
- Collaboration on writing with Cristina David.
- Accumulation of benchmarks and translation of SyGuS Linear Integer Arithmetic benchmarks into bitvectors.
- Running and write up of the experiments.

[1] `https://github.com/CVC4/CVC4`

In this chapter, we present the key contribution of this dissertation: a new approach to program synthesis that combines the strengths of a counterexample-guided inductive synthesiser with those of a solver for a first-order theory in order to perform a more efficient exploration of the solution space, without relying on user guidance. Our inspiration for this proposal is DPLL($\mathcal{T}$), which has boosted the performance of solvers for many fragments of quantifier-free first-order logic [53, 103]. DPLL($\mathcal{T}$) combines reasoning about the Boolean structure of a formula with reasoning about theory facts to decide satisfiability of a given formula.

In an attempt to generate similar technological advancements in program synthesis, we propose a new algorithm for program synthesis called CounterExample-Guided Inductive Synthesis($\mathcal{T}$), or CEGIS($\mathcal{T}$)for short, where $\mathcal{T}$ is a given first-order theory for which we have a specialised solver. Similar to its counterpart DPLL($\mathcal{T}$), the CEGIS($\mathcal{T}$) architecture features communication between a synthesiser and a theory solver, which results in a more efficient exploration of the search space.

While standard CEGIS architectures [72, 135] already make use of SMT solvers, the typical role of such a solver is restricted to validating candidate solutions and providing concrete counterexamples that direct subsequent search. The concrete counterexamples are used to prune the search space of candidate programs. However, a key disadvantage of this restricted use is that a concrete counterexample may only eliminate a single candidate solution. This is inefficient and can result in an enumerative search of the candidate programs. By contrast, CEGIS($\mathcal{T}$) allows the theory solver to communicate generalised constraints back to the synthesiser, thus enabling more significant pruning of the search space. A generalised constraint can rule out a larger portion of the search space of candidate programs, preventing enumerative behaviour in many cases.

There are instances of more sophisticated collaboration between a program synthesiser and theory solvers. The most advanced such instance is the program synthesiser inside the CVC4 SMT solver [120]. This approach features a very tight coupling between the two components (i.e., the synthesiser and the theory solvers) that takes advantage of the particular strengths of the SMT solver by reformulating the synthesis problem as the problem of refuting a universally quantified formula. This is effective because SMT solvers are better at refuting universally quantified formulae than at proving them. Conversely, in our approach, we maintain a clear separation between the synthesiser and the theory solver while performing comprehensive and well-defined communication between the two components. This enables the flexible combination of

CEGIS with a variety of theory solvers, which excel at exploring different solution spaces.

Relationship to related work In this section we describe how our work differs from the related work; a more comprehensive overview of the related work is found in Chapter 2.

A well-known application of CEGIS is program sketching [134, 136], where the programmer uses a partial program, called a *sketch*, to describe the desired implementation strategy, and leaves the low-level details of the implementation to an automated synthesis procedure. Inspired by sketching, Syntax-Guided Program Synthesis (SyGuS) [8] requires the user to supplement the logical specification provided to the program synthesiser with a syntactic template that constrains the space of solutions. In contrast to SyGuS, our aim is to improve the efficiency of the exploration to the point that user guidance is no longer required.

Another very active area of program synthesis is denoted by component-based approaches [7, 45, 47, 48, 63, 65, 109]. Such approaches are concerned with assembling programs from a database of existing components and make use of various techniques, from counterexample-guided synthesis [63] to type-directed search with lightweight SMT-based deduction and partial evaluation [47] and Petri-nets [48]. The techniques developed in the current paper are applicable to any component-based synthesis approach that relies on counterexample-guided inductive synthesis.

Heuristics for constant synthesis are presented in [35], where the solution language is parameterised, inducing a lattice of progressively more expressive languages. One of the parameters is word width, which allows synthesising programs with constants that satisfy the specification for smaller word widths. Subsequently, heuristics extend the program (including the constants) to the required word width. As opposed to this work, CEGIS($\mathcal{T}$) denotes a systematic approach that does not rely on ad-hoc heuristics.

Regarding the use of SMT solvers in program synthesis, they are frequently employed as oracles. By contrast, Reynolds et al. [120] present an efficient encoding able to solve program synthesis constraints directly within an SMT solver. Their approach relies on rephrasing the synthesis constraint as the problem of refuting a universally quantified formula, which can be solved using first-order quantifier instantiation. Conversely, in our approach we maintain a clear separation between the synthesiser and the theory solver, which communicate in a well-defined manner. Since

the publication of CEGIS($\mathcal{T}$), the algorithm has been integrated in CVC4 version 1.7, as elaborated on in Section 4.6.

4.1 Preliminaries

4.1.1 DPLL vs DPLL($\mathcal{T}$)

DPLL($\mathcal{T}$) is an extension of the DPLL algorithm, used by most propositional SAT solvers, by a theory $\mathcal{T}$. DPLL is illustrated in Figure 4.1, and DPLL($\mathcal{T}$) in Figure 4.2. We give a brief overview of DPLL($\mathcal{T}$) and compare DPLL($\mathcal{T}$) with CEGIS($\mathcal{T}$).

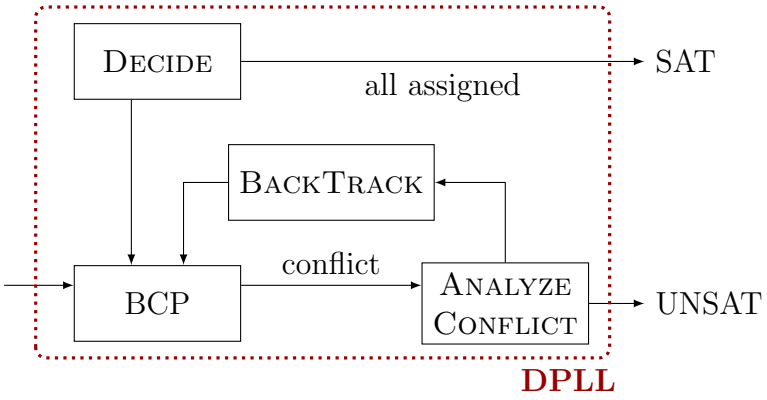


Figure 4.1: DPLL

Given a formula F from a theory $\mathcal{T}$, a propositional formula F_p is created from F in which the theory atoms are replaced by Boolean variables (the "propositional skeleton"). The standard DPLL algorithm, comprising Decide, Boolean Constraint Propagation (BCP), Analyze-Conflict and BackTrack, generates an assignment to the Boolean variables in F_p, as illustrated in Figure 4.1. The theory solver then checks whether this assignment is still consistent when the Boolean variables are replaced by their original atoms. If so, a satisfying assignment for F has been found. Otherwise, a constraint over the Boolean variables in F_p is passed back to Decide, and the process repeats.

In the very first SMT solvers, a full assignment to the Boolean variables was obtained, and then the theory solver returned only a single counterexample, similar to the implementations of CEGIS that are standard now. Such SMT solvers solvers are prone to enumerating all possible counterexamples, and so the key improvement in DPLL($\mathcal{T}$) was the ability to pass back a more general constraint over the variables in the formula as a counterexample [53]. Furthermore, modern variants of DPLL($\mathcal{T}$) call the theory solver on partial assignments to the variables in F_p. Our proposed,

new synthesis algorithm offers equivalents of both of these ideas that have improved DPLL($\mathcal{T}$).

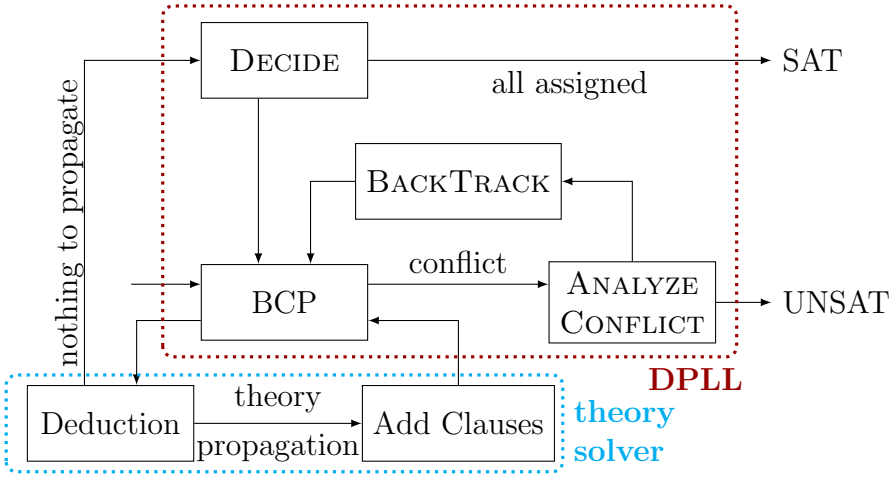


Figure 4.2: DPLL($\mathcal{T}$) with theory propagation

4.1.2 CEGIS

Recall that the program synthesis problem, as defined in Equation 1.1 in Chapter 1.3 we are solving is $\exists P. \forall \vec{x} \in \mathcal{I}. \sigma(\vec{x}, P)$. Briefly recall the architecture of the CounterExample Guided Synthesis algorithm, a paradigm used to solve program synthesis problems, which comprises of a synthesis and a verification phase, illustrated in Figure 4.3. The

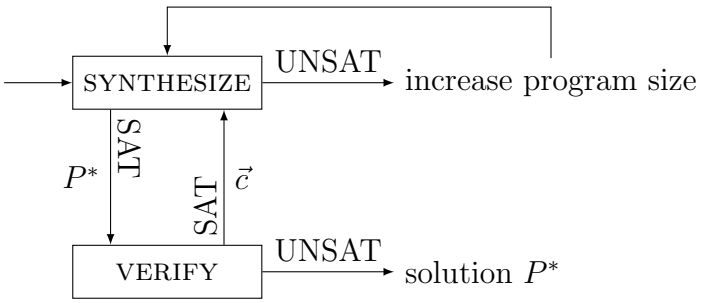


Figure 4.3: Standard CEGIS architecture

synthesis phase solves the formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$, and the verification phase solves the formula $\exists \vec{x} \in \mathcal{I}. \neg\sigma(\vec{x}, P)$. The program size of the program the synthesiser is searching for is restricted to some number of instructions L, which the algorithm initialises to be 1. If the synthesiser is unable to find a solution, there is no candidate program with size $\leq L$ that satisfies the current set of inputs, and we increase the program size. By this method, CEGIS will always find the shortest solution available.

CEGIS can be thought of as using a logical specification and converting that to examples; the input to the algorithm is a formal specification, candidate solutions are generated for that formal specification that hold for a subset of possible inputs $\mathcal{I}_\mathcal{G}$, and then verified across the entire set of inputs $\mathcal{I}$. If the verification fails, it produces a counterexample in the form of an additional input for which the specification fails. This counterexample is effectively an input-output pair, where the output is that the property σ was violated, and is returned to the learning block and the cycle repeats. The learning block is in effect performing program synthesis using input-output pairs.

4.2 Motivating example

In each iteration of a standard CEGIS loop, the communication from the verification phase back to the synthesis phase is restricted to concrete counterexamples. This is particularly detrimental when synthesising programs that require non-trivial constants, i.e., constants that do not otherwise feature in the benchmark. In such a setting, it is typical that a counterexample provided by the verification phase only eliminates a single candidate solution and, consequently, the synthesiser ends up enumerating possible constants.

For illustration, let's consider the trivial problem of synthesising a function $f(x)$ where $f(x) < 0$ if $x < 334455$ and $f(x) = 0$, otherwise. This would be expressed as a SyGuS problem as follows:

```
(set-logic BV)

(synth-fun f (x (BitVec 32)) (BitVec 32))
(declare-var x (BitVec 32))

(constraint (ite (bvult x 334455 ) (bvult (f x) 0) 0))

(check-synth)
```

One possible solution is $f(x) = ite\ (x < 334455)\ {-1}\ 0$, where ite stands for *if then else*.

In order to make the synthesis task even simpler, we are going to assume that we know a part of this solution, namely we know that it must be of the form $f(x) = ite\ (x{<}?)\ {-1}\ 0$, where "?" is a placeholder for the missing constant that we must synthesise. This is an artificial starting point, we can imagine that we are starting part-way through the CEGIS algorithm, i.e., after several iterations.
A plausible scenario for a run of CEGIS is presented next:

- the synthesis phase guesses $f(x) = ite\ (x < 0)\ {-1}\ 0$

- The verification phase then returns $x = 0$ as a counterexample.
- In the next iteration of the CEGIS loop, the synthesis phase guesses $f(x) = ite\ (x < 1)\ {-1}\ 0$ (which works for $x = 0$)
- The verifier then produces $x = 1$ as a counterexample.
- The synthesis phase then guesses $f(x) = ite\ (x < 2)\ {-1}\ 0$
- And so on ...

Following this pattern, the synthesis phase may end up enumerating all the candidates

$$f(x) = ite\ (x < 2)\ {-1}\ 0$$
$$\dots$$
$$f(x) = ite\ (x < 334454)\ {-1}\ 0$$

before finding the solution. This is caused by the fact that each of the concrete counterexamples $0, \dots, 334454$ eliminates one candidate only from the solution space. If we can propagate more information from the verifier to the synthesis phase in each iteration of the CEGIS loop we could avoid this enumeration.

4.3 Architecture of CEGIS($\mathcal{T}$)

In this section, we describe the architecture of CEGIS($\mathcal{T}$), which is obtained by augmenting the standard CEGIS loop with a theory solver. As we are particularly interested in the synthesis of programs with constants, we present CEGIS($\mathcal{T}$) from this particular perspective. In such a setting, CEGIS is responsible for synthesising program skeletons, whereas the theory solver generates constraints over the literals that denote constants. These constraints are then propagated back to the synthesiser. In other words, the synthesis phase, instead of synthesising using input examples, synthesises using logical constraints.

In order to explain the main ideas behind CEGIS($\mathcal{T}$) in more detail, we first differentiate between a *candidate solution*, a *candidate solution skeleton*, a *generalised candidate solution* and a *final solution*.

Definition 1 (Candidate solution)**.** *Using the notation in Section 4.1.2, a program P is a* candidate solution *if it is a satisfiability witness for the formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$ for $\vec{x} \in \mathcal{I}_{\mathcal{G}}$, where $\mathcal{I}_{\mathcal{G}}$ is a subset of all the possible inputs to the program, $\mathcal{I}$.*

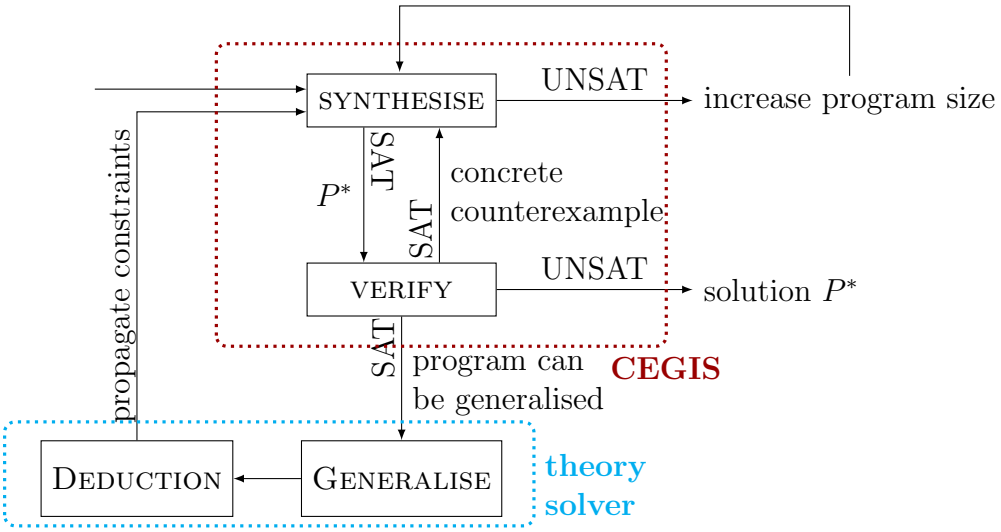


Figure 4.4: CEGIS($\mathcal{T}$)

Definition 2 (Candidate solution skeleton)**.** *Given a candidate solution P^*, the* skeleton *of P^*, denoted by $P^*[?]$, is obtained by replacing each constant in P^* with a hole.*

Definition 3 (Generalised candidate solution)**.** *Given a candidate solution skeleton $P^*[?]$, we obtain a* generalised candidate *$P^*[\vec{v}]$ by filling each hole in $P^*[?]$ with a distinct symbolic variable, i.e., variable v_i will correspond to the i-th hole. Then $\vec{v} = [v_1, \ldots, v_n]$, where n denotes the number of holes in $P^*[?]$.*

Definition 4 (Final solution)**.** *A candidate solution P^* is a* final solution *P if the formula $\forall \vec{x}.\sigma(P, \vec{x})$ is valid when P is replaced with P^*.*

Example 1. *Consider the example in Section 4.2, synthesising a function $f(x)$ where $f(x) < 0$ if $x < 334455$ and $f(x) = 0$, otherwise.*

If $\vec{x}_{inputs} = \{0\}$, then $f(x) = -2$ is a candidate solution.

The corresponding candidate skeleton is $f[?](x) = ?$ and the generalised candidate is $f[v_1](x) = v_1$.

A final solution for this example is $f(x) = ite\ (x < 334455)\ {-1}\ 0$.

The communication between the synthesiser and the theory solver in CEGIS($\mathcal{T}$) is illustrated in Figure 4.4 and can be described as follows:

- The CEGIS architecture (enclosed in a red rectangle) deduces the candidate solution P^*, which is provided to the theory solver.

- The theory solver (enclosed in a blue rectangle) obtains the skeleton $P^*[?]$ of P^* and generalises it to $P^*[\vec{v}]$ in the box marked GENERALISE. Subsequently, DEDUCTION attempts to find a constraint over $\vec{v}$ describing those values for which $P^*[\vec{v}]$ is a final solution. This constraint is propagated back to CEGIS. Whenever there is no valuation of $\vec{v}$ for which $P^*[\vec{v}]$ becomes a final solution, the constraint needs to block the current skeleton $P^*[?]$.

The CEGIS($\mathcal{T}$) algorithm is given as Alg. 2 and proceeds as follows:

- **CEGIS synthesis phase:** checks the satisfiability of $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$and obtains a candidate solution P^*. If this formula is unsatisfiable, then the synthesis problem has no solution for the fixed program length.

- **CEGIS verification phase:** checks whether there exists a concrete counterexample for the current candidate solution by checking the satisfiability of the formula $\neg\sigma(P^*, \vec{x})$. If the result is UNSAT, then P^* is a final solution to the synthesis problem. If the result is SAT, a concrete counterexample $c\vec{e}x$ can be extracted from the satisfying assignment.

- **Theory solver:** if P^* contains constants, then they are eliminated, resulting in the $P^*[?]$ skeleton, which is afterwards generalised to $P^*[\vec{v}]$. The goal of the theory solver is to find $\mathcal{T}$-implied literals and communicate them back to the CEGIS part in the form of a constraint, $C(P, P^*, \vec{v})$. In Alg. 2, this is done by $Deduction(\sigma, P^*[\vec{v}])$. The result of $Deduction(\sigma, P^*[\vec{v}])$ is of the following form: whenever there exists a valuation of $\vec{v}$ for which the current skeleton $P^*[?]$ is a final solution, $res{=}true$ and $C(P, P^*, \vec{v}){=}\bigwedge_{i=1 \cdot n} v_i{=}c_i$, where c_i are constants; otherwise, $res{=}false$ and $C(P, P^*, \vec{v})$ needs to block the current skeleton $P^*[?]$, i.e., $C(P, P^*, \vec{v}){=}P[?]{\neq}P^*[?]$.

- **CEGIS learning phase:** adds new information to the problem specification. If we did not use the theory solver (i.e., the candidate P^* found by the synthesiser did not contain constants or the problem specification was out of the theory solver's scope), then the learning would be limited to adding the concrete counterexample $c\vec{e}x$ obtained from the verification phase to the set $\vec{x}_{inputs}$. However, if the theory solver is used and returns $res{=}true$, then the second element in the tuple contains valuations for $\vec{v}$ such that $P^*[\vec{v}]$ is a final solution.

If *res=false*, then the second element blocks the current skeleton and needs to be added to σ.

Algorithm 2 CEGIS($\mathcal{T}$)

```
1: function CEGIS(T)(specification σ)
2:     while true do
3:         /* CEGIS synthesis phase */
4:         if ∀x⃗_inputs.σ(P, x⃗_inputs) is UNSAT then return Failure;
5:         else
6:             P* = satisfiability witness for ∀x⃗_inputs.σ(P, x⃗_inputs);
7:             /* CEGIS verification phase */
8:             if ¬(σ(P*, x⃗)) is UNSAT then return Final solution P*;
9:             else
10:                cex⃗ = satisfiability witness for ¬(σ(P*, x⃗));
11:                /* Theory solver */
12:                if P* contains constants then
13:                    Obtain P*[?] from P*;
14:                    Generalise P*[?] to P*[v⃗];
15:                    (res, C(P, P*, v⃗)) = Deduction(σ, P*[v⃗]);
16:                end if
17:            end if
18:        end if
19:        /* CEGIS learning phase */
20:        if res then
21:            C(P, P*, v⃗) is of the form ⋀_{i=1·n} v_i = c_i.
22:            return Final solution P*[c⃗];
23:        else
24:            σ(P, x⃗) = σ(P, x⃗) ∧ C(P, P*, v⃗);
25:            x⃗_inputs = x⃗_inputs ∪ {cex⃗};
26:        end if
27:    end while
28: end function
```

4.4 Theory solvers

4.4.1 Fourier-Motzkin for verification

In this section we describe a theory solver based on FM variable elimination. Other techniques for eliminating existentially quantified variables can be used. For instance, one might use cylindrical algebraic decomposition [32] for specifications with non-linear arithmetic. In our case, whenever the specification σ does not belong to linear arithmetic, the FM theory solver is not called.

First recall the Fourier-Motzkin algorithm. FM is an algorithm for eliminating variables from a system of linear inequalities. The basic idea is to heuristically pick a variable from the system of linear inequalities, and eliminate it by projecting its constraints onto the rest of the system, resulting in new constraints. The projection forms a new problem with more constraints but one variable fewer. This process is repeated until we obtain a single variable with constraints.

Example 2. *Consider the following system of inequalities*

$$\begin{aligned} 2x + y + z &\leq 10 \\ x + 2y + z &\geq 10 \\ x + y &\leq 10 \end{aligned}$$

First isolate z:

$$\begin{aligned} z &\leq 10 - 2x - y \\ z &\geq 10 - x - 2y \\ x + y &\leq 10 \end{aligned}$$

Eliminate z:

$$\begin{aligned} 10 - x - 2y &\leq 10 - 2x - y \\ x + y &\leq 10 \end{aligned}$$

Simplify:

$$\begin{aligned} x &\leq y \\ x + y &\leq 10 \end{aligned}$$

We repeat the procedure to eliminate y, first isolating y:

$$\begin{aligned} y &\geq x \\ y &\leq 10 - x \end{aligned}$$

And eliminate y

$$x \leq 5$$

Thus there is a solution for the original system of linear inequalities iff $x \leq 5$.

We would apply this technique in the verification block of CEGIS, in order to produce counterexamples in the form of constraints over program variables, i.e., we wish to produce a constraint over variables $\vec{v}$ describing the situation when $P^*[\vec{v}]$ is a final solution. For this purpose, we consider the formula

$$\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x}),$$

where $\vec{v}$ is a satisfiability witness if the specification σ admits a counterexample $\vec{x}$ for P^*. Let $E(\vec{v})$ be the formula obtained by eliminating $\vec{x}$ from $\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x})$. If $\neg E(\vec{v})$ is satisfiable, any satisfiability witness gives us the necessary valuation for $\vec{v}$:

$$C(P, P^*, \vec{v}) = \bigwedge_{i=1 \cdot n} v_i = c_i.$$

If $\neg E(\vec{v})$ is UNSAT, then the current skeleton $P^*[?]$ needs to be blocked. This reasoning is supported by Lemma 1 and Corollary 1.

Lemma 1. *Let $E(\vec{v})$ be the formula that is obtained by eliminating $\vec{x}$ from $\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x})$. Then, any witness $\vec{v^{\#}}$ to the satisfiability of $\neg E(\vec{v})$ gives us a final solution $P^*[\vec{v^{\#}}]$ to the synthesis problem.*

Proof. From the fact that $E(\vec{v})$ is obtained by eliminating $\vec{x}$ from $\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x})$, we get that $E(\vec{v})$ is equivalent with $\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x})$ (we use $\equiv$ to denote equivalence):

$$E(\vec{v}) \equiv \exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x}).$$

Then:

$$\neg E(\vec{v}) \equiv \forall \vec{x}. \sigma(P^*[\vec{v}], \vec{x}).$$

Consequently, any $\vec{v^{\#}}$ satisfying $\neg E(\vec{v})$ also satisfies $\forall \vec{x}. \sigma(P^*[\vec{v}], \vec{x})$. From $\forall \vec{x}. \sigma(P^*[\vec{v^{\#}}], \vec{x})$ and Definition 4 we get that $P^*[\vec{v^{\#}}]$ is a final solution.

□

Corollary 1. *Let $E(v)$ be the formula that is obtained by eliminating $\vec{x}$ from $\exists \vec{x}. \neg\sigma(P^*[\vec{v}], \vec{x})$. If $\neg E(\vec{v})$ is unsatisfiable, then the corresponding synthesis problem does not admit a solution for the skeleton $P^*[?]$.*

Proof. Given that $\neg E(\vec{v}) \equiv \forall \vec{x}. \sigma(P^*[\vec{v}], \vec{x})$, if $\neg E(\vec{v})$ is unsatisfiable, so is $\forall \vec{x}. \sigma(P^*[\vec{v}], \vec{x})$, meaning that there is no valuation for $\vec{v}$ such that the specification σ is obeyed for all inputs $\vec{x}$. □

For the current skeleton $P^*[?]$, the constraint $E(\vec{v})$ generalises the concrete counterexample $c\vec{e}x$ (found during the CEGIS verification phase) in the sense that the instantiation $\vec{v}^{\#}$ of $\vec{v}$ for which $c\vec{e}x$ failed the specification, i.e., $\neg\sigma(P^*[\vec{v}^{\#}], c\vec{e}x)$, is a satisfiability witness for $E(\vec{v})$. This is true as $E(\vec{v}) \equiv \exists\vec{x}.\neg\sigma(P^*[\vec{v}], \vec{x})$, which means that the satisfiability witness $(\vec{v}^{\#}, c\vec{e}x)$ for $\neg\sigma(P^*[\vec{v}], \vec{x})$, projected on $\vec{v}$, is a satisfiability witness for $E(\vec{v})$.

4.4.1.1 Disjunctions

The specification σ and the candidate solution may contain disjunctions. However, most theory solvers (and in particular the FM variable elimination [23]) work on conjunctive fragments only. A naïve approach could use case-splitting, i.e., transforming the formula into Disjunctive Normal Form (DNF) and then solving each clause separately. This can result in a number of clauses exponential in the size of the original formula. Instead, we handle disjunctions using the Boolean Fourier-Motzkin procedure [82, 140]. As a result, the constraints we generate may be non-clausal.

4.4.1.2 Applying CEGIS($\mathcal{T}$) with FM to the motivational example

We recall the example in Section 4.2 and apply CEGIS($\mathcal{T}$). The problem is

$$\exists f.\forall x.\, x{<}334455 \rightarrow f(x){<}0 \wedge x{\geq}334455 \rightarrow f(x){=}0$$

which gives us the following specification:

$$\sigma(f, x) = (x{\geq}334455 \vee f(x){<}0) \wedge (x{<}334455 \vee f(x){=}0).$$

The first synthesis phase generates the candidate $f^*(x){=}0$ for which the verification phase returns the concrete counterexample $x{=}0$. As this candidate contains the constant 0, we generalise it to $f^*[v_1](x){=}v_1$, for which we get

$$\sigma(f^*[v_1], x) = (x{\geq}334455 \vee v_1{<}0) \wedge (x{<}334455 \vee v_1{=}0).$$

Next, we use FM to eliminate x:

$$\exists x.\neg(\sigma(f^*[v_1], x)) = \exists x.(x{<}334455 \wedge v_1{\geq}0) \vee (x{\geq}334455 \wedge v_1{\neq}0).$$

Note that, given that the formula $\neg\sigma(f^*[v_1], x)$ is in DNF, for convenience we directly apply FM to each disjunct and obtain $E(v_1) = v_1{\geq}0 \vee v_1{\neq}0$, which characterises all the values of v_1 for which there exists a counterexample. When negating $E(v_1)$ we get $v_1{<}0 \wedge v_1{=}0$, which is UNSAT. As there is no valuation of v_1 for which the current

f^* is a final solution, the result returned by the theory solver is $(\mathit{false}, f[?]{\neq}f^*[?])$, which is used to augment the specification. Subsequently, a new CEGIS($\mathcal{T}$) iteration starts. The learning phase has changed the specification σ to

$$\sigma(f,x) = (x{\geq}334455 \vee f(x){<}0) \wedge (x{<}334455 \vee f(x){=}0) \wedge f[?]{\neq}?.$$

This forces the synthesis phase to pick a new candidate solution with a different skeleton. The new candidate solution we get is $f^*(x) = \mathit{ite}\ (x{<}100)\ -3\ 1$, which works for the previous counterexample $x{=}0$. However, the verification phase returns the counterexample $x{=}100$. Again, this candidate contains constants which we replace by symbolic variables, obtaining

$$f^*[v_1,v_2,v_3](x) = \mathit{ite}\ (x{<}v_1)\ v_2\ v_3.$$

Next, we use FM to eliminate x:

$$\begin{aligned}
&\exists x.\neg(\sigma(f^*[v_1,v_2,v_3],x)) = \\
&\exists x.\neg(x{\geq}334455 \vee (x{<}v_1 \rightarrow v_2{<}0 \wedge x{\geq}v_1 \rightarrow v_3{<}0) \wedge \\
&\qquad x{<}334455 \vee (x{<}v_1 \rightarrow v_2{=}0 \wedge x{\geq}v_1 \rightarrow v_3{=}0)) = \\
&\exists x.\neg((x{\geq}334455 \vee x{\geq}v_1 \vee v_2{<}0) \wedge (x{\geq}334455 \vee x{<}v_1 \vee v_3{<}0) \wedge \\
&\qquad (x{<}334455 \vee x{\geq}v_1 \vee v_2{=}0) \wedge (x{<}334455 \vee x{<}v_1 \vee v_3{=}0)) = \\
&\exists x.(x{<}334455 \wedge x{<}v_1 \wedge v_2{\geq}0) \vee (x{<}334455 \wedge x{\geq}v_1 \wedge v_3{\geq}0) \vee \\
&\qquad (x{\geq}334455 \wedge x{<}v_1 \wedge v_2{\neq}0) \vee (x{\geq}334455 \wedge x{\geq}v_1 \wedge v_3{\neq}0).
\end{aligned}$$

As we work with integers, we can rewrite $x{<}334455$ to $x{\leq}334454$ and $x{<}v_1$ to $x{\leq}v_1{-}1$. Then, we obtain the following constraint $E(v_1,v_2,v_3)$ (as aforementioned, we applied FM to each disjunct in $\neg\sigma(f^*[v_1,v_2,v_3],x)$):

$$E(v_1,v_2,v_3) = v_2{\geq}0 \vee (v_1{\leq}334454 \wedge v_3{\geq}0) \vee (v_1{\geq}334456 \wedge v_2{\neq}0) \vee v_3{\neq}0$$

whose negation is

$$\neg E(v_1,v_2,v_3) = v_2{<}0 \wedge (v_1{>}334454 \vee v_3{<}0) \wedge (v_1{<}334456 \vee v_2{=}0) \wedge v_3{=}0$$

A satisfiability witness is $v_1{=}334455$, $v_2{=}-1$ and $v_3{=}0$. Thus, the result returned by the theory solver is $(\mathit{true}, v_1{=}334455 \wedge v_2{=}-1 \wedge v_3{=}0)$, which is used by CEGIS to obtain the final solution

$$f^*(x) = \mathit{ite}\ (x{<}334455)\ {-}1\ 0\ .$$

4.4.2 SMT for generalised verification

For our second variant of a theory solver, we make use of an off-the-shelf SMT solver that supports quantified first-order formulae. This approach is more generic than the one described in Section 4.4.1, as there are solvers for a broad range of theories.

Recall that our goal is to obtain a constraint $C(P, P^*, \vec{v})$ that either characterises the valuations of $\vec{v}$ for which $P^*[\vec{v}]$ is a final solution or blocks $P^*[?]$ whenever no such valuation exists. Consequently, we use the SMT solver to check the satisfiability of the formula

$$\Phi = \forall \vec{x}.\, \sigma(P^*[\vec{v}], \vec{x}).$$

If Φ is satisfiable, then any satisfiability witness $\vec{c}$ gives us a valuation for $\vec{v}$ such that P^* is a final solution: $C(P, P^*, \vec{v}) = \bigwedge_{i=1 \cdot n} v_i = c_i$. Conversely, if Φ is unsatisfiable, then $C(P, P^*, \vec{v})$ must block the current skeleton $P^*[?]$: $C(P, P^*, \vec{v}) = P[?] \neq P^*[?]$.

Note that Φ is still a formula with an alternating quantifier, as there is an implicit quantifier at the beginning of the formula asking if there exists a witness that satisfies the formula:

$$\Phi = \exists \vec{v}.\, \forall \vec{x}.\, \sigma(P^*[\vec{v}], \vec{x}).$$

This is thus still not a trivial formula to solve, and, depending on the propositional structure of the formula, an SMT solver may take an infeasibly long time to solve this formula. To avoid this, we introduce constraints on the variables in $\vec{x}$ to break the formula up into multiple potentially simpler formulas that are solved in parallel, under a short timeout.

For every variable $v_i \in \vec{v}$, we create two formulas:

1. $\Phi \wedge (v_i < c)$
2. $\Phi \wedge (v_i > c)$

We use the same value for c that v_i took in the concrete counterexample generated from the verifier. As a result, we do not need to consider the case where $v_i = c$ because we already know that the specification σ is not satisfied by that value of v_i.

For each variable $v_i \in \vec{v}$ we introduce two formulas, so for a candidate program with n variables in $\vec{v}$, we will solve $2 \cdot n$ formulas in parallel. Consider the case where $\vec{v}$ only contains one variable, and so we solve only the two formulas above. The following outcomes are possible:

- Both formula are unsatisfiable, and we return clauses that block the full skeleton $P^*[?]$

- The solver returns unsatisfiable for formula 1, i.e., there is no valid candidate when $v_i < c$, and so we return a clause $P^*[?] \implies (v_i > c)$
- The solver returns unsatisfiable for formula 2, i.e., there is no valid candidate when $v_i > c$, and so we return a clause $P^*[?] \implies (v_i < c)$
- The solver returns satisfiable for either one of the formula. In this case, the values of $\vec{v}$ found by the solver are a valid solution to the synthesis problem and we return $P^*[\vec{v}]$ as the solution.
- The solver exceeds the timeout for both formulas. In this case, we are forced default to normal CEGIS behaviour for one iteration, and we return only the concrete counterexample.

Note that we only bound one variable per formula, to avoid an exponential increase in the number of formulas we must solve for each variable in $\vec{v}$.

4.4.2.1 Applying SMT-based CEGIS($\mathcal{T}$) to the motivational example

Again, we recall the example in Section 4.2. We will solve it by using SMT-based CEGIS($\mathcal{T}$) for the theory of linear arithmetic. For this purpose, we assume that the synthesis phase finds the same sequence of candidate solutions as in Section 4.2. Namely, the first candidate is $f^*(x)$=0, which gets generalised to $f^*[v_1](x)$=v_1. Then, the first SMT call is for $\forall x.\, \sigma(v_1, x)$, where

$$\sigma(v_1, x) = (x \geq 334455 \vee v_1 < 0) \wedge (x < 334455 \vee v_1 = 0).$$

The SMT solver returns UNSAT, which means that there is no satisfying program with this template, i.e., $C(f, f^*, v_1) = f[?] \neq ?$.
The second candidate is $f^*(x) = ite\ (x < 100)\ {-3}\ 1$, which generalises to $f^*[v_1, v_2, v_3](x) = ite\ (x < v_1)\ v_2\ v_3$. The corresponding call to the SMT solver is for

$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x).$$

As described above, we partition this into the following SMT calls:

$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_1 < 100)$$
$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_1 > 100)$$
$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_2 < -3)$$
$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_2 > -3)$$
$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_3 < 1)$$
$$\forall x.\, \sigma((ite\ (x < v_1)\ v_2\ v_3), x) \wedge (v_3 > 1)$$

The SMT solver returns the satisfiability witness $v_1 = 334455$, $v_2 = -1$ and $v_3 = 0$ from the first formula solved. Then $C(f, f^*, v_1, v_2, v_3) = v_1{=}334455 \wedge v_2{=} -1 \wedge v_3{=}0$, which gives us the same final solution we obtained when using FM in Section 4.2.

4.5 Experimental Evaluation

4.5.1 Benchmarks

As described in Chapter 3, we use a set of bitvector benchmarks from the Syntax-Guided Synthesis (SyGuS) competition [10] and a set of benchmarks synthesising safety invariants and danger invariants for C programs [34]. All benchmarks are written in SyGuS-IF [116], a variant of SMT-LIB2.

Given that the syntactic restrictions (called the *grammar* or the *template*) provided in the SyGuS benchmarks contain all the necessary non-trivial constants, we removed them completely from these benchmarks. Removing just the non-trivial constants and keeping the rest of the grammar (with the only constants being 0 and 1) would have made the problem much more difficult, as the constants would have had to be incrementally constructed by applying the operators available to 0 and 1.

We group the benchmarks into three categories: invariant generation, which covers danger invariants, safety invariants and the class of invariant generation benchmarks from the SyGuS competition; hackers/crypto, which includes benchmarks from hackers-delight and cryptographic circuits; and comparisons, composed of benchmarks that require synthesising longer programs with comparisons, e.g., finding the maximum value of 10 variables. The distribution of benchmarks across these categories is shown in Table 4.1.

category	total	
invariant generation	47	from SyGuS competition and program analysis
comparisons	7	e.g., find the maximum
hacker's	6	from Hacker's Delight [76]
other SyGuS	23	
controller synthesis	-	Not used in this chapter

Table 4.1: Categories of the benchmarks

4.5.2 Experimental Setup

Recall the experimental set up for this dissertation: we conduct the experimental evaluation on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM and Linux OS. We use the Linux *times* command to measure the CPU time used for each benchmark. The runtime is limited to 600 s per benchmark. We use MiniSat [42] as the SAT solver, and Z3 v4.5.1 [37] as the SMT-solver in CEGIS($\mathcal{T}$) with SMT-based theory solver. The SAT solver could, in principle, be replaced with Z3 to solve benchmarks over a broader range of theories.

We present results for four different configurations of CEGIS:

- CEGIS($\mathcal{T}$)-FM: CEGIS($\mathcal{T}$) with Fourier Motzkin as the theory solver;
- CEGIS($\mathcal{T}$)-SMT: CEGIS($\mathcal{T}$) with Z3 as the theory solver;
- CEGIS: basic CEGIS as described in Section 1.3;

We compare our results against CVC4, version 1.5. As we are interested in running our benchmarks without any syntactic template, the first reason for choosing CVC4 [18] as our comparison point is the fact that it performs well when no such templates are provided. This is illustrated by the fact that it won the Conditional Linear Integer Arithmetic track of the SyGuS competition 2017 [10], one of two tracks where a syntactic template was not used. The other track without syntactic templates is the invariant generation track, in which CVC4 was close second to LoopInvGen [107]. A second reason for picking CVC4 is its overall good performance on all benchmarks, whereas LoopInvGen is a solver specialised to invariant generation.

We also give a row of results for a hypothetical multi-core implementation, as would be allowed in the SyGuS Competition, running three configurations in parallel:

Benchmark	multi-core		CEGIS($\mathcal{T}$)-SMT		CEGIS($\mathcal{T}$)-FM		CEGIS		CVC4	
	#	s	#	s	#	s	#	s	#	s
comparisons	1	<0.1	1	<0.1	1	0.6	1	<0.1	7	<0.1
hackers	3	23.7	3	25.4	3	31.0	3	23.7	6	<0.1
inv	25	423.0	25	441.5	15	84.3	19	430.2	6	6.1
other	15	61.4	15	61.4	13	85.1	15	70.0	14	121.2
total solved	44	262.9	44	273.5	32	77.0	38	244.6	33	52.5

Table 4.2: Experimental results – for every set of benchmarks, we give the number of benchmarks solved by each configuration within the timeout and the average time taken per solved benchmark

CEGIS($\mathcal{T}$)-FM, CEGIS($\mathcal{T}$)-SMT and CEGIS. A link to the full experimental environment, including scripts to reproduce the results, all benchmarks and the tool, is provided in the footnote as an Open Virtual Appliance (OVA)[2].

4.5.3 Results

The results as published in [5] are reproduced here. An updated set of results is in Section 4.6.3, which expands the benchmark set and uses an updated version of CVC4. Additionally, it includes the comparison with the CVC4 implementation of CEGIS($\mathcal{T}$).

We evaluate CEGIS($\mathcal{T}$) on the SyGuS and program analysis benchmarks. There are 83 benchmarks in total. Recall the categories we split the benchmarks into, as described in Chapter 3 and reiterated in Table 4.1

The results are given in Table 6.7. In combination, our CEGIS implementations (i.e., CEGIS multi-core) solve 11 more benchmarks than CVC4, but the average time per benchmark is significantly higher.

As expected, both CEGIS($\mathcal{T}$)-SMT and CEGIS($\mathcal{T}$)-FM solve more of the invariant generation benchmarks which require synthesising arbitrary constants than CVC4. Conversely, CVC4 performs better on benchmarks that require synthesising long programs with many comparison operations, e.g., finding the maximum value in a series of numbers. CVC4 solves more of the hackers-delight and cryptographic circuit benchmarks, none of which require constants.

Our implementation of basic CEGIS (and consequently of all configurations built on top of this) only increases the length of the synthesised program when no program of a shorter length exists. Thus, it is expensive to synthesise longer programs. However,

[2] `www.cprover.org/synthesis`

a benefit of this architecture is that the programs we synthesise are the minimum possible length. Many of the expressions synthesised by CVC4 are very large. This has been noted previously in the Syntax-Guided Synthesis Competition [11], and synthesising without the syntactic template causes the expressions synthesised by CVC4 to be even longer.

In order to validate the assumption that CVC4 works better without a template than with one where the non-trivial constants were removed (see Section 4.5.1), we also ran CVC4 on a subset of the benchmarks with a syntactic template comprising the full instruction set we give to CEGIS, plus the constants 0 and 1. Note for some benchmarks it is not possible to add a grammar because the SyGuS-IF language does not allow syntactic templates for benchmarks that use the loop invariant syntax. With a grammar, CVC4 solves fewer of the benchmarks, and takes longer per benchmark. The syntactic template is helpful only in cases where non-trivial constants are needed and the non-trivial constants are contained within the template.

We ran EUSolver on the benchmarks with the syntactic templates, but the bitvector support is incomplete and missing some key operations. As a result EUSolver was unable to solve any benchmarks in the set, and so we have not included the results in the table.

Benefit of literal constants We have investigated how useful the constants in the problem specification are, and have tried a configuration that seeds all constants in the problem specification as hints into the synthesis engine. This proved helpful for basic CEGIS only but not for the CEGIS($\mathcal{T}$) configurations. Our hypothesis is that the latter do not benefit from this because they already have good support for computing constants. We dropped this option in the results presented in this section.

4.6 CVC4

In this section we describe the workings of CVC4, the prominent SyGuS solver that we have used as a comparison point for CEGIS($\mathcal{T}$), and the new implementation of CEGIS($\mathcal{T}$) within CVC4. CVC4 is primarily an SMT-solver. However, in 2015, the CVC4 team introduced the first program synthesis engine implemented *inside* an SMT solver [121]. CVC4 contains several different algorithms that it makes use of for program synthesis, depending on the type of problem it is presented with. Notably, since the publication of our work on CEGIS($\mathcal{T}$), the CVC4 team have implemented the same algorithm inside CVC4.

In this section we first give details of the algorithms that CVC4 version 1.5 uses for program synthesis. This is the version of CVC4 used to produce the results described in Section 4.5.3. We then describe the CVC4 implementation of CEGIS($\mathcal{T}$), and present an updated set of results comparing the performance of this implementation to our own implementation.

In general, CVC4 uses refutation to solve synthesis problems [121]. Given the synthesis problem

$$\exists P.\, \forall \vec{x}.\, \sigma(P, \vec{x})$$

CVC4 tries to establish the unsatisfiability of its negation

$$\forall P.\, \exists \vec{x}.\, \neg\sigma(P, \vec{x}) \tag{2}$$

such that a solution for the first formula can be obtained from the refutation of the second, rather than from the valuation of P in a model of the first formula. The exact method by which this equation is solved depends on whether the problem is *single-invocation* or not.

4.6.1 Single-invocation problems

Some program synthesis problems, whilst they have the syntactic form of a second-order problem, are essentially first-order. Single-invocation problems fall into this category. A synthesis problem is single invocation if, for the problem: $\exists P.\, \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}.\, \sigma(\vec{x}, P)$ all invocations of $P(\vec{x})$ in σ are called with the same arguments $\vec{x}$. For example, consider the task of synthesising a program P which computes the maximum of two inputs x and y:

Example 3.

$$\begin{aligned} \exists P. \forall xy, P(x,y) \geq x \\ \wedge P(x,y) \geq y \wedge \\ (P(x,y) = x \vee P(x,y) = y) \end{aligned}$$

All four invocations of P in this specification are called with the same arguments, x and y, in the same order.

The synthesis formula $\exists P.\, \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}.\, \sigma(\vec{x}, P)$ is logically equivalent to

$$\forall \vec{x} \in \mathcal{I} \exists z\, .\sigma(\vec{x}, z)$$

where z is a variable of the same type as the return type of $P(\vec{x})$. In refutation-based synthesis, instead of trying to solve the formula above, we try to prove the unsatisfiability of the negation:

$$\exists \vec{x} \forall z \neg \sigma(\vec{x}, y).$$

We replace $\vec{x}$ with $\vec{a}$, a tuple of uninterpreted constants of the correct sort:

$$\exists \vec{a} \forall z \neg \sigma(\vec{a}, y).$$

We can then use an SMT solver to find a finite set of instances of $\vec{a}$ such that the formula is unsatisfied. We can repeat this process until we have found all the sets of constants for which the formula is unsatisfied. Provided that the program synthesis grammar allows an if-then-else expression, this is sufficient to solve the synthesis problem. Consider the example above:

Example 4. *We start by negating invocations of $P(x, y)$ with constant integer z. The original constraints are as follows:*

$$\begin{aligned} \neg \exists P. \forall x, y,\ & P(x, y) \geq x \\ & \wedge P(x, y) \geq y \wedge \\ & (P(x, y) = x \vee P(x, y) = y) \end{aligned}$$

Replacing the invocations of $P(x, y)$ with constant integer z:

$$\begin{aligned} \neg \forall x, y \exists z.\ & z \geq x \\ & \wedge z \geq y \wedge \\ & (z = x \vee z = y) \end{aligned}$$

This simplifies to:

$$\exists x, y. \forall z \neg (z \geq x \wedge z \geq y \wedge (z = x \vee z = y))$$

This is now first-order linear integer arithmetic, and solvable by first-order $\forall$-instantiation. We now replace x and y with a tuple of uninterpreted constants of the correct type, which we call $\vec{a}$, where $\vec{a} = \{a_1, a_2\}$. The formula is unsatisfiable when y is instantiated to be a_1 and when y is instantiated to be a_2, giving a solution for the original program synthesis problem that is:

$$P = \begin{cases} y & : \quad y \geq x \\ x & : \quad \textit{otherwise} \end{cases}$$

4.6.1.1 Beyond single-invocation problems

If the synthesis problem is not single invocation, CVC4 applies syntax-guided enumerative techniques to generate candidate programs, in a refinement loop similar to CEGIS, using the first-order solver components of CVC4 for the synthesis and verification phases. CVC4 enumerates through programs based on the syntactic template provided. If no syntactic template is provided, CVC4 will generate a default grammar [122]. This differs from the approach used by our implementation, which uses a less complex approach of encoding the synthesis problem for a given program size and allowing the solver to select candidate programs by assigning to the selector variables in the encoding. CVC4 has a variety of enumeration techniques:

- constraint-based enumeration
- brute-force enumeration
- a hybrid approach combining both brute-force and constraint-based enumeration

4.6.2 Integration of CEGIS($\mathcal{T}$) into CVC4

Recall that CEGIS($\mathcal{T}$) returns counterexample constraints from the verification phase to the synthesis phase which potentially block the candidate skeleton program $P[?]$. Due to the different enumeration techniques used, CVC4 has no way of preventing the synthesis phase from generating new candidates that differ from the skeleton program only by the constant values.

Integration of a first-order solver in the verification phase could potentially, given a candidate program with the correct program skeleton but incorrect constants, generate the correct solution. However, if it is unable to generate a correct solution, the synthesis phase will not be able to make use of the counterexample constraints.

Due to the limitations of this approach, namely that the synthesis phase is not able to make use of the counterexample constraints, the variant of CEGIS($\mathcal{T}$) that has been implemented by CVC4 uses a slight adaptation to the synthesis phase. In this approach, the synthesis phase generates skeleton programs rather than concrete candidate programs. The skeleton is then made concrete by the verification phase. If there does not exist a correct concrete candidate based on the skeleton candidate, the synthesis phase is then able to block the skeleton candidate using the counterexample constraints returned from the verification phase. This implementation is available using the command line `sygus-repair-const`.

Benchmark	CEGIS($\mathcal{T}$)-multicore		CVC4		CVC4-CEGIS($\mathcal{T}$)		CVC4-multi-core	
	#	s	#	s	#	s	#	s
comparisons	1	0.2	7	0.6	1	2551.1	7	0.6
hackers	3	23.7	6	<0.1	0	-	6	<0.1
inv	25	423.0	29	36.5	26	295.6	31	4.1
other	15	61.4	17	93.1	15	276.4	18	95.1
total solved	44	26.3	59	44.8	42	342.4	62	29.7

Table 4.3: Experimental results – for every set of benchmarks, we give the number of benchmarks solved by each configuration within the timeout and the average time taken per solved benchmark

4.6.3 Performance of CEGIS($\mathcal{T}$) as implemented in CVC4

We present the updated results with comparisons with CVC4 version 1.7 in Table 4.3. The CVC4 implementation of CEGIS($\mathcal{T}$) performs substantially better than our implementation. This is unsurprising, as CVC4 is a mature tool containing a portfolio of enumeration techniques. There have also been improvements to the CVC4 default performance in version 1.7, which is now able to solve 59 of the benchmarks. However, we note that the combination of CVC4 and CVC4-CEGIS($\mathcal{T}$) is able to solve 3 benchmarks more than just CVC4, and the average solving time per benchmark is 50% quicker.

4.7 Conclusions

In this chapter CEGIS($\mathcal{T}$), a new approach to program synthesis that combines the strengths of a counterexample-guided inductive synthesiser with those of a theory solver to provide a more efficient exploration of the solution space. We discussed two options for the theory solver, one based on FM variable elimination and one relying on an off-the-shelf SMT solver. Our experiments results showed that, although slower than CVC4 1.5, our implementation of CEGIS($\mathcal{T}$) can solve more benchmarks within a reasonable time that require synthesising arbitrary constants, where CVC4 version 1.5 fails. This result supports our research hypothesis that synthesis of programs containing arbitrary constants is computationally feasible without provision of syntactic templates. Further supporting our hypothesis, CVC4 version 1.7 now contains an integrated implementation of CEGIS($\mathcal{T}$), which proves to be more successful than our implementation of CEGIS($\mathcal{T}$), and a hypothetical multi-core implementation of of

CVC4-CEGIS($\mathcal{T}$) and CVC4's existing single invocation strategies proves to be the most successful solving combination over our benchmark set.

Chapter 5

A Novel Encoding of the Synthesis Problem

Recall that the CEGIS algorithm consists of two phases: a synthesis phase, which guesses candidate programs by solving the formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$ where $\mathcal{I}_{\mathcal{G}}$ is a subset of all possible inputs to the program $\mathcal{I}$; and a verification phase, which verifies whether a candidate program P satisfies the full specification by solving the formula $\exists \vec{x} \in \mathcal{I}. \neg\sigma(\vec{x}, P)$.

To obtain the results reported in this dissertation, we use a propositional SAT solver in the synthesise phase of CEGIS to solve the formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$. In this chapter we give the full details of the encoding of the space of possible programs up to a given length as a Boolean formula. First we formally define what a program is. We then describe how the space of programs has been encoded in previous work [35]. Finally we present a novel encoding, which we used in [5]. We give a comparison with previous encodings of the SAT formula produced and the time taken for a solver to solve the problems in our benchmark set. The new encoding reduces the size of the SAT formula that the synthesis block is required to solve compared to previous work and, as a result, gives a substantial speed up in solving time by two orders of magnitude.

Related work Much work exists on encodings for SAT and SMT solving [6, 110, 131, 132, 154]. However, there is limited literature on how the space of possible programs is encoded for constraint-based program synthesis as the specifics of the encoding is viewed as an implementation detail. A notable exception is the interpreter-based encoding which we describe fully in this chapter [35]. The details of the encoding are often omitted from published work, as the results in this chapter show that encodings have a great impact on solving time.

The general idea behind constraint-based synthesis is to encode the specification and the syntactic program restrictions to a single formula, as we do in our implementation. The encoding must map specific values of the variables in the formula to specific choices of expressions in the candidate program. Jha et al. [72] use an encoding based on these principles, but, differing from our work, the variables encode larger sub-expressions whereas variables in our work encode individual grammatical elements. Jha et al.'s component based synthesis depends on having a syntactic template or some heuristic by which to generate these sub-expressions.

Many solvers such as EUSolver [12] use some enumeration technique whereby they enumerate through potential candidate programs and then use a solver to verify whether the candidate programs are correct or not. By contrast, our work uses a SAT solver to generate the potential candidate programs from the search space, and so the encoding of the search space likely will have greater impact on performance.

Work like SKETCH [134] splits the encoding problem into "sketch generation" and "sketch completion". Human generated program sketches (templates with holes) are translated by a compiler into equivalent SAT formulae such that satisfying models can be mapped to the corresponding choices for holes. The exact details of how this compilation works are omitted.

CVC4 [119] enumerates through candidate solutions by maintaining an evolving set of constraints that prune the search space. The exact details of how the search space is encoded are omitted from published papers.

5.1 Defining the representation of a program

The programs we synthesise are expressed in SyGuS-IF, and do not allow loops or recursion. We use P to denote a program with c input arguments, which is a mapping from a vector of c input arguments of type $type_{IN}$ to a single output of type $type_{OUT}$. That is $P : type_{IN}^{c} \rightarrow type_{OUT}$. In the SyGuS competition benchmarks, all inputs to programs have the same type, and we work with this restriction although it would be possible to extend the techniques presented to programs with multiple input types. We use $input_i$ to denote the value of the i^{th} input argument, and $output$ to denote the value of the output. The output type is either the same type as the input argument type, $type_{IN}$, or a Boolean.

A program may use any operator from the background theory. Throughout this chapter we will use bitvector operators as an exemplar, but the same approach can be applied to Linear Integer Arithmetic. The program may also use any input parameter

from the vector of c input arguments passed to the program, and any literal constant of the type $type_{IN}$.

The full set of bitvector operators we use is given in Figure 5.1. In order to make this chapter more readable, we present the remainder of the chapter using unsigned operators only, which we denote using the more conventional mathematical symbols with infix notation instead of prefix notation, e.g., we write $x \leq y$ or $x <= y$ instead of `(bvule x y)`.

Note that some bitvector operators exist in SMT-Lib which we have not included in the set of operators we construct the programs from, for example `concat`, which concatenates two bitvectors together. The operators we omitted are syntactic sugar and can be recreated using a combination of bit shifts and logical operations. There is a trade-off in CEGIS between the number of operations in the vocabulary we construct programs from and the synthesis time. Adding more operators increases the number of possible programs and thus the size of the formula to be solved by the synthesis phase, and the time it would take the SAT solver to solve the formula. However, introducing more "syntactic sugar" operators may mean that a shorter program is a correct solution to the synthesis problem, and our CEGIS implementation searches for shorter programs first and so may hit on the solution at an earlier iteration. Many synthesis tools exploit this trade-off by introducing further custom operations that are equivalent to combinations of other operators, for example, we might add an operator that computes the minimum of a set of numbers. We feel that omitting these operators does not present a challenge for our research hypothesis, that synthesis without a syntactic template is computationally feasible, as omitting these operators has not reduced the expressivity of the programs we can synthesise and we use the same set of operators for all SyGuS-IF benchmarks.

Definition 5. *The syntax of a program in bitvector arithmetic [82], shown here for bitvectors of width 8, that returns a bitvector or a Boolean, is thus defined by the following rules. The equivalent syntaxes for programs manipulating different size bitvectors can be obtained by adjusting the definition for a constant literal to the required bitvector width:*

```
Program P :: P ∧ P | ¬P | (P) | atom
     atom :: term rel term | Boolean-Identifier
      rel :: < | = | ≤
     term :: term op term | ∼term | constant |
                          atom?term:term | input
    input ::  input0 | input1 | ... | input c
 constant :: 00000000 | ... | 11111111
       op :: + | - | . | / | << | >> | & | | | ⊗
```

```
Core operations:

(bvnot (_ BitVec m) (_ BitVec m))
       'bitwise negation'
(bvand (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'bitwise and'
(bvor (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'bitwise or'
(bvneg (_ BitVec m) (_ BitVec m))
       '2s complement unary minus'
(bvadd (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'addition modulo 2^m'
(bvmul (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'multiplication modulo 2^m'
(bvudiv (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'unsigned division, truncating towards 0'

(bvshl (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'shift left (equivalent to multiplication by 2^x where x is the
          value of the second argument)'
(bvlshr (_ BitVec m) (_ BitVec m) (_ BitVec m))
       'logical shift right (equivalent to unsigned division by 2^x
          where x is the value of the second argument)'

(bvult (_ BitVec m) (_ BitVec m) Bool)
       'binary predicate for unsigned less-than'

(= (_ BitVec m) (_ BitVec m) Bool)
      returns true iff its two arguments are identical

(ite Bool (_ BitVec m) (_ BitVec m) (_ BitVec m))
      returns its second argument or its third depending on whether
      its first argument is true or not

Syntactic sugar:

(bvxor (_ BitVec s) (_ BitVec t) (_ BitVec m))
      'abbreviates (bvor (bvand s (bvnot t)) (bvand (bvnot s)t))'
(bvule (_ BitVec s) (_ BitVec t) Bool)
      'abbreviates (or (bvult s t) (= s t))'
```

Figure 5.1: bitvector operations used in this dissertation, defined in SMT-Lib [19]

Similar syntax can be defined for programs in Linear Integer Arithmetic.

5.2 Encoding a program using an interpreter

In this section we present the encoding used in [35], which we use as the comparison point for our novel encoding presented in Section 5. A natural way of encoding a finite-state program such as the ones we wish to synthesise, is a sequence of instructions, in a RISC-like representation. An example program in ARM 32-bit RISC, which multiplies two numbers together, is given below:

```
1 LDR R1, [4]
2 LDR R2, [8]
3 MUL R1, R2
4 STR [4], R1
```

Each instruction is comprised of an opcode, e.g., `LDR`, which specifies the operation to be performed and a sequence of operands, which are either registers or locations in memory on which the operation is performed. Registers and memory locations may be re-used any number of times. The program above loads a value located in memory position 4 into register $R1$, then loads the value located in memory location 8 into register $R2$, multiplies them together and stores the result in memory location 4.

RISC programs are typically encoded into a machine instruction set, which are usually fixed-length bitvectors. This bitvector comprises of an opcode, which encodes the mnemonic, fields for the indices of the operand registers, and the immediate constants, if any. In [35], an instruction encoding is used which strongly resembles that of a typical RISC machine instruction set, except that the instructions are pre-split into opcode, operand indices, and constants, and a separate array is used for each. In addition, the language only has registers and no memory, and so does not need the load or store instructions. The following structure is used to represent a program of length `L` with `c` inputs and `k` constants, is thus:

```
1 struct programt
2 {
3   op_t ops[L]; // op codes
4   typet params[l*2] // operands
5   typet inputs[c] // program inputs
6   typet program_constant[k] // constants
7 }
```

The operands refer to indices of registers and the values of program input arguments. The outputs of previous program instructions are written into these registers. As above, an operand may be any one of the input arguments passed to the programs, or a constant literal, or the result of a previous instruction. In contrast to the RISC

assembler, the encoding does not use a separate memory, i.e., there are no load or store instructions.

In previous work [35] this encoding has been shown to be optimal with respect to the number of bits used for storing the encoding. Whilst this may be the case for an encoding based on instruction sets and an interpreter, it nevertheless is not optimal with respect to the size of the SAT formula that is produced for synthesis and verification.

The encoding of the candidate program is only part of the problem. We will now describe how the synthesis and verification stages of the CEGIS algorithm operate using this instruction set encoding, and how the SAT formulae are generated. Our experiments show that the new encoding we introduce in this dissertation results in smaller SAT formulae being solved by the synthesis and verification stages of CEGIS, and thus a shorter synthesis time.

Bounded Model Checking We give a brief overview of Bounded Model Checking (BMC), which is a technique used for proving or refuting properties about transition systems, and is used for synthesis and verification in CEGIS when using the interpreter-based encoding. Usually these properties take the form of safety/liveness or reachability properties, i.e., a certain state or set of states is never reached, or a certain state or set of states is reached. Bounded Model Checking has been successfully applied to the verification of programs [83].

Bounded Model Checking constructs a Boolean formula that represents an unwinding of the transition system over a finite number of steps, and uses a propositional SAT or SMT solver to determine if the property holds. If the property does not hold, a counterexample is generated in the form of a trace that violates the property. BMC is a fast and scalable technique in comparison to previous symbolic model checking techniques based on symbolic manipulation of binary decision diagrams. It is sound, in that any counterexample generated is guaranteed to be a genuine counterexample. It is incomplete because the transition system is unwound only a *finite* number of steps, but, because the programs we are synthesising are finite, loop-free and recursion-free, and the list of counterexample inputs obtained is also finite, it is complete for our purposes.

In bounded model checking for software, a property is typically given as an assertion by the user and BMC constructs a transition system that represents the control flow of the program. Instead of bounding the total number of steps, BMC for software places a bound on the number of loop iterations and recursive calls.

The Boolean formula constructed represents the states reachable within this bound together with the negation of the property. If the formula is satisfiable, then there exists a counterexample trace through the program that violates the property. In the case of CEGIS synthesis, we use bounded model checking on Algorithm 3, and the candidate program is extracted from the counterexample trace that violates the assertion. Assumptions are used in bounded model checking for software to restrict non-deterministic choices. In the case of the synthesis algorithm, the assumptions restrict the non-deterministic choice of candidate program P to be ones which satisfy the property σ.

5.2.1 Synthesis using an interpreter

In this section we describe how the encoding above is used within CEGIS as in [35].

5.2.1.1 Synthesis

We perform synthesis by constructing a Boolean formula, which, if satisfied, will give us a candidate program in the form of the satisfiability witness. The formula we are aiming to build is $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$ for a given program length and set of inputs $\mathcal{I}_{\mathcal{G}}$. In the case where the program is encoded using an instruction set, we need a method to convert the program encoded as sequence of instructions into a formula, which will involve some kind of interpreter which can identify the operator that corresponds to an op code and then deduces what the outcome of applying that operator to the operands would be.

The method used in previous work [35] is to build an interpreter for the program in C code and to perform bounded model checking of the algorithm given in Algorithm 3. The bounded model checker produces a Boolean formula, which is then solved by a SAT solver, and results in a candidate program that can be extracted from the satisfiability witness. In Algorithm 3, the interpreter is the function called `exec`. The function that we perform bounded model checking on takes as input a program P encoded using our encoding. Recall the synthesis formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$. The function calls the interpreter multiple times, once for every input $\vec{x} \in \mathcal{I}_{\mathcal{G}}$, and contains an assumption that fails if P fails on any of the subset of inputs we are trying to synthesise a program for at this iteration.

Recall that the program is represented as a list of instructions, each one comprised of an op code, which identifies an operator, and two operand registers. The interpreter is an algorithm that iterates over the list of instructions and performs the corresponding

Algorithm 3 Synthesis of candidate solution using an interpreter

```
1: function synthesise(I_G, L)
2:     prog P = nondet(L);          ▷ encoding for nondet program of fixed length L
3:     for x⃗ ∈ I_G do
4:         out = exec(P, x⃗)                                  ▷ run interpreter on P
5:         assume(check_sigma(x⃗, out));
6:     end for
7:     assert(0)
8: end function
```

operation on the operands. The function *check_sigma* checks that the property σ holds for the given input, $\vec{x}$, and output *out*.

Size of formula generated The formula generated by bounded model checking will encompass all the paths in the function given in Algorithm 3. This will initialise the variable P with a non-deterministic choice of program, the unwind the loop N times, each time passing a new input to the program and verifying that the output produced by the program satisfies the requirement σ. Although the only non-determinism in the formula should be the program, this is still potentially a larger formula than necessary:

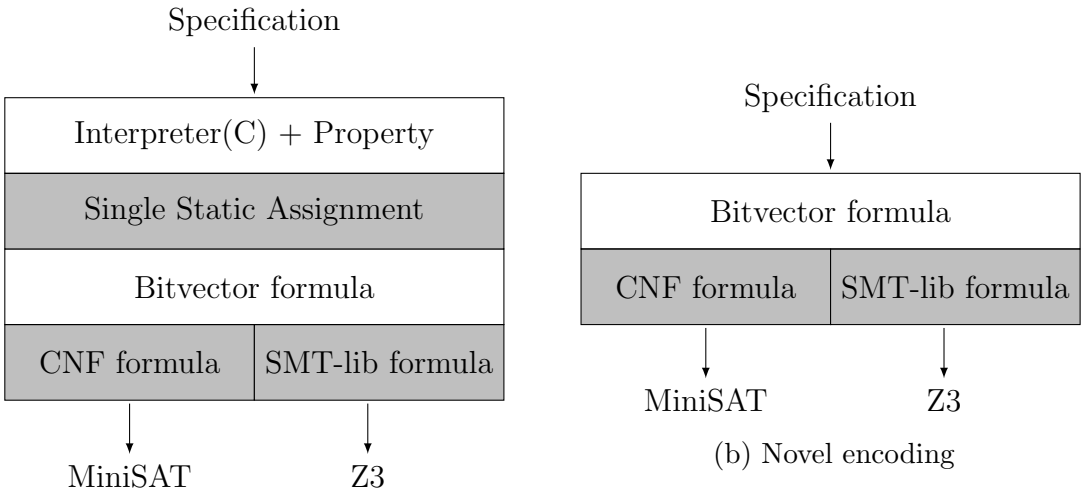
- The program P is represented as an array of op codes, and operands, which will use some fixed number of bits b to represent each number, and unless we have precisely 2^b operands some redundant bits will be introduced.
- Some instructions return Boolean values, which will be stored in the same type as a bitvector.
- Some combinations of op code and operands are redundant; for instance we might nondeterministically choose an instruction that performs $input_1 - input_1$. It is possible to add constraints to remove these redundant combinations but at the cost of increasing the size of the Boolean formula to be solved.

5.2.1.2 Verification

Similarly the verification block of the CEGIS loop uses bounded model checking, and verifies a program that takes the candidate program P with nondeterministic inputs and contains an assertion that is violated if the candidate program does not satisfy the specification for some input. The bounded model checker will return a counterexample, if it exists, that is an input for which the program fails the specification. The C program used in the verification block is given in Algorithm 4.

Algorithm 4 Verify a candidate solution using an interpreter

```
1: function verify(programP)
2:     x⃗ = nondet()                    ▷ non-deterministic assignment to input x⃗
3:     out = exec(P, x⃗);                ▷ run interpreter on P
4:     assert(check_sigma(x⃗, out));
5: end function
```



(a) Interpreter-based encoding

(b) Novel encoding

Figure 5.2: Layers of the interpreter-based and novel encoding

5.3 A Novel Encoding

In this section we present the novel encoding introduced in this dissertation. This encoding significantly reduces the size of the SAT formulae produced for the synthesis and verification stages of CEGIS, and thus speeds up the total solving time.

Our programs use bitvector operators, with a total set of possible operators as set out in Figure 5.1. The total space of programs that can be represented that use bitvector arithmetic can be expressed as a grammar [82], as shown in Definition 5. We stress that this grammar is not the same as a syntactic template used in the SyGuS competition, as a syntactic template restricts the search space of the programs by expressing only a small subset of this grammar.

In contrast to the interpreter-based encoding presented previously, we will show how we use this grammar to efficiently generate a formula, which we can pass directly to an SMT or SAT solver without need for an interpreter or use of bounded model checking, shown in Figure 5.2b. This method generates smaller formula, which can be solved an order of magnitude faster than the formula generated by the interpreter-based encoding.

The first step is to introduce *selector variables*. The selector variables perform the function of the opcodes in the encoding using an interpreter, that is, they select which of the operations should be performed and on which operands. To illustrate this encoding style, we will use an exemplar with a significantly reduced grammar, so we will restrict the possible operations to + and −. We do this solely to increase readability, and all of our experimental work is done with the full set of operations as shown in Figure 5.1. In essence we use a direct encoding of the set of operators as opposed to the binary encoding used by [35].

In our exemplar we consider synthesising a program that takes two input arguments, which are bitvectors of width two, returns a bitvector, and uses a reduced syntax that only contains the operations + and −:

```
Program P :: constant | constant op constant
     term :: term op term | input | constant
    input :: input_0 | input_1
 constant :: | 00 | 01 | 11 | 10
       op :: + | -
```

Let us use these rules to construct a formula that encodes all syntactically correct programs of thee instructions long.
We use I_0 to denote the variable that holds the value of the first instruction. Following the rules above, that may be either of the input arguments, or a constant. We introduce two selector variables, denoted sel_0 and sel_1, to select between these options, and a new variable C_0 to denote the value of the constant.

$$I_0 = \begin{cases} input_1 & : \quad sel_1 \\ input_0 & : \quad sel_0 \\ C_0 & : \quad \text{otherwise} \end{cases}$$

The possible values that C_0 can take are given by the following equation. Note this isn't a restriction as it allows all possible values that the bitvector can represent, and we can omit this clause from the Boolean formula we give to the SAT solver:

$$C_0 = 00 \vee C_0 = 10 \vee C_0 = 11 \vee C_0 = 01$$

We now have one further value, the output of instruction 0, and so instruction 1 can either build a new expression from either of the input values or a constant, or it can apply an operator to the expression we have before. We have two operators to choose from, + or −. There's no point applying a − to two identical operands because that

gives zero, which we could use a constant for, so we omit that option. This is a key difference between the novel encoding presented here and the interpreter-based encoding presented in the previous section; optimisation is done when constructing the formula, instead of by adding constraints after constructing the formula. We introduce three new selector variables to choose between these options.

$$I_1 = \begin{cases} I_0 + I_0 & : \quad sel_2 \\ input_1 & : \quad sel_3 \\ input_0 & : \quad sel_4 \\ C & : \quad \text{otherwise} \end{cases}$$

Finally, instruction 3 has full use of all of the operators and introduces 5 selector variables.

$$I_2 = \begin{cases} I_1 - I_0 & : \quad sel_5 \\ I_0 - I_1 & : \quad sel_6 \\ I_0 + I_1 & : \quad sel_7 \\ I_1 + I_1 & : \quad sel_8 \\ I_0 + I_0 & : \quad sel_9 \\ input_1 & : \quad sel_1 0 \\ input_0 & : \quad sel_1 1 \\ C & : \quad \text{otherwise} \end{cases}$$

A syntactically correct program of four instructions long can be extracted from any assignment for the formula

$$I_0 \wedge I_1 \wedge I_2 \wedge I_3.$$

5.3.0.1 Program length

Recall from Section 1.3 that our CEGIS implementation begins by searching initially for a solution in the space of programs one instruction long. Thus initially, the synthesis encoding only needs to encode the first instruction. If no solution exists within this encoding, we then extend the synthesis encoding to two instructions long, and so on. As a result, we are guaranteed to find the shortest solution that satisfies the specification. This property is true of both the novel encoding and the interpreter-based encoding.

5.3.1 CEGIS with the novel program encoding

5.3.1.1 Synthesis

Recall we are building the formula $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$. It is quite straight forward to substitute the formula $I_0 \wedge I_1 \wedge I_2 \wedge I_3$ into this equation. $\vec{x} \in \mathcal{I}_{\mathcal{G}}$ is a vector of all the counterexample inputs generated by the verification block so far. We initiate the loop with a single input as a vector of 0's, e.g., for a program which takes two inputs $\vec{x} = 0, 0$. We skolemise the existential quantifier and expand the universal quantifier, so for n counterexamples this becomes:

$$\sigma(P, \vec{x}_1) \wedge \sigma(P, \vec{x}_2) \wedge \ldots \wedge \sigma(P, \vec{x}_n),$$

where $P = I_0 \wedge I_1 \wedge I_2 \wedge I_3$.

Size of the formula generated The formula generated here is smaller than the formula generated by the interpreter-based encoding:

- We introduce fewer selector variables than the interpreter-based encoding.
- We can remove some sources of redundancy by simply not encoding the option to, for instance, have the instruction $input_1 - input_1$. Other examples of redundancy include commutativity, for instance $input_1 + input_2$ is equivalent to $input_2 + input_1$. By contrast, the interpreter-based encoding must add assumptions to rule out these combinations after the formula is constructed.

The instruction set encoding, however, is able to re-use registers any number of times; our novel encoding cannot reuse any variables in this way. Our novel encoding can be thought of as similar to single static assignment(SSA) form [123], an intermediate representation used in compilers where each variable is assigned exactly once.

5.3.1.2 Verification

The verification formula is $\exists \vec{x} \in \mathcal{I}. \neg\sigma(\vec{x}, P)$. We take the candidate program from the previous synthesis step, and leave $\vec{x}$, the inputs to the program to be determined by the solver. We solve the formula $\exists \vec{x} \in \mathcal{I}. \neg\sigma(\vec{x}, P)$ with a SAT solver.

category	total	
invariant generation	-	not used in this chapter
comparisons	7	e.g., find the maximum
hacker's	6	from Hacker's Delight [76]
other SyGuS	23	
controller synthesis	29	6 time discretisations per benchmark

Table 5.1: Categories of the benchmarks . We do not use the invariant generation benchmarks in this chapter.

5.4 Experimental comparison

5.4.1 Experimental setup

Recall the experimental set up for this dissertation: we conduct the experimental evaluation on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM and Linux OS. We use the Linux *times* command to measure CPU time used for each benchmark. The runtime is limited to 3600 s per benchmark. We use MiniSat [42] as the SAT solver, version 2.2.1.

We compare the novel encoding with the encoding from [35] on a set of benchmarks taken from the Syntax Guided Synthesis Competition. The benchmarks are distributed across the categories shown in Table 5.1. We note that our encodings generated by different implementations and, whilst they use many of the same building blocks for the synthesis procedure, it was necessary to alter a significant amount of code in order to implement the new encoding. It is thus hard to compare how much of the difference in results is due to the encoding and how much is due to the new sections of implementation that it was necessary to change. We compare our encodings on both the Syntax Guided Synthesis benchmarks and the controller synthesis benchmarks. The controller synthesis benchmarks only require synthesis of constant values, and not of a program or expression, and so do not use the synthesis encoding. Any speed-up obtained on the controller synthesis benchmarks must thus be down to other changes in the implementation that are independent of the exact program encoding. This allows us to approximate how much of the speed-up in solving time is due to the program encoding and how much is due to other implementation changes.

In order to simplify comparison of the size of the formula generated by each encoding, we present a comparison of the number of variables in the formula as outputted by MiniSAT, and ignore the number of clauses.

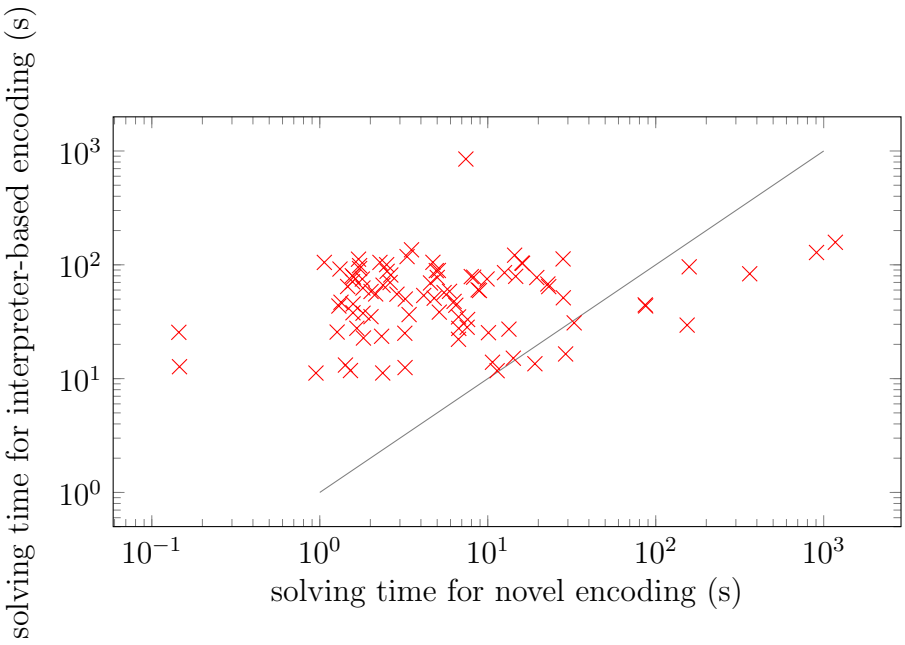


Figure 5.3: Solving time using the interpreter-based encoding and the novel encoding on the controller synthesis benchmarks. Points that fall above the grey line indicate benchmarks that were solved faster by the novel encoding. The average solving time is 68.3 s per bechmark for the interpreter-based encoding and 38.6 s for the novel encoding.

5.4.2 Controller synthesis results

The first comparisons we do are on the set of benchmarks that synthesise controllers for Linear Time Invariant systems, as detailed in Chapter 3. Recall that these benchmarks require synthesis of a matrix of fixed-point values, and as such do not require the encoding for the program to be synthesised. Thus, any speed improvement will be due to the change in representation of the counterexamples, and any miscellaneous improvement in implementation. The solving times for both encodings on the controller benchmarks are shown in Figure 5.3. The average solving time for the interpreter-based encoding is 68.3 s, and the average solving time for the novel encoding is 38.6 s. The average number of variables in the final synthesis formula for the novel encoding is $145,263$, and for the interpreter-based encoding $161,138$, i.e., the novel encoding formulae are 90% of the size of the interpreter-based encoding formulae. In fact, this difference includes one anomalous result. This difference is not substantial, and Figure 5.4 shows that the formula size is actually smaller for many of the interpreter-based encoding runs. This suggests that the formula size is not the key contributor for the difference in solving time here.

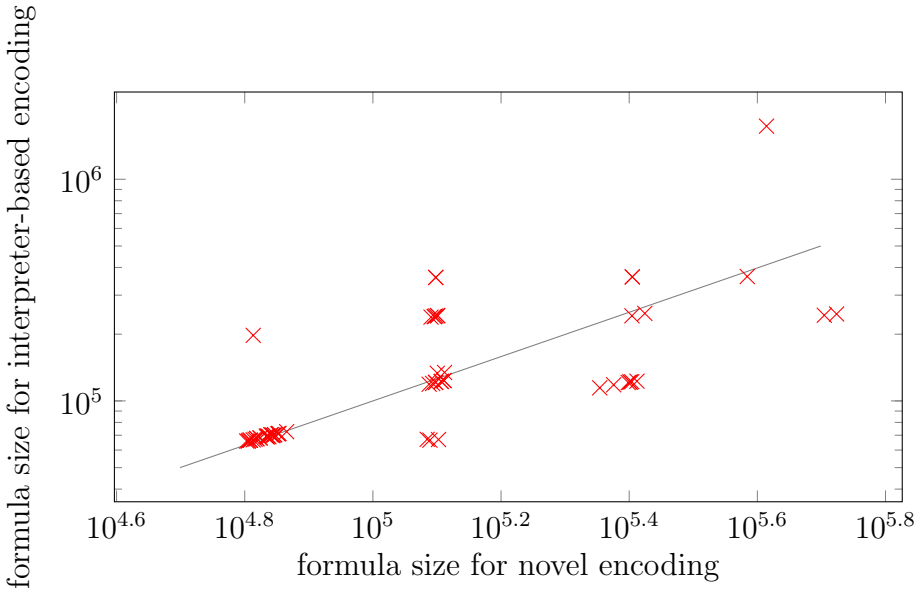


Figure 5.4: Comparison of size of the final synthesis formula for the novel encoding vs the interpreter-based encoding on the controller synthesis benchmarks. Points that fall above the grey line indicate benchmarks where the novel encoding produced a smaller formula.

Benchmark	comparisons		hackers		inv		other		total	
	#	s	#	s	#	s	#	s	#	s
interpreter-based encoding	2	256.5	3	50.7	-	-	9	339.3	14	265.6
novel encoding	1	<0.1	3	23.7	-	-	13	6.5	17	9.2
total benchmarks	7	-	6	-	0	-	16	-	29	-

Table 5.2: Results summary for interpreter-based encoding

5.4.3 SyGuS benchmarks

We compare the old, interpreter-based encoding and the novel encoding on benchmarks from the Syntax Guided Synthesis Competition. On a subset of 29 SyGuS benchmarks, the old encoding solves 14 in an average time of 265.6 s per benchmark, and the new encoding solves 17 in an average time of 9.2 s per benchmark. The results are given in Table 5.2. In Figure 5.5 we give the comparison of run times for the 14 benchmarks solved by both encodings. In Figure 5.6 we give the number of variables in the final synthesis formula for each encoding. The number of variables needed by the novel encoding is substantially lower than the interpreter-based encoding, which is likely the reason for the significant speed up in solving time.

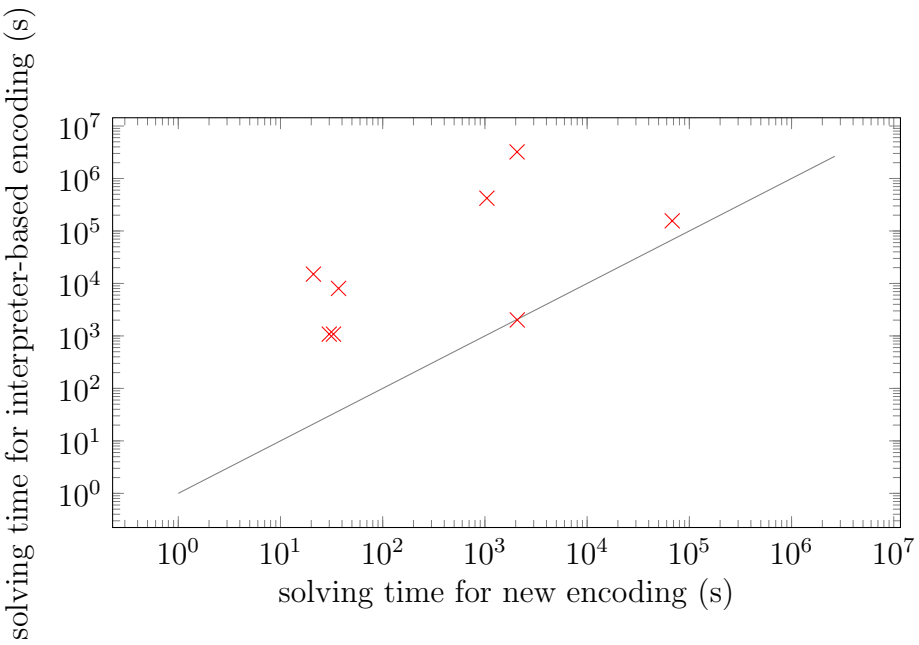


Figure 5.5: Solving time using the interpreter-based encoding and the novel encoding on the SyGuS benchmarks. Points that fall above the grey line indicate benchmarks that were solved faster by the novel encoding. The average solving time is 265.6 s per bechmark for the interpreter-based encoding and 9.2 s for the novel encoding.

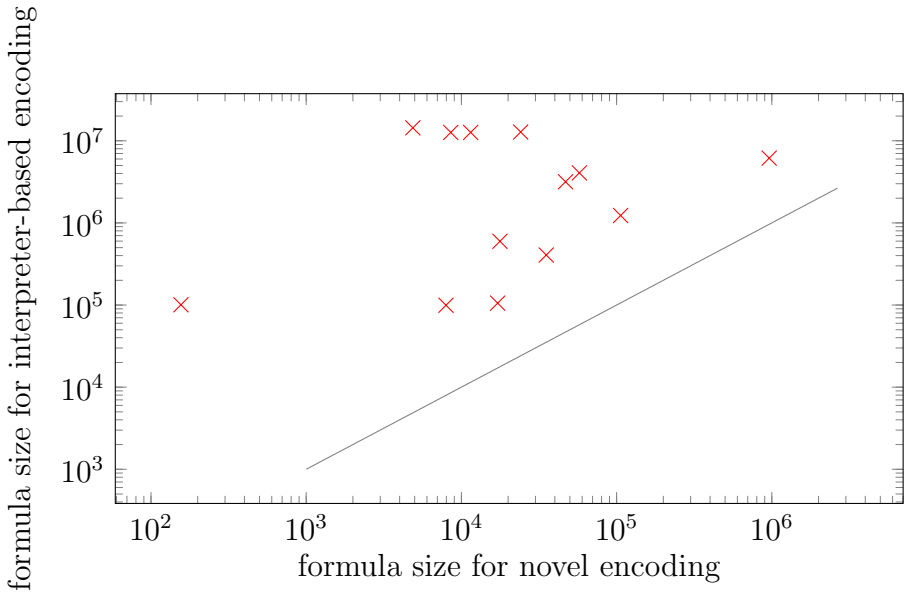


Figure 5.6: Comparison of size of the final synthesis formula for the novel encoding vs the interpreter-based encoding on the SyGuS benchmarks. Points that fall above the grey line indicate benchmarks where the novel encoding produced a smaller formula.

5.5 Conclusions

The novel encoding presented in this chapter reduces solving time by more than two orders of magnitude on the set of SyGuS benchmarks it is presented on. We are only able to compare on a small number of benchmarks, since these are the only ones which are solved by the interpreter-based encoding, and the others timed-out. Nevertheless the speed difference on these benchmarks is so significant, we are confident that the novel encoding provides a substantial speed-up in solving. Our research hypothesis is that synthesis of programs with arbitrary constants is computationally feasible without user-provided syntactic guidance. Whilst the novel encoding does not specifically change the way CEGIS handles constant synthesis, the two-orders of magnitude speed-up contributes to making synthesis of such programs practically possible within a reasonable time-out.

The novel encoding is able to solve some benchmarks that the interpreter-based encoding cannot solve. These benchmarks are the benchmarks that the novel encoding takes longest to solve, either due to them requiring longer solutions or solutions requiring more challenging formula to be solved by the SAT solver (categorising which types of formua are harder for SAT solvers is beyond the scope of this dissertation, but it is sufficient to say that formula involving non-linear operations such as division or bit-shifts are generally more challenging). The interpreter-based encoding simply could not solve these benchmarks within the time limit.

The speed-up exhibited on the controller synthesis benchmarks is only $2x$, which is significantly less than the $100x$ we see on the SyGuS benchmarks. We have found that the novel encoding, as expected, does not reduce the size of the formulae produced for the control benchmarks (which do not synthesise expressions, only constants). Thus, we can deduce that some of the speed-up shown is due to other factors. This could be due to the time the interpreter-based encoding spends translating from C to a bitvector formula to a CNF formula, which is done ever iteration, as illustrated in Figure 5.2a and Fig 5.2b.

We conclude the novel encoding is an improvement on all benchmarks where we are required to synthesize expressions, i.e., any benchmark that requires us to synthesise a program more than one instruction long.

Chapter 6

Incremental first-order solvers in CEGIS

In this chapter we discuss the use of incremental first-order solving techniques in CEGIS. Specifically we investigate use of an incremental satisfiability solver in the synthesis block of a CEGIS loop. The work in this chapter was designed and implemented for a paper published at CAV 2018 [5], although the work is a only a minor contribution to that paper and is elaborated on further here. We evaluate the benefits of incremental first-order solving on the full set of benchmarks. Our research hypothesis is that synthesis of programs containing arbitrary constants is computationally feasible without user-given syntactic guidance. However, if implemented according to the state-of-the-art before our contribution, CEGIS exceeds the time-out threshold solving a large proportion of the SyGuS benchmarks. Incremental first-order solving has provided a speed-up of 3.5 in verification applications [127], and we approached this work speculating that the use of incremental first-order solvers in CEGIS may provide a similar increase in solving time for program synthesis and enable CEGIS to solve a larger number of SyGuS benchmarks. We find that, whilst incremental provides speed ups in some cases, the speed up is not uniform across all benchmarks. However, we find that the addition of incremental SAT to the multi-core engine, described in Chapter 4, does increase the number of benchmarks the multi-core engine is able to solve as well as the reducing the average solving time.

Related work We remind the reader that the work that broadly relates to the material in this dissertation is elaborated on in Chapter 2; we now make an explicit claim why our contribution is novel when compared to related work that is closest to our Incremental solving in verification is a well established area of research. Incremental first-order solvers have been used in bounded model checking [127] for verification

of automotive software. Here incremental satisfiability solving is used when bounds on loops in the code are increased. Compared to non-incremental bounded model checking on automotive software, incremental bounded model checking was found to be between 3.5 times faster. The ratio of time spent in synthesis and verification depends on the problem that CEGIS is given. In general the synthesis step will take more time than verification, and in [35] the synthesis is reported to take 86% of the total time, where verification takes 14% and the outer CEGIS loop takes 7%. We might expect that use of incremental bounded model checking in the synthesis engine gives a speed-up of 3.5 on the synthesis step, resulting in an over-all speed up of approximately 2.

The IC3 model checking algorithm is another example of use of incremental first-order solving in verification. IC3 over-approximates an invariant that implies a property; it refines the candidate invariant by asking a SAT solver if there is a state within the candidate invariant that can violate the property in one step; if the answer is yes, clauses are added that remove the state, and the process is repeated. The calls made to the SAT solver are hence very similar, as the candidate invariant is incrementally updated and all other clauses remain the same. If we implemented IC3 using a non-incremental SAT solver, every time we made a SAT call the solver would start from the beginning of the algorithm. Hence, successful implementations of IC3 use incremental SAT solving [26], which effectively is able to re-use information from previous SAT calls, i.e., the conflict clauses we have learnt, and possibly any heuristic parameters.

CVC4 uses incremental solving for solving SMT problems [20], making use of CaDiCaL [44] as the back-end SAT solver. It is not clear whether CVC4 makes use of incremental solving for synthesis problems.

6.1 Preliminaries

6.1.1 Propositional SAT solving with CDCL

Conflict Driven Clause Learning (CDCL) is an extension of DPLL, the original algorithm used by satisfiability solvers which was presented in Chapter 4, and recalled here in Figure 6.1.

CDCL differs from DPLL only in that the back tracking is non-chronological. In DPLL, if assigning a value to a variable causes a conflict, the algorithm backtracks to the immediately preceding level and assigns the opposite value to the variable. This can cause a chain of chronological back-tracking. In CDCL, if there is a conflict, the

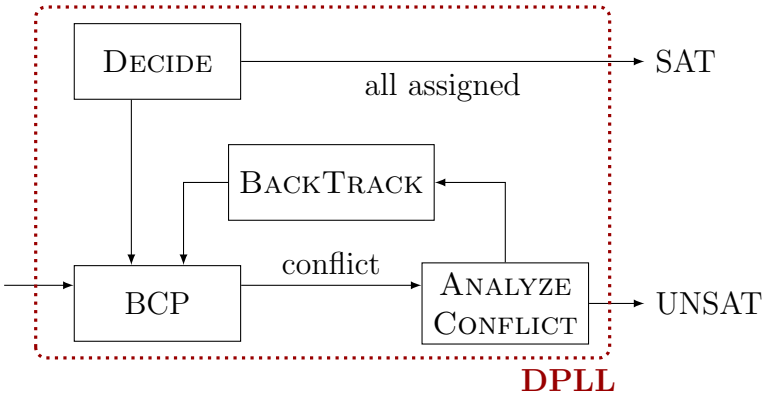


Figure 6.1: The DPLL algorithm

algorithm finds a cut in the implication graph for the conflict, constructs a *learnt clause* that is the negation of the assignments that led to the conflict, and then makes a jump back to the earliest point at which any of the variables in this clause were assigned to.

6.1.2 Incremental SAT solving

Incremental SAT solving is used when solving multiple similar SAT problems. Incremental solving strives to re-use as much information as possible from the previous SAT problems for solving future ones. Specifically, incremental SAT preserves the clauses learnt from conflicts.

Incremental SAT solving was first investigated in the 1990s [69]. Reusing the information is trivially easy if the formulae being solved are growing monotonically, i.e., new clauses are added to the previous formula. A solution to $C_1 \wedge C_2 \wedge C_3$ can trivially start from the place where the solution for $C_1 \wedge C_2$ left off. However, in many cases, we need to remove clauses as well as add new ones. The commonly used solver interface to deliver this feature is *solving under assumptions* [42, 99, 101].

When solving under assumptions, an incremental SAT solver is given the input of F_i, A_i at each SAT call i. Formula F_i is in CNF, and $A_i = l_1, l_2, ..., l_n$ is a conjunction of *assumptions*, i.e., assignments to variables. The SAT solver treats the assumptions as special literals that give the initial set of decisions. If it backtracks above the decision level of any assumption, the formula is unsatisfiable. Hence, clauses learnt are independent of the assumptions (because we did not backtrack above the decision level of the assumptions), and can be re-used when the assumptions no longer hold.

To enable removal of clauses we add a new, negated literal to each clause that we might want to remove for future SAT queries e.g., we add a new negated literal

to $c_1 = x_1 \wedge x_2$, which yields $c_1' = \neg a_1 \wedge x_1 \wedge x_2$. When $a_1 = TRUE$, $c_1 = c_1'$, and when $a_1 = FALSE$, c_1' is $TRUE$. To remove the clause we solve under the assumption that $a_1 = FALSE$. The clause c_1' is thus effectively eliminated from the SAT solvers consideration, because $\neg a_1 = TRUE$ and so the clause is satisfied.

There are also incremental SMT solvers. The SMT-lib specification format allows for incremental solving and a growing number of SMT solvers now support this feature.

6.1.3 Incremental SAT and formula preprocessing

Modern SAT solvers almost all have some kind of preprocessing step where the formula is simplified, which might involve eliminating redundant variables by resolution and removal of clauses by subsumption. Effective preprocessing has been shown to decrease the runtime of SAT solvers substantially [41]. In our implementation disabling the SAT solver preprocessing reduces the number of SyGuS benchmarks that can be solved within the timeout from 38 to 32.

However, a known issue with preprocessing is that, in it's basic form, it is incompatible with incremental solving as it may remove redundant variables that will be needed in subsequent solving steps. Sophisticated techniques exist to overcome this problem [100], but a straightforward solution, if the variables that will be relevant for subsequent SAT calls are known a priori, is to use "freezing", i.e., to mark these variables specifically to tell the preprocessor not to eliminate them.

In the case of CEGIS, we know that the variables and constants that must not be eliminated are ones that correspond to the counterexample inputs and the program, i.e., in the synthesis problem $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$we must not eliminate the constants corresponding to the set of inputs $x \in \mathcal{I}_{\mathcal{G}}$ and the variables corresponding to the program P.

This is relatively straightforward to implement. However, there is still a trade-off between using preprocessing and using incremental solving, which we will experimentally evaluate.

6.2 Using incremental solving in CEGIS

We implement incremental CEGIS, an improvement to the CEGIS algorithm by adding incremental solving. Recall the CEGIS algorithm, shown in Figure 6.2. The current synthesis block is based around the construction of a Boolean formula that is satisfied if there exists a candidate solution that works for a subset of possible inputs. In each iteration, the change in this formula is either that another input is added to the subset

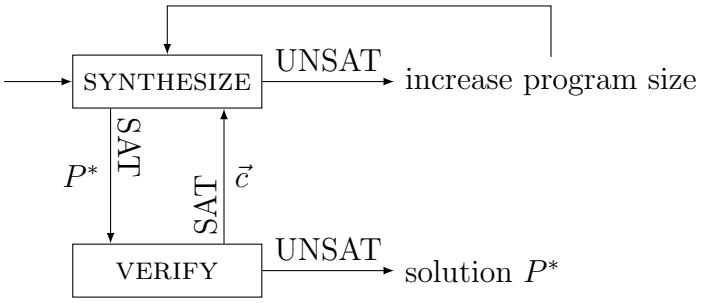


Figure 6.2: Recall CEGIS architecture

of possible inputs that the solution must work for, or the length of the program encoding is extended (e.g., from two instructions to three instructions). In the first case, the first-order solver will thus in many cases repeat the same steps at the beginning of each synthesis step. This presents a good use case for incremental satisfiability solving because multiple very similar queries being made both in the synthesis module. Each time CEGIS returns from the verification block with a counterexample, it adds new clauses to the synthesis problem, but the old clauses are still required to be satisfied. Consider the example previously given in Chapter 4:

Example 5. *Recall the specification, which requires the solver to synthesise a function* $f(x)$ *where* $f(x) < 0$ *if* $x < 334455$ *and* $f(x) = 0$*, otherwise:*

```
(set-logic BV)
(synth-fun f (x (BitVec 32)) (BitVec 32))
(declare-var x (BitVec 32))
(constraint (ite (bvult x 334455 ) (bvult (f x) 0) 0))
(check-synth)
```

Synthesis: *The synthesis block solves the formula* $\exists P. \forall \vec{x} \in \mathcal{I}_{\mathcal{G}}. \sigma(\vec{x}, P)$*. Suppose that we are mid-way through the run of CEGIS presented in Section 4.2. The first counterexample we obtain is* $x = 0$*. The synthesis block then solves the formula:*

$$(x_0 < 334455 \wedge f(x_0) < 0 \vee x_0 \geq 334455 \wedge f(x_0) = 0) \wedge (x_0 = 0)$$

That is, in SyGuS-IF:

```
(constraint (and (ite (bvult x0 334455 ) (bvult (f x0) 0) 0)) (= x0
    0))
```

Suppose the second counterexample obtained is $x = 1$, the synthesis block must then solve the formula:

$$(x_0 < 334455 \wedge f(x_0) < 0 \vee x_0 \geq 334455 \wedge f(x_0) = 0) \wedge (x_0 = 0) \wedge$$
$$(x_1 < 334455 \wedge f(x_1) < 0 \vee x_1 \geq 334455 \wedge f(x_1) = 0) \wedge (x_1 = 1)$$

That is, in SyGuS-IF:

```
(constraint (and
(and (ite (bvult x0 334455 ) (bvult (f x0) 0) 0)) (= x0 0)
(and (ite (bvult x1 334455 ) (bvult (f x1) 0) 0)) (= x1 1)))
```

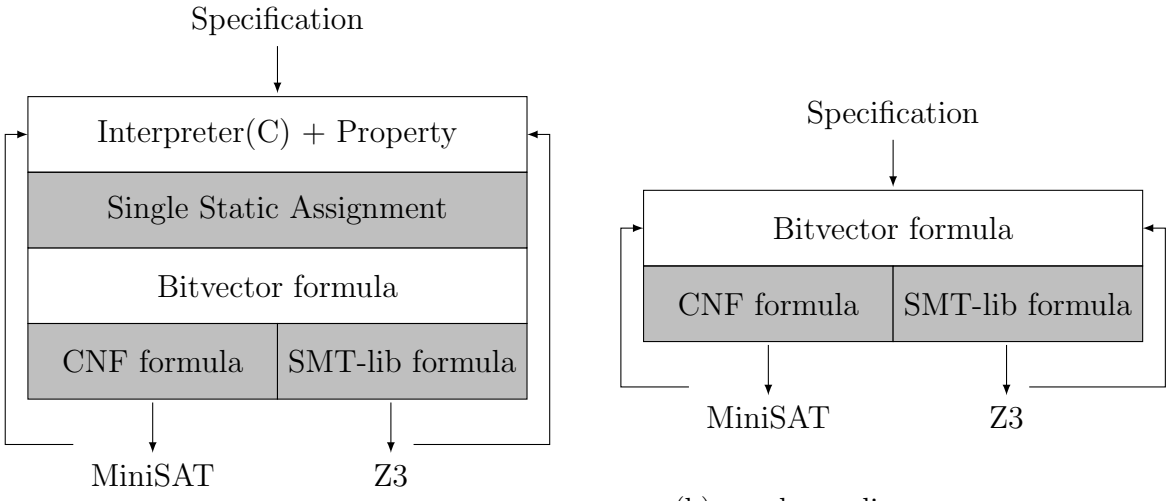
And suppose that the third counterexample obtained is $x = 2$, the synthesis block must then solve the following:

$$(x_0 < 334455 \wedge f(x_0) < 0 \vee x_0 \geq 334455 \wedge f(x_0) = 0) \wedge (x_0 = 0) \wedge$$
$$(x_1 < 334455 \wedge f(x_1) < 0 \vee x_1 \geq 334455 \wedge f(x_1) = 0) \wedge (x_1 = 1) \wedge$$
$$(x_1 < 334455 \wedge f(x_2) < 0 \vee x_2 \geq 334455 \wedge f(x_2) = 0) \wedge (x_2 = 2)$$

```
(constraint (and
(and (ite (bvult x0 334455 ) (bvult (f x0) 0) 0)) (= x0 0)
(and (ite (bvult x1 334455 ) (bvult (f x1) 0) 0)) (= x1 1))
(and (ite (bvult x2 334455 ) (bvult (f x2) 0) 0)) (= x2 2)))
```

At each synthesis iteration, if a new input is added, a new conjunct is added to the formula from the previous synthesis step. This makes CEGIS a potential candidate for use of incremental SAT solving, whereby the SAT solver remembers information learnt from the previous problem. Note that, when the program size is increased the Boolean formula will change substantially as another instruction is added to the program encoding. At this point, we must reset the SAT solver because the SAT problems between those two iterations will be too different.

Verification: The verification phase is not a clear candidate use case for incremental SAT. Each time the verification block is called, it is verifying a different candidate program, and so, whilst it would be possible to work out how much of the candidate solution differs from the previous one, that would likely be computationally expensive for potentially a small benefit (there is no reason for consecutive candidate solutions to be similar). The time spent in the verification phase in CEGIS is typically less than 20% of the total solving time.



(a) Interpreter-based encoding from [35]

(b) novel encoding

Figure 6.3: Comparison between the two encodings presented in Section 5

6.2.1 Impact of the novel encoding on incremental solving

Recall that in Chapter 5 we presented a novel method for encoding the search space for candidate programs in CEGIS. We also described the previous encoding, based on instruction sets, as used in [35]. Figure 6.3a and Figure 6.3b provide some intuition as to why the novel encoding will benefit more from the use of incremental solving than the interpreter-based encoding, namely that the novel encoding allows a loop almost directly around the SAT solver as it works entirely in bitvector formulae. The previous instruction set encoding requires translation from C to bitvectors which must be included in the incremental solving loop.

6.3 Illustrative Example

6.3.1 Minimal Example

Example 6. *In order to illustrate the use of incremental SAT in CEGIS, let us consider the following simple example:*

```
(set-logic BV)
(synth-fun f ((in (BitVec 8) )) (BitVec 8))
(declare-var in (BitVec 8) )
(constraint (= ( f in ) (bvor in #x01)))
(check-synth)
```

This specification requires synthesis of a program that has a single input and returns the result of performing a bit-wise or with that input and the bitvector 00000001*. Recall that CEGIS begins by attempting to synthesising a program of length* 1 *that satisfies a single input. The synthesis encoding for a program of length* 1*, which allows one constant,*C_0*, and has one input, in, is*

$$I_0 = \begin{cases} in & : & sel_0 \\ C_0 & : & otherwise \end{cases}$$

We use I_j *to denote the output of instruction* j*. We use* f *to denote the output of function* f*. We are encoding a program of length* 1 *and so the output of* f *is the output of* I_0*. The full formula for this first synthesis step is:*

$$\begin{array}{rcl} I_0 & = & \begin{cases} in & : & sel_0 \\ C_0 & : & otherwise \end{cases} \\ \\ \wedge \quad f & = & I_0 \\ \wedge \quad f & = & (in \,|\, 00000001) \end{array}$$

Using non-incremental CEGIS with the novel encoding and non-incremental solving, this is formula is represented by MiniSAT using 25 variables, 46 clauses and is solved with one decision and one Boolean constraint propagation step. The first solution returned by CEGIS is $f = in$*. The first synthesis step only requires finding a program that works for a single input A, and this solution works for the input* 1*. The counterexample returned by the verification phase is that this solution fails for input value* 0*. We must add a constraint for each counterexample and duplicate the encoding, and so we introduce the notation* I_{j-cexk}*, which denotes the output of instruction* j *when the input to the program is counterexample* k*, and* f_{cexk} *which denotes the output of applying the program* f *to the* k^{th} *counterexample. Thus the next synthesis formula is:*

$$\begin{array}{rcl} I_{0-cex0} & = & \begin{cases} 0 & : & sel_0 \\ C_0 & : & otherwise \end{cases} \\ \\ \wedge \quad f_{cex0} & = & I_{0-cex0} \\ \wedge \quad f_{cex0} & = & (0 \,|\, 00000001) \end{array}$$

In non-incremental CEGIS this is solved with 17 variables, 24 clauses, one decision and 17 propagations, giving the next candidate program to be $f = 1$*. The following counterexample is* 2*. At this point, we need to extend the length of the program because*

iter.	CDCL stats						candidate P	#cex
	vars	clauses	starts	conflicts	decs	props		
1	25	46	1	0	17	26	$f = in$	0
2	17	24	1	0	2	18	$f = 1$	2
3	25	24	–	–	–	–	no solution	–
increase program size								
4	95	299	–	–	–	–	no solution	–
increase program size								
5	1352	5268	2	131	479	28506	$f = 1 \oplus in$	1
6	2004	7902	2	159	449	51623	$f = 1 \,\vert\, in$	none

Table 6.1: CEGIS solving stats, showing solver starts, conflicts, decisions and propagations, using non-incremental solving with preprocessor enabled

there are no more solutions for a program of length 1*, and so the next formula for synthesis is:*

$$
\begin{aligned}
I_{0-cex0} &= \begin{cases} 0 & : \; sel_0 \\ C_0 & : \; otherwise \end{cases} \\
\wedge \; I_{0-cex1} &= \begin{cases} 2 & : \; sel_0 \\ C_0 & : \; otherwise \end{cases} \\
\wedge \; f_{cex0} &= I_{0-cex0} \\
\wedge \; f_{cex0} &= (0 \,\vert\, 00000001) \\
\wedge \; f_{cex1} &= I_{0-cex1} \\
\wedge \; f_{cex1} &= (2 \,\vert\, 00000001)
\end{aligned}
$$

This formula is 95 variables, 299 clauses and has no solutions. The program size must now be increased to three instructions. The resulting formula has 1352 variables and 5268 clauses, and, during solver, the solver makes 178 decisions, 74 conflicts are encountered and 1972 propagations performed. After this formula, we obtain the candidate solution $f = 1^{in}$*), and then after a further iteration we finally obtain the correct solution. Table 6.1 presents the size of the synthesis formula at each iteration, the number of starts, decisions, conflicts and propagations made by the SAT solver, and the candidate solutions for normal CEGIS without incremental SAT and without using preprocessing. Table 6.2 presents the same information for the incremental implementation with no preprocessing.*

The differences to note between the two tables are that the incremental setup constructs marginally larger formula, but the number of conflicts encountered and

iter.	CDCL stats						candidate P	#cex
	vars	clauses	starts	conflicts	decs	props		
1	25	46	1	0	17	26	$f = in$	0
2	33	68	1	0	10	33	$f = 1$	2
3	41	68	–	–	–	–	no solution	–
increase program size								
4	141	491	–	–	–	–	no solution	–
increase program size								
5	2012	7940	2	121	517	36116	$f = 1 \oplus in$	1
6	2664	10442	1	17	50	7766	$f = 1 \,\vert\, in$	none

Table 6.2: CEGIS solving stats, showing solver starts, conflicts, decisions and propagations using incremental solving with preprocessor enabled

decisions and propagations performed by the SAT solver is reduced substantially in iteration 6, because the incremental solver is able to re-use the work done in iteration 5.

6.3.2 A longer example

One thing the example above shows is that incremental SAT is not necessarily going to be very useful in cases where the final solution is found using only a few iterations at each program size and where the SAT formulae are generally small. When the final solution requires solving bigger formulae, incremental SAT should become more useful. Consider the following example, taken from the SyGuS competition invariant track, and where a valid final solution is $x - y \geq x$. Note this solution depends on the overflow of bitvectors and is not equivalent to $y \leq 0$. This benchmark is solved by incremental CEGIS with preprocessing enabled in 468 seconds and by non-incremental CEGIS with preprocessing enabled in 1324 seconds. With preprocessing disabled, incremental CEGIS solves the benchmark in 2523 s and non-incremental CEGIS solves the benchmark in 13247 s.

In Table 6.4 and Table 6.3, we give the formula size for the iterations that occur after the program size is increased to four instructions, with preprocessing enabled, and the number of conflicts encountered and starts, decisions and propagations performed by the SAT solver in each iteration for both implementations. The sequences of candidate programs produced by the two different implementations are not the same, and so the formula size differs for each iteration slightly.

The graph in Figure 6.4 shows the cumulative number of conflicts encountered by

	non-incremental CDCL stats					
iterations	vars	clauses	starts	conflicts	decisions	propagations
19	534644	1885211	77	21504	112546	90363818
20	570272	2010981	180	56996	274477	221694821
21	605900	2136559	252	80154	428136	452224767
22	641528	2262330	255	99397	466039	596163076
23	677156	2388101	444	162654	743020	1006937341
24	712784	2513872	255	93504	439075	697146125
25	748412	2639450	893	377652	1498854	3224483151

Table 6.3: CEGIS solving stats using non-incremental solving for benchmark inv_gen_cegar2.sl from the SyGuS competition [10], showing iterations after the program size is increased to 4 instructions. The solving stats are given per iteration.

the solver for incremental and non-incremental CEGIS, with preprocessing disabled. Figure 6.6 shows the cumulative time spent in the solving phase of CEGIS, for the same benchmark. The step jumps in cumulative time occur when the program size is increase and so the solver is reset and the solving must start from scratch. We can see that with preprocessing disabled, incremental solving is clearly beneficial. The number of conflicts encountered and time spent in the solving phase increase exponentially as iterations increase for the non-incremental solving. The non-incremental implementation takes more than 5 times longer to solve the benchmark than the incremental implementation despite doing three fewer iterations. The number of iterations each implementation needs is dependent on the sequence of counterexamples obtained, which is dependent on SAT solver heuristics. However, with preprocessing enabled, as shown in Figure 6.5 and Figure 6.7 the case is less clear-cut; the non-incremental solving benefits far more from the preprocessing than the incremental solving, and with the preprocessing enabled the non-incremental implementation takes only 2.5 times as long, despite performing one more iteration than the incremental implementation. The number of conflicts resolved per iteration are roughly comparable with preprocessing enabled.

6.4 Results on full Benchmark Sets

The results on the single benchmark we presented in the preceding section suggest incremental solving in CEGIS could be beneficial. However, this is only one exemplar. In this section we evaluate the benefit of incremental solving in CEGIS using the novel

	non-incremental CDCL stats					
iterations	vars	clauses	starts	conflicts	decisions	propagations
16	463979	1636157	62	15160	63257	75116989
17	499607	998327	1	4	965	179270
18	535235	1061752	109	29957	141043	223481115
19	570863	1128189	371	134271	595847	1035244439
20	606491	1195395	2	103	1738	2422165
21	642119	1261554	44	9944	40834	125652760
22	677747	1327823	62	15776	60929	200783067
23	713375	1394276	125	35274	119957	440154269
24	749003	1461693	1	74	1741	940978

Table 6.4: CEGIS solving stats using incremental solving for benchmark inv_gen_cegar2.sl from the SyGuS competition [10], showing iterations after the program size is increased to 4 instructions. The solving stats are given per iteration.

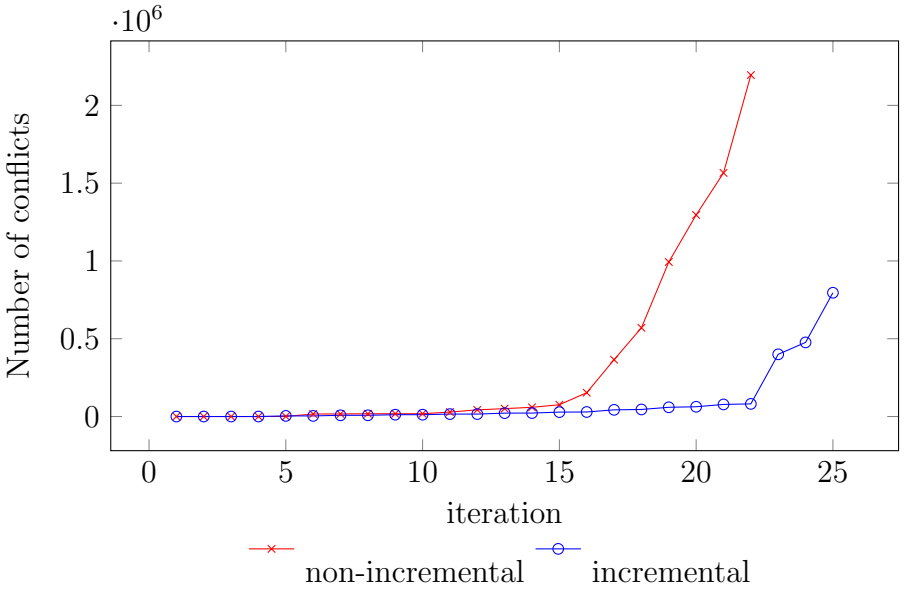


Figure 6.4: Cumulative conflicts resolved, shown for incremental and non-incremental solving with solver preprocessing disabled

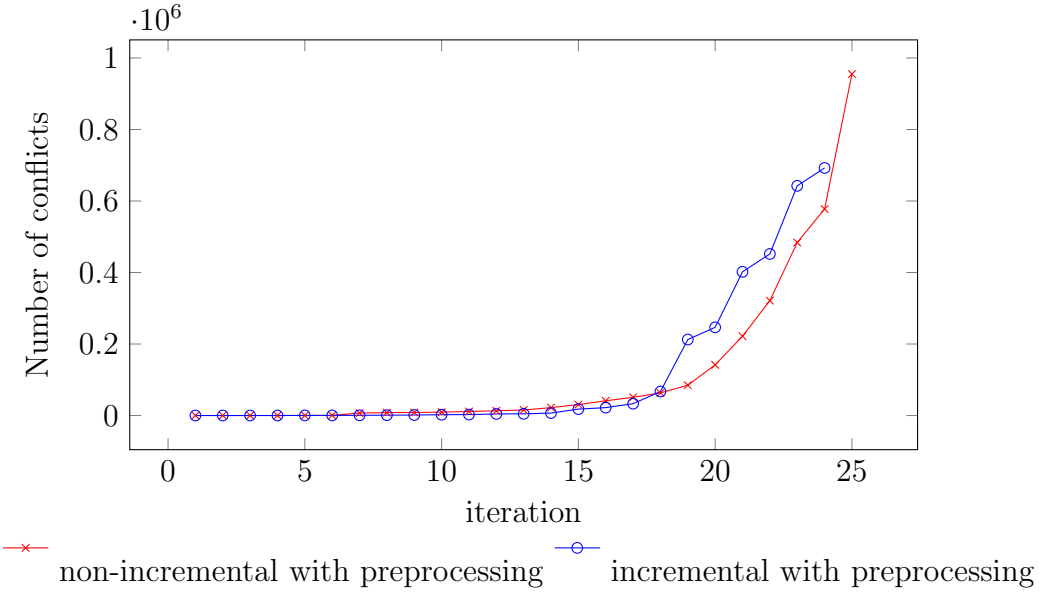


Figure 6.5: Cumulative conflicts resolved, shown for incremental and non-incremental solving with solver preprocessing enabled

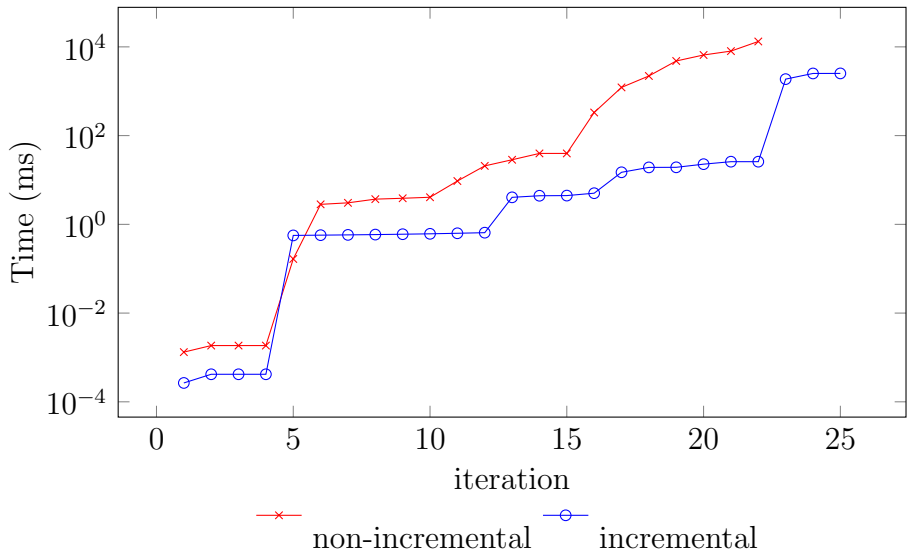


Figure 6.6: Cumulative time spent in synthesis phase, shown for incremental and non-incremental solving with solver preprocessing disabled

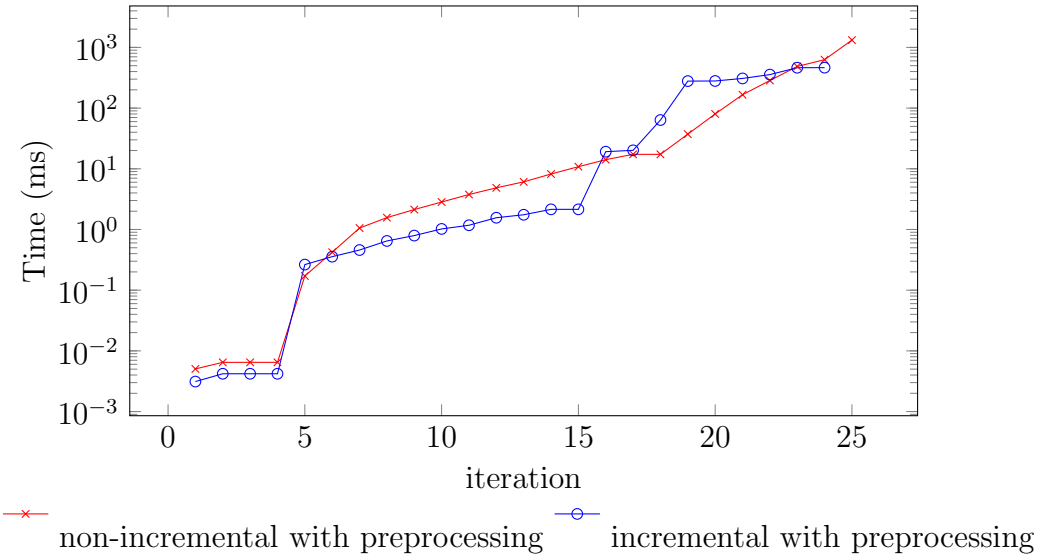


Figure 6.7: Cumulative time spent in synthesis phase, shown for incremental and non-incremental solving with solver preprocessing enabled

category	total	
invariant generation	47	from SyGuS competition and program analysis
comparisons	7	e.g., find the maximum
hacker's	6	from Hacker's Delight [76]
other SyGuS	23	
controller synthesis	29	6 time discretisations per benchmark

Table 6.5: Categories of the benchmarks

encoding presented in Chapter 5 and the interpreter-based encoding used in [10]. We evaluate on a set of benchmarks taken from program analysis, the SyGuS competition, and also the controller synthesis benchmarks. The benchmarks are split across categories as shown in Table 6.5.

6.4.1 Synthesising Controllers

First we present our evaluation of incremental CEGIS in comparison to standard CEGIS on the controller synthesis benchmarks as detailed in Chapter 3. Recall that these benchmarks require the synthesis of a matrix of fixed-point values such that a safety specification is met for a Linear Time Invariant system. These benchmarks do not require synthesis of expressions, and so are a simplified case of program synthesis. They thus provide an interesting case to evaluate incremental CEGIS on, because

they remove the risk of the incremental synthesis procedure choosing the wrong initial program structure and being forced to extensively explore that structure before finding a counterexample strong enough to refute the template completely.

There is an element of luck involved in the speed of solving these benchmarks, owing to the completeness threshold. The completeness threshold is not specific to a benchmark, but to the combination of benchmark and controller. In the original work published at CAV, we initialise the algorithm with a bound k, and the algorithm synthesises a controller that is safe for k steps, and then we check whether the controller synthesised has a completeness threshold less than k, i.e., is always safe. If it does not, we increase k and repeat the process. A lucky synthesis algorithm will guess a controller that has a completeness threshold lower than k, an unlucky one may need to repeat the process many times with ever increasing k values.

It is not possible to integrate the requirement that the controller be safe up until the completeness threshold into the CEGIS synthesis block itself because calculating the completeness threshold involves computing Eigenvalues, which we have found SAT solvers to be very poor at (perhaps unsurprisingly, given the non-linearity involved).

To reduce the element of luck when comparing incremental vs standard CEGIS, we fix the bound k and synthesise controllers that remain safe for up to ten time steps. We also omit the precision check, as described in the previous sections, and fix the precision used to represent the plant and controller. The precision check is the same for both incremental and standard CEGIS, and so is a constant overhead. Both incremental and non-incremental CEGIS compared in this chapter use the new encoding described in Chapter 5. We run each benchmark with several different time discretisations. This gives us a total of 87 benchmarks that can be solved by both incremental and non-incremental CEGIS.

Incremental CEGIS does not perform well on these benchmarks, performing an average of 32 s slower than non-incremental CEGIS with preprocessing enabled. The reason for this is likely that these benchmarks typically only require 4 or 5 iterations, and so the benefit of incremental solving is outweighed by the limited preprocessing the solver is able to perform. If we look at the size of the formula in the final synthesis step in cases where the number of iterations each implementation performs is identical, incremental CEGIS solves a formula 20% larger than non-incremental CEGIS.

Each iteration also takes relatively longer than the SyGuS benchmark iterations due to the complexity of the controller synthesis specification. This means that a solver taking more iterations due to luck of the sequence of counterexamples obtained can increase the synthesis time significantly. In 17 out of 87 benchmarks, non-incremental

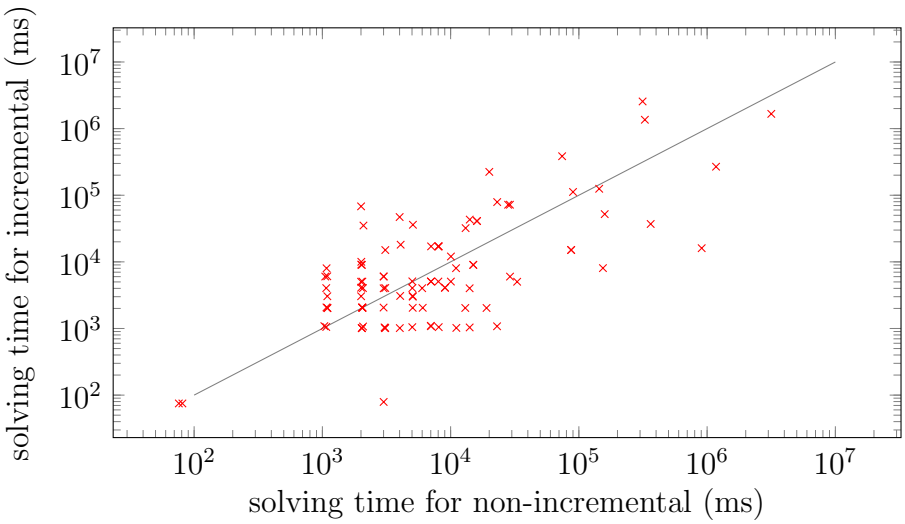


Figure 6.8: Solving time of incremental vs non-incremental with preprocessing disabled on the novel encoding. Points that fall underneath the grey line indicate benchmarks that were solved faster by incremental solving than non-incremental. The average time per benchmark for incremental solving is 436.2 s, soling 36 benchmarks, and for non-incremental solving 141.2 s, solving 32 benchmarks.

CEGIS takes more iterations to find a solution than incremental CEGIS, and in 13 out of 87 benchmarks incremental CEGIS takes more iterations to find a solution than non-incremental CEGIS. This is down to the heuristic choices inside the SAT solver. If we omit the results where either CEGIS requires a different number of iterations, incremental CEGIS performs an average of 36 s slower than non-incremental CEGIS.

6.4.2 SyGuS Competition benchmarks

Interpreter-based encoding: We give results for a subset of the SyGuS competition benchmarks using the interpreter-based encoding from [10], and as detailed in Chapter 5. We run the implementation on 29 benchmarks from the SyGuS competition, and the implementations using non-incremental and incremental solving both solve the same 14 benchmarks. We give these results in Figure 6.9 and Table 6.6. All benchmarks are solved faster by incremental than non-incremental and the average time taken is 265.6 s for non-incremental solving and 81.7 s for incremental solving. This improvement of $3.2x$ is better than the anticipated $2x$. In Figure 6.10 we give a comparison between the size of the formula produced by the final synthesis phase for incremental and non-incremental solving and find that the formula sizes are comparable, and so the difference in solving time is likely due to the incremental solving.

Benchmark	comparisons		hackers		inv		other		total	
	#	s	#	s	#	s	#	s	#	s
interpreter-based encoding:										
CEGIS	2	256.5	3	50.7	-	-	9	339.3	14	265.6
CEGIS-inc	2	189.5	3	23.7	-	-	9	77.0	14	81.7
novel encoding:										
CEGIS	1	<0.1	3	23.7	-	-	13	6.5	17	9.2
CEGIS-inc	2	<0.1	3	168.0	-	-	13	3.3	17	32.2
total benchmarks	7	-	6	-	0	-	16	-	29	-

Table 6.6: Results summary for interpreter-based encoding

	CEGIS		CEGIS-np		CEGIS-i		CEGIS-i-np		multicore		multicore-i	
	#	s	#	s	#	s	#	s	#	s	#	s
comp	1	<0.1	1	<0.1	1	<0.1	1	<0.1	1	<0.1	1	<0.1
hackers	3	23.7	2	2.5	3	168.1	3	452.7	3	23.7	3	23.4
inv	19	430.3	15	291.0	18	223.8	18	790.7	25	423.0	25	389.0
other	15	70.0	14	10.6	15	18.7	14	8.0	15	61.4	15	18.6
total slvd	38	244.6	32	141.2	37	130.1	36	436.2	44	263.0	44	229.0

Table 6.7: Experimental results for the novel encoding – for every set of benchmarks, we give the number of benchmarks solved by each configuration within the timeout and the average time taken per solved benchmark

Novel encoding: We also run the full benchmark set using the novel encoding proposed in Chapter 5. In Chapter 4 we presented a table of results on the SyGuS benchmarks that included a hypothetical multi-core solver. Here we show the results on the same set of benchmarks for CEGIS, CEGIS with the preprocessor disabled (CEGIS-np), CEGIS with incremental solving with preprocessing(CEGIS-i), CEGIS with incremental solving with the preprocessor disabled (CEGIS-i-np), and the multi-core solver if the incremental solving with preprocessing was included as an extra core (multicore-i). For comparison we show the multicore engine (multicore) from Chapter 4, which comprised of CEGIS-FM, CEGIS and CEGIS-SMT, and the plain implementation of CEGIS:

The average time taken to solve each benchmark is improved by using incremental-solving, although one fewer benchmark is solved. A graph showing the comparison between incremental and non-incremental solving with preprocessing enabled is shown

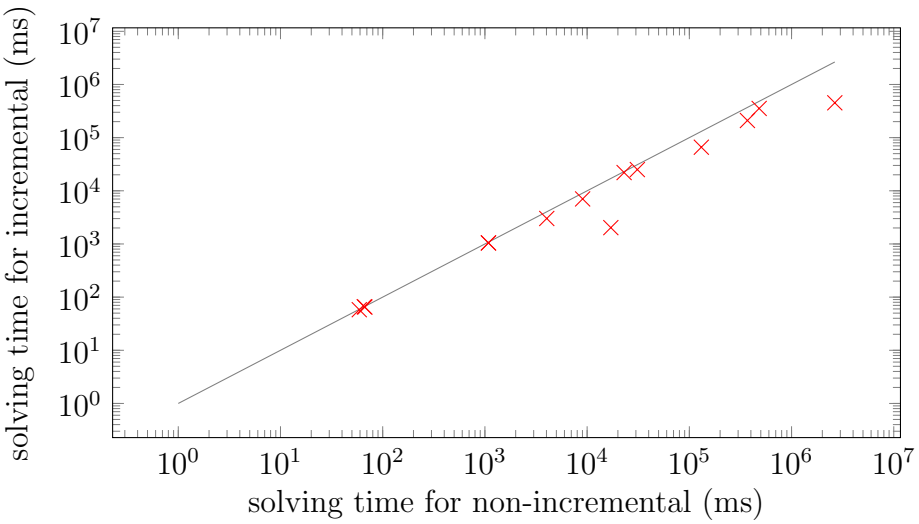


Figure 6.9: Solving time of incremental vs non-incremental with preprocessing enabled on the interpreter-based encoding. Points that fall underneath the grey line indicate benchmarks that were solved faster by incremental solving than non-incremental. The average solving time is 265.6 s for non-incremental solving and 81.7 s for incremental solving.

in Figure 6.11. The results with the preprocessing disabled are shown in Figure 6.12. Here, incremental solving is more clearly beneficial, and incremental solving solves 36 benchmarks, whilst non-incremental solving only solves 32. The average time per benchmark is higher for incremental solving because the extra benchmarks solved are challenging. There is, however, a more clear trend illustrated on the graph where incremental solving in most cases is faster than non-incremental solving.

Both with and without preprocessing, there are several cases where non-incremental solving beats incremental. This is because with more complex benchmarks the choice of counterexamples can be critical in leading the solver to the correct solution; a particular counterexample early on in the CEGIS algorithm can be decisive in leading the algorithm to explore a specific direction of potential programs, which may not contain the solution. CEGIS may then need many more iterations before it hits on the correct solution. For this reason, it is hard to draw the conclusion that incremental solving is substantially better than non-incremental solving in CEGIS, and we would be able to have more confidence in our conclusions if we had a larger benchmark set, on which these anomalies would be less significant.

Note that the average times are slower than for the interpreter-based encoding because the benchmark set includes more challenging benchmarks, of which 37 are solved by incremental solving and 38 with non-incremental solving.

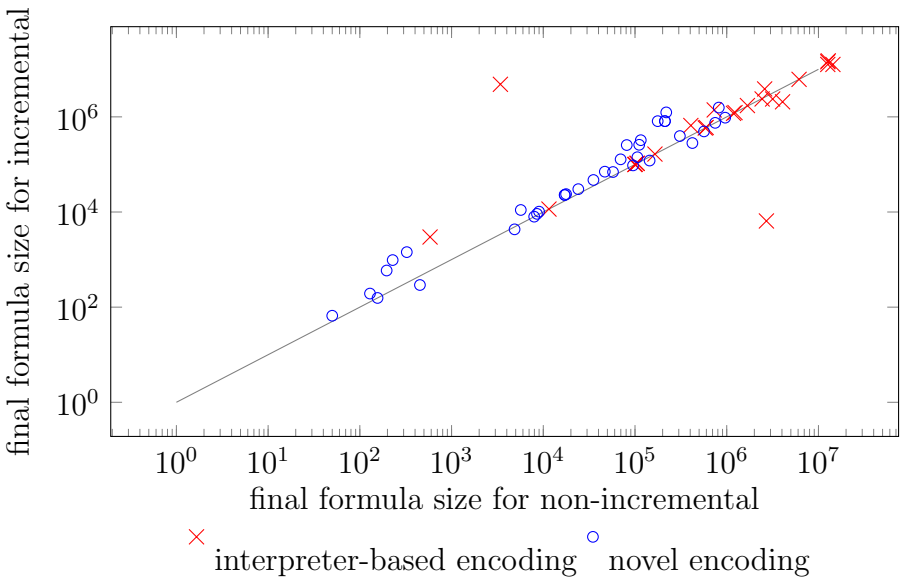


Figure 6.10: Number of variables in final synthesis formula using the interpreter-based encoding and novel encoding. Points that fall above the grey line indicate benchmarks where the formula was smaller for incremental solving than non-incremental.

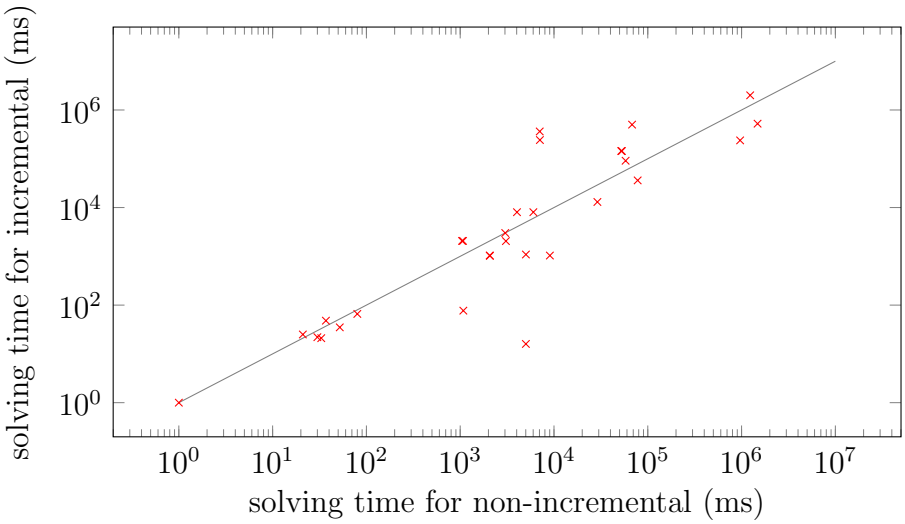


Figure 6.11: Solving time of incremental vs non-incremental with preprocessing enabled on the novel encoding. Points that fall underneath the grey line indicate benchmarks that were solved faster by incremental solving than non-incremental. The average time per benchmark for incremental solving is 130 s, solving 37 benchmars, and for non-incremental solving 244.6 s, solving 38 benchmarks.

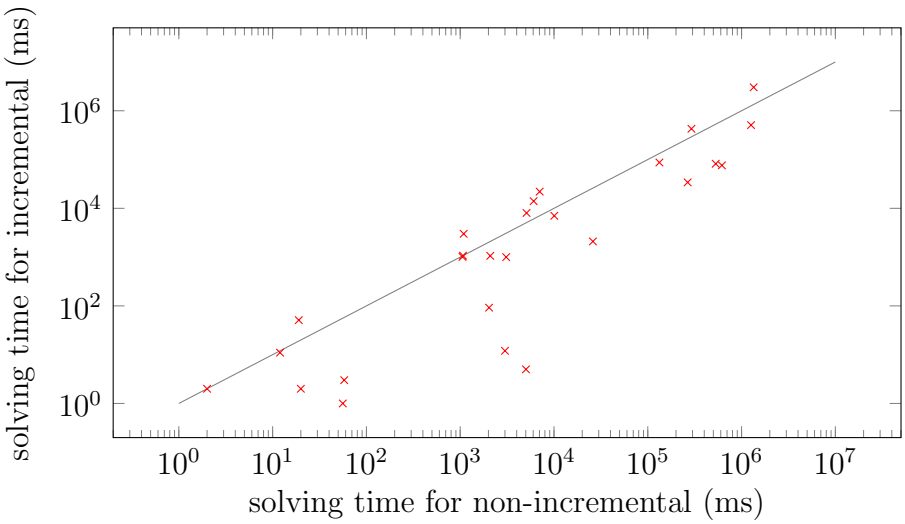


Figure 6.12: Solving time of incremental vs non-incremental with preprocessing disabled on the novel encoding. Points that fall underneath the grey line indicate benchmarks that were solved faster by incremental solving than non-incremental. The average time per benchmark for incremental solving is 436.2 s, soling 36 benchmarks, and for non-incremental solving 141.2 s, solving 32 benchmarks.

6.5 Conclusions

Whilst incremental solving in CEGIS had seemed initially to be a promising development, it turned out to be no better than simply using the SAT solver preprocessor. We enabled the preprocessor with the incremental solving by using a crude technique of freezing all the variables that correspond to counterexample inputs obtained by CEGIS. This proves to be too crude for preprocessing to be effective. The results show that without preprocessing, incremental solving increases the number of SyGuS benchmarks CEGIS is able to solve by four.

We combine the implementation of CEGIS using incremental solving with the preprocessor enabled into the multi-core engine mentioned in Chapter 4, and it gives a slight improvement of 6% in the average time per benchmark solved.

Our research hypothesis looks to establish that synthesis of programs containing arbitrary constants is computationally feasible without user-given syntactic guidance. CEGIS reaches the time-out threshold on many of our program synthesis benchmarks and, approaching this work, we speculated that incremental first-order solving may provide a speed-up large enough to prevent some of these time-out results. However, the speed-up provided by incremental first-order solving was not significant enough to allow us to solve any additional benchmarks, and does not provide any contributions

towards improving computational feasibility of synthesis of programs with arbitrary constants.

There exists promising work developing satisfiability solvers that use preprocessing with incremental solving, and when these solvers are competitive, incremental solving in CEGIS may be able to provide a more significant speed-up. Given the current state-of-the-art in SAT solver pre-processing, we hypothesize that incremental CEGIS is only able to outperform a standard CEGIS implementation on benchmarks where the SAT solver pre-processing does not already provide a significant speed-up.

Chapter 7

Conclusions

Recall the research hypothesis explored by this dissertation, which is that synthesis of programs containing arbitrary constants is computationally feasible without provision of syntactic templates. To that end, we have presented the following algorithmic contributions:

- In Chapter 4 we presented CEGIS($\mathcal{T}$), a novel algorithm for program synthesis based on combining SAT-based counterexample guided inductive synthesis with a first-order solver. CEGIS($\mathcal{T}$) is an extension to CEGIS in the same way that DPLL($\mathcal{T}$) is an extension of DPLL. CEGIS($\mathcal{T}$) abstracts the synthesis problem to a higher level, then uses a first-order theory solver to provide counterexamples to abstracted candidate solutions to the synthesis phase. We evaluated CEGIS($\mathcal{T}$) on a set of benchmarks taken from the SyGuS competition with the syntactic template removed and find that CEGIS($\mathcal{T}$) is able to solve benchmarks that elude our standard implementation of CEGIS. A variant of CEGIS($\mathcal{T}$) has been implemented in CVC4, and is able to solve benchmarks that elude the standard algorithms of CVC4.

- We introduced a novel encoding of the synthesis problem in Chapter 5, which we showed to reduce the size of the SAT formula generated in the synthesis phase of CEGIS, and to increase the total solving time by two orders of magnitude on SyGuS competition benchmarks.

- In Chapter 6 we presented the use of incremental first-order solvers in CEGIS. We evaluate this contribution and find that, whilst CEGIS implemented with incremental solving provides a speed-up of 50% over CEGIS with non-incremental solving if the solver preprocessing is disabled, once the preprocessing is enabled the speed-up provided is significantly smaller and, for many benchmarks, is

Benchmark	comparisons		hackers		inv		other		total	
	#	s	#	s	#	s	#	s	#	s
interpreter-based encoding:										
CEGIS	2	256.5	3	50.7	-	-	9	339.3	14	265.6
CEGIS-inc	2	189.5	3	23.7	-	-	9	77.0	14	81.7
novel encoding:										
CEGIS	1	<0.1	3	23.7	-	-	13	6.5	17	9.2
CEGIS-inc	2	<0.1	3	168.0	-	-	13	3.3	17	32.2
total benchmarks	7	-	6	-	0	-	16	-	29	-

Table 7.1: Results summary

outweighed by the disadvantage of being unable to use pre-processing effectively with incremental solving. A larger benchmark set would be needed to establish whether this speed-up is consistent. As first-order solvers develop that allow effective preprocessing to be used with incremental solving, the use of incremental first-order solvers in CEGIS may become more beneficial.

A full summary of our results is shown in Table 7.1 and Table 7.2 .

7.0.1 High-level summary

Our results show that the synthesis of programs containing arbitrary constants is computationally feasible without user-provided syntactic templates. The key algorithmic enabler to this is CEGIS($\mathcal{T}$), which prevents the enumerative behaviour often encountered in CEGIS when attempting to synthesise programs containing constants. CEGIS($\mathcal{T}$) solves most benchmarks that require non-trivial constants faster than CEGIS.

However, it benefits from being run in a multi-core solver since there are cases where CEGIS($\mathcal{T}$) is slower than CEGIS. We hypothesise in some cases this is because it changes the sequence of counterexamples obtained, and thus may need to perform more iterations. It is known that the sequence of counterexamples obtained affects the speed of CEGIS [73]. In other cases, we hypothesise that CEGIS($\mathcal{T}$) may perform slower when it generates SAT formulae that are harder to solve than the ones generated by CEGIS. Classifying SAT formula that are more easily solved remains an open problem [130].

The results for the use of incremental first-order solvers in CEGIS show that incremental solving is beneficial when used as part of the hypothetical multi-core

Benchmark	comparisons		hackers		inv		other		total	
	#	s	#	s	#	s	#	s	#	s
novel encoding:										
CEGIS	1	<0.1	3	23.1	19	430.2	15	70.0	38	244.6
CEGIS-np	1	<0.1	2	2.5	15	291.0	14	10.6	32	141.2
CEGIS-inc	1	<0.1	3	168.1	18	223.8	15	18.7	37	130.1
CEGIS-i-np	1	<0.1	3	452.7	18	790.7	14	8.0	36	436.2
CEGIS($\mathcal{T}$)-FM	1	0.6	3	31.0	15	84.3	13	85.1	32	77.0
CEGIS($\mathcal{T}$)-SMT	1	<0.1	3	25.4	25	441.5	15	61.4	44	273.5
CEGIS($\mathcal{T}$)-multicore	1	<0.1	3	23.4	25	389.0	15	18.6	44	229.0
CVC4-1.5	7	<0.1	6	<0.1	6	6.1	14	121.1	33	52.5
CVC4-1.7	7	0.6	6	<0.1	29	36.5	17	93.1	59	44.8
CVC4-CEGIS($\mathcal{T}$)	1	2551.1	0	-	26	295.6	15	276.4	42	342.4
CVC4-multicore	7	0.6	6	<0.1	31	4.1	18	95.1	62	29.7
total benchmarks	7	-	6	-	47	-	23	-	83	-

Table 7.2: Results summary for encoding comparison. CEGIS-multicore includes CEGIS($\mathcal{T}$)-FM, CEGIS($\mathcal{T}$)-SMT, CEGIS and CEGIS-inc. CVC4-multicore includes CVC4 version 1.7 and CVC4-CEGIS($\mathcal{T}$)

solver, but is in general limited by the necessary restrictions on the SAT solver pre-processing. We hypothesise that the benchmarks where using incremental first-order solving is an advantage are the benchmarks where the SAT solver pre-processing has less benefit. When SAT solver pre-processing advances and becomes more compatable with incremental solving, this will be worth revisiting.

The novel encoding presented in Chapter 5 provides a significant speed-up over the interpreter-based encoding previously used, in all cases where the benchmarks require synthesising a program longer than a single instruction.

7.1 Future Work

CEGIS($\mathcal{T}$) naturally opens up opportunities for further work: one such area is exploration of further methods of generalisation, as in this dissertation we only look at one method of generalisation. Further, we only evalulate CEGIS($\mathcal{T}$) with two possible first-order solvers, and there are many more first-order solving techniques which may be appropriate for integration in the CEGIS($\mathcal{T}$) algorithm.

In this dissertation we evaluated a hypothetical multi-core solver, where several configurations of CEGIS and CEGIS($\mathcal{T}$) are run in parallel and the result is taken

from the configuration which returns fastest. In future work it would be interesting to explore whether CEGIS and CEGIS($\mathcal{T}$) would benefit from sharing intermediate results between the configurations as they run in parallel, specifically sharing intermediate solutions and counterexamples.

Appendix A

Example Benchmarks and Solutions

A.1 Example benchmarks from the SyGuS competition

The following benchmark is an example benchmark from the SyGuS Competition, in SyGuS-IF format.

```
(set-logic BV)

(synth-fun inv ((i (BitVec 32))(l (BitVec 32))) Bool
)

(declare-var i (BitVec 32))
(declare-var l (BitVec 32))
(declare-var i1 (BitVec 32))
(declare-var l1 (BitVec 32))
(declare-var l2 (BitVec 32))

(constraint
(=> (= l #x00000000) (inv i l))
)

(constraint
(=> (and (inv i l) (=> (= l #x00000004) (= l1 #x00000000)) (=> (not (
    = l #x00000004)) (= l1 l)) (not (or (bvult l1 #x00000000) (bvuge
    l1 #x00000005))) (= l2 (bvadd l1 #x00000001))) (inv i l2))
)

(constraint
(=> (and (inv i l) (=> (= l #x00000004) (= l1 #x00000000)) (=> (not (
    = l #x00000004)) (= l1 l)) (or (bvult l1 #x00000000) (bvuge l1 #
    x00000005))) false)
)

(check-synth)
```

The solution we synthesise for this benchmark is:

```
(not (= (bvlshr (_ bv20 32) 1) (_ bv0 32)))
```

Here is a simpler benchmark from the SyGuS competition, which synthesises a function which computes the maximum of two inputs:

```
(set-logic BV)

(synth-fun max2 ((x (BitVec 32))(y (BitVec 32))) (BitVec 32)
)

(declare-var x (BitVec 32))
(declare-var y (BitVec 32))

(constraint (bvuge (max2 x y) x))
(constraint (bvuge (max2 x y) y))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))

(check-synth)
```

The solution we synthesise for this benchmark is:

```
(ite (bvuge x y) x y)
```

The following two benchmarks are taken from a paper on using Program Synthesis for Program Analysis [34]. The first benchmark requires synthesising a danger invariant, as described in Section 3.1.2 and ranking function:

```
(set-logic BV)

(declare-var x (BitVec 32))
(declare-var y (BitVec 32))


(define-fun G ((x (BitVec 32))) Bool
        (bvult x #x00000006))

(define-fun B_x ((x (BitVec 32))) (BitVec 32)
    (bvadd x #x00000001))

(define-fun B_y ((y (BitVec 32))) (BitVec 32)
    (bvmul y #x00000002))

(define-fun A ((x (BitVec 32))) Bool
    (not (= x #x00000006)))

(synth-fun D ((x (BitVec 32))(y (BitVec 32))) Bool )

(synth-fun R ((x (BitVec 32))(y (BitVec 32))) (BitVec 32))

(constraint
(implies (and (D x y) (G x)) (and (bvugt (R x y) #x00000000) (and (
   bvugt (R x y) (R (B_x x) (B_y y))) (D (B_x x) (B_y y))))))
```

```
(constraint
(implies (and (D x y) (not (G x))) (not (A x))))

(constraint (D #x00000000 #x00000001))

(check-synth)
```

The solution we synthesise for this benchmark is:

```
(define-fun D ((x (BitVec 32))(y (BitVec 32))) Bool
        (not (= (bvlshr (_ bv83 32) |synth::parameter0|) (_ bv0 32)))
           )

(synth-fun R ((x (BitVec 32))(y (BitVec 32))) (BitVec 32)
        (bvlshr (_ bv16384 32) |synth::parameter0|))
```

The following benchmark synthesises a loop invariant, which relies on the overflow semantics of bitvectors:

```
(set-logic BV)

(synth-inv inv-f ((x (BitVec 32))))

(declare-primed-var x (BitVec 32))

(define-fun pre-f ((x (BitVec 32))) Bool
    (= x #x0000000A))

(define-fun trans-f ((x (BitVec 32))(x! (BitVec 32))) Bool
    (and (bvuge x #x0000000A) (= x! (bvadd x #x00000002))))

(define-fun post-f ((x (BitVec 32))) Bool
    (or (= #x00000000 (bvurem x #x00000002))(bvuge x #x0000000A)))

(inv-constraint inv-f pre-f trans-f post-f)
(check-synth)
```

An invariant we synthesize as a solution to this benchmark is:

```
 (not (= (bvshl (bvadd (_ bv31 32) x) (_ bv31 32)) (_ bv0 32)))
```

A.2 Example benchmark for controller synthesis

An example benchmark which synthesizes a safe controller for a Linear Time Invariant system, as described in Section 3.2, is as follows:

```
// Benchmark plant variables
#define NSTATES 5

plant_typet A[5][5]=
{{0.9048374, 0.2100611, 0.02066266, 0.06504802, 0.3755344 }
{0, 0.7788008, 0.09033475, 0.3156122, 2.396729}
{0, 0, 0.0007927521, 0.006308523, 0.5637409}
{0, 0, 0, 0.000003726653, 0.03367110}
{0, 0, 0, 0, 0.006737947}};

plant_typet B[5]=B = {1.985159, 23.32613, 52.22808, 8.898953, 2.979786};

controller_typet K[NSTATES];
plant_typet states[NSTATES];

int main(void)
{
  K = EXPRESSION();
  for(int i=0; i<NSTATES; i++)
  {
    states[i]=nondet();
    assume(states[i] < initial_state_upper_bound);
    assume(states[i] > initial_state_lower_bound);
  }

  assert(check_stability());
  assert(check_safety());
}
```

```
bool check_stability()
{
  calculate_characteristic_polynomial();
  return(jury_criterion_satisfied());
}
```

```
bool check_safety()
{
  for(int i=0; i<NUMBER_LOOPS; i++)
  {
    input = -K * (controller_typet)states;
    if(input > input_upper_bound ||
      input < input_lower_bound)
        return false;

    states = A * states + B * (plant_typet)input;
    for(int i=0; i<NSTATES; i++)
    {
      if(states[i] > safety_upper_bound ||
          states[i] < safety_lower_bound)
          return false;
    }
  }
  return true;
}
```

A solution synthesised for this benchmark is:

```
controller_typet K[5]={-0.002506256, 0.004489899, 0.002985001,
   0.0008187294, 0.02851868e-2};
```

Appendix B

Publications and Contributions

- Counterexample Guided Inductive Synthesis Modulo Theories [5], CAV 2018 – in collaboration with Pascal Kesseli, Daniel Kroening, Cristina David and Alessandro Abate. The work presented in this chapter has subsequently been implemented as part of CVC4 and is available in release 1.7[1]. My contributions to this work are as follows:
  - Implementation of the SyGuS front-end including parsing
  - Implementation of the use of incremental satisfiability within CEGIS, detailed in Chapter 6.
  - Collaboration on development of the general CEGIS($\mathcal{T}$) algorithm with Daniel Kroening
  - Collaboration on the CEGIS($\mathcal{T}$)-FM algorithm with Daniel Kroening
  - Collaboration on writing with Cristina David
  - Accumulation of benchmarks and translation of SyGuS Linear Integer Arithmetic benchmarks into bitvectors
  - Running and write up of the experiments.
- Automated Formal Synthesis of Digital Controllers for State-Space Physical Plants [4], CAV 2017 – with Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli and Daniel Kroening.

  My contributions to this work are as follows:

[1] `https://github.com/CVC4/CVC4`

  - Conversion of the general control problem into a program synthesis problem for the multi-stage back end.
  - Integration of the translated control problem into the multi-stage back end, in close collaboration with Pascal Kesseli.
  - Running of experiments for both the multi-stage and acceleration based back end, in close collaboration with Pascal Kesseli.
  - Write up of the multi-stage back end in collaboration with Cristina David.
  - Formalisation of the maths behind the multi-stage back end, including the proof of stability of closed-loop models with fixed-point controller error

- DSSynth: an automated digital controller synthesis tool for physical plants [2], ASE 2017 – Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas Cordeiro, Cristina David, Pascal Kesseli and Daniel Kroening.

  My contributions to the tool paper presented at ASE are limited to my work on the implementation and experimentation for the CAV 2017 paper, as described above.

- Automated Experiment Design for Data-Efficient Verification of Parametric Markov Decision Processes [113], QEST 2017 – with Viraj Wijesuriya, Sofie Haesaert and Alessandro Abate My contributions to this work are as follows:
  - Lead the direction of research with support from S. Haesaert
  - Developed the main algorithm
  - Implementation and experimental evaluation in collaboration with V. Wijesuriya

- Data-Efficient Bayesian Verification of Parametric Markov Chains [112], QEST 2016 – with Viraj Wijesuriya, Sofie Haesaert and Alessandro Abate My contributions to this work are as follows:
  - Lead the direction of research in collaboration with S. Haesaert
  - Developed the main algorithm in collaboration with S. Haesart
  - Implementation and experimental evaluation

Bibliography

[1] Software verification competition. https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/.

[2] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lennon Chaves, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. DSSynth: an automated digital controller synthesis tool for physical plants. In *International Conference on Automated Software Engineering (ASE)*, pages 919–924. IEEE Computer Society, 2017.

[3] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, and Daniel Kroening. Sound and automated synthesis of digital stabilizing controllers for continuous plants. In *Hybrid Systems: Computation and Control (HSCC)*, pages 197–206. ACM, 2017.

[4] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of digital controllers for state-space physical plants. In *Computer Aided Verification (CAV)*, volume 10426 of *LNCS*, pages 462–482. Springer, 2017.

[5] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2018.

[6] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To encode or to propagate? the best choice for each constraint in SAT. In *CP*, volume 8124 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2013.

[7] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, volume 8044 of *LNCS*, pages 934–950. Springer, 2013.

[8] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, 2013.

[9] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis through unification. In *CAV*, volume 9207 of *LNCS*, pages 163–179. Springer, 2015.

[10] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2017: Results and analysis. *CoRR*, abs/1711.11438, 2017.

[11] Rajeev Alur, Dana Fisman, Rishabh Singh, and Abhishek Udupa. Syntax guided synthesis competition. `http://sygus.seas.upenn.edu/SyGuS-COMP2017.html`, 2017.

[12] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, volume 10205 of *LNCS*, pages 319–336, 2017.

[13] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic verification of control system implementations. In *International Conference on Embedded Software (EMSOFT)*, pages 9–18. ACM, 2010.

[14] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.

[15] K.J. Åström and T. Hägglund. *Advanced PID Control*. ISA-The Instrumentation, Systems, and Automation Society, 2006.

[16] K.J. Åström and B. Wittenmark. *Computer-controlled systems: theory and design*. Prentice Hall information and system sciences series. Prentice Hall, 1997.

[17] Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *DISC*, volume 2848 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2003.

[18] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[20] Clark W. Barrett, Haniel Barbosa, Martin Brain, Duligur Ibeling, Tim King, Paul Meng, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. CVC4 at the SMT competition 2019. *CoRR*, abs/1806.08775, 2019.

[21] C. Belta, B. Yordanov, and E.A. Gol. *Formal methods for discrete-time dynamical systems*. Springer Verlag, 2016.

[22] Iury Bessa, Hussama Ismail, Reinaldo Palhares, Lucas C. Cordeiro, and Joao Edgar Chaves Filho. Formal non-fragile stability verification of digital control systems with uncertainty. *IEEE Trans. Computers*, 66(3):545–552, 2017.

[23] Aart J. C. Bik and Harry A. G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical report, Rijksuniversiteit Leiden, 1994.

[24] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *Symposium on Computer Arithmetic (ARITH)*, pages 160–167. IEEE, 2015.

[25] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the boost interval arithmetic library. *Theor. Comput. Sci.*, 351(1):111–118, 2006.

[26] Gianpiero Cabodi, Paolo Camurati, Alan Mishchenko, Marco Palena, and P. Pasini. SAT solver management strategies in IC3: an experimental approach. *Formal Methods in System Design*, 50(1):39–74, 2017.

[27] Tom Castle and Colin G. Johnson. Evolving high-level imperative program trees with strongly formed genetic programming. In *EuroGP*, volume 7244 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2012.

[28] T. C. Chen, C. Y. Chang, and K. W. Han. Reduction of transfer functions by the stability-equation method. *Journal of the Franklin Institute*, 308(4):389 – 404, 1979.

[29] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, volume 1962, pages 23–35, 1962.

[30] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: Guessing formal specifications using testing. In *TAP*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.

[31] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[32] George E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.

[33] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *CAV (2)*, volume 9780 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2016.

[34] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. Danger invariants. In *Formal Methods (FM)*, volume 9995 of *LNCS*, pages 182–198. Springer, 2016.

[35] Cristina David, Daniel Kroening, and Matt Lewis. Using program synthesis for program analysis. 9450:483–498, 2015.

[36] Iury Valente de Bessa, Hussama Ismail, Lucas C. Cordeiro, and João E. C. Filho. Verification of fixed-point digital controllers using direct and delta forms realizations. *Design Autom. for Emb. Sys.*, 20(2):95–126, 2016.

[37] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[38] Marc Peter Deisenroth and Carl Edward Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *ICML*, pages 465–472. Omnipress, 2011.

[39] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *ICSE*, pages 345–356. ACM, 2016.

[40] P. S. Duggirala and M. Viswanathan. Analyzing real time linear control systems using software verification. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 216–226, Dec 2015.

[41] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[42] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[43] S. Fadali and A. Visioli. *Digital Control Engineering: Analysis and Design*, volume 303 of *Electronics & Electrical*. Elsevier/Academic Press, 2009.

[44] Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019.

[45] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *NDSS*. The Internet Society, 2017.

[46] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435. ACM, 2018.

[47] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, pages 422–436. ACM, 2017.

[48] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In *POPL*, pages 599–612. ACM, 2017.

[49] Ian J Fialho and Tryphon T Georgiou. On stability and performance of sampled-data systems subject to wordlength constraint. *IEEE Transactions on Automatic Control*, 39(12):2476–2481, 1994.

[50] Gene Franklin, David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Pearson, 7th edition, 2015.

[51] Joyce Friedman. Alonzo church. application of recursive arithmetic to the problem of circuit synthesissummaries of talks presented at the summer institute for symbolic logic cornell university, 1957, 2nd edn., communications research division, institute for defense analyses, princeton, nj, 1960, pp. 3–50. 3a-45a. *The Journal of Symbolic Logic*, 28(4):289–290, 1963.

[52] Zoran Gajic, Myo-Taeg Lim, Dobrila Skataric, Wu-Chung Su, and Vojislav Kecman. *Optimal control: weakly coupled systems and applications*. CRC Press, 2008.

[53] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *CAV*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[54] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[55] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512. ACM, 2016.

[56] Chris Gearhart. Genetic programming as policy search in markov decision processes. *Genetic Algorithms and Genetic Programming at Stanford*, pages 61–67, 2003.

[57] Jens Gottlieb, Elena Marchiori, and Claudio Rossi. Evolutionary algorithms for the satisfiability problem. *Evolutionary Computation*, 10(1):35–50, 2002.

[58] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *PLDI*, pages 341–352. ACM, 1992.

[59] Sumit Gulwani. Dimensions in program synthesis. In *FMCAD*, page 1. IEEE, 2010.

[60] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[61] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[62] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[63] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[64] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *POPL*, pages 277–289. ACM, 2007.

[65] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61. ACM, 2011.

[66] Sumit Gulwani and Mark Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD Conference*, pages 803–814. ACM, 2014.

[67] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. *ACM SIGPLAN Notices*, 43(6):281–292, 2008.

[68] Fatima Zohra Hadjam and Claudio Moraga. RIMEP2: evolutionary design of reversible digital circuits. *JETC*, 11(3):27:1–27:23, 2014.

[69] John N. Hooker. Solving the incremental satisfiability problem. *J. Log. Program.*, 15(1&2):177–186, 1993.

[70] Roger A Horn and Charles Johnson. *Matrix Analysis*. Cambridge University Press, 1990.

[71] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46. ACM, 2010.

[72] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.

[73] Susmit Jha and Sanjit A. Seshia. Are there good mistakes? A theoretical analysis of CEGIS. In *SYNT*, volume 157 of *EPTCS*, pages 84–99, 2014.

[74] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005.

[75] Vladimir Jojic, Sumit Gulwani, and Nebojsa Jojic. Probabilistic inference of programs from input/output examples. Technical report, MSR-TR-2006-103), July, 2006.

[76] Henry S. Warren Jr. *Hacker's Delight, Second Edition*. Pearson Education, 2013.

[77] Susumu Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI*, volume 3157 of *Lecture Notes in Computer Science*, pages 75–84. Springer, 2004.

[78] Susumu Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 111–126. Intellect, 2005.

[79] Gal Katz and Doron A. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.

[80] Petar V. Kokotovic, John J. Allemong, James R. Winkelman, and Joe H. Chow. Singular perturbation and iterative separation of time scales. *Automatica*, 16(1):23–33, 1980.

[81] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

[82] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1st edition, 2008.

[83] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[84] William B. Langdon and Riccardo Poli. *Foundations of genetic programming*. Springer, 2002.

[85] Tessa A. Lau, Pedro M. Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43. ACM, 2003.

[86] Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[87] Daniel Liberzon. Hybrid feedback stabilization of systems with quantized signals. *Automatica*, 39(9):1543–1554, 2003.

[88] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Feature model synthesis with genetic programming. In *SSBSE*, volume 8636 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2014.

[89] J. Liu and N. Ozay. Finite abstractions with robustness margins for temporal logic-based control synthesis. *Nonlinear Analysis: Hybrid Systems*, 22:1–15, 2016.

[90] Moshe Looks, Ben Goertzel, and Cassio Pennachin. Learning computer programs with the bayesian optimization algorithm. In *GECCO*, pages 747–748. ACM, 2005.

[91] William Luyben. External versus internal open-loop unstable processes. *Ind. Eng. Chem. Res.*, 7(3):2713–2720, Jun 1998.

[92] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[93] Henry Massalin. Superoptimizer - A look at the smallest program. In *ASPLOS*, pages 122–126. ACM Press, 1987.

[94] Manuel Mazo, Jr., Anna Davitian, and Paulo Tabuada. PESSOA: A tool for embedded controller synthesis. In *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 566–569. Springer, 2010.

[95] Richard H Middleton and Graham C Goodwin. *Digital control and estimation: a unified approach*. Prentice Hall Professional Technical Reference, 1990.

[96] Tom M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2):203–226, 1982.

[97] Ramon E Moore. *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.

[98] Vojtech Mrazek and Zdenek Vasícek. Evolutionary design of transistor level digital circuits using discrete simulation. In *EuroGP*, volume 9025 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2015.

[99] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012.

[100] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 256–269. Springer, 2012.

[101] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately incremental SAT. In *SAT*, volume 8561 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2014.

[102] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *ICSE*, pages 772–781. IEEE Computer Society, 2013.

[103] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[104] V. A. Oliveira, E. F. Costa, and J. B. Vargas. Digital implementation of a magnetic suspension control system for laboratory experiments. *IEEE Transactions on Education*, 42(4):315–322, Nov 1999.

[105] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630. ACM, 2015.

[106] Abdelkrim K Oudjida, Nicolas Chaillet, Ahmed Liacha, Mohamed L Berrandjia, and Mustapha Hamerlain. Design of high-speed and low-power finite-word-length PID controllers. *Control Theory and Technology*, 12(1):68–83, 2014.

[107] Saswat Padhi and Todd D. Millstein. Data-driven loop invariant inference with automatic feature synthesis. *CoRR*, abs/1707.02029, 2017.

[108] Junkil Park, Miroslav Pajic, Insup Lee, and Oleg Sokolsky. Scalable verification of linear controller software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 662–679. Springer, 2016.

[109] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, pages 408–418. ACM, 2014.

[110] Juan Antonio Navarro Pérez and Andrei Voronkov. Encodings of problems in effectively propositional logic. In *SAT*, volume 4501 of *Lecture Notes in Computer Science*, page 3. Springer, 2007.

[111] Bruno Picasso and Antonio Bicchi. Stabilization of LTI systems with quantized state-quantized input static feedback. In *International Workshop on Hybrid Systems: Computation and Control*, pages 405–416. Springer, 2003.

[112] Elizabeth Polgreen, Viraj B. Wijesuriya, Sofie Haesaert, and Alessandro Abate. Data-efficient bayesian verification of parametric markov chains. In *QEST*, volume 9826 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2016.

[113] Elizabeth Polgreen, Viraj B. Wijesuriya, Sofie Haesaert, and Alessandro Abate. Automated experiment design for data-efficient verification of parametric markov decision processes. In *QEST*, volume 10503 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2017.

[114] S. Pramod and M. Chidambaram. Closed loop identification of transfer function model for unstable bioreactors for tuning PID controllers. *Bioprocess Engineering*, 22(2):185–188, Feb 2000.

[115] Mukund Raghothaman, Andrew Reynolds, and Abishek Udupa. *The SyGuS Language Standard Version 2.0*, 2019 (accessed July 1st, 2019).

[116] Mukund Raghothaman and Abhishek Udupa. Language to specify syntax-guided synthesis problems. *CoRR*, abs/1405.5590, 2014.

[117] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Counter-example guided synthesis of control Lyapunov functions for switched systems. In *Conference on Decision and Control (CDC)*, pages 4232–4239, 2015.

[118] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Robust controller synthesis of switched systems using counterexample guided framework. In *International Conference on Embedded Software (EMSOFT)*, pages 8:1–8:10. ACM, 2016.

[119] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis.

In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[120] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV (2)*, volume 9207 of *LNCS*, pages 198–216. Springer, 2015.

[121] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-based synthesis in smt. *Formal Methods in System Design*, 2017.

[122] Andrew Reynolds, Arjun Viswanathan, Haniel Barbosa, Cesare Tinelli, and Clark Barrett. Datatypes with shared selectors. volume 10900, pages 591–608. Springer, 2018.

[123] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.

[124] Eric Rosen. An existential fragment of second order logic. *Arch. Math. Log.*, 38(4-5):217–234, 1999.

[125] Stéphane Ross, Joelle Pineau, Brahim Chaib-draa, and Pierre Kreitmann. A bayesian approach for learning and planning in partially observable markov decision processes. *Journal of Machine Learning Research*, 12:1729–1770, 2011.

[126] Pierre Roux, Romain Jobredeaux, and Pierre-Loïc Garoche. Closed loop analysis of control command software. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 108–117. ACM, 2015.

[127] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Successful use of incremental BMC in the automotive industry. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 62–77. Springer, 2015.

[128] Lukás Sekanina. Evolutionary design of digital circuits: Where are current limits? In *AHS*, pages 171–178. IEEE Computer Society, 2006.

[129] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446. AAAI Press / The MIT Press, 1992.

[130] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81(1-2):17–29, 1996.

[131] Daniel Sheridan. Comparing SAT encodings for model checking. In *CP*, volume 2239 of *Lecture Notes in Computer Science*, page 784. Springer, 2001.

[132] Daniel Sheridan. *Temporal logic encodings for SAT-based bounded model checking.* PhD thesis, University of Edinburgh, UK, 2006.

[133] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26. ACM, 2013.

[134] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[135] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.

[136] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[137] M. W. Spong. The swing up control problem for the Acrobot. *IEEE Control Systems*, 15(1):49–55, Feb 1995.

[138] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S Foster. Path-based inductive synthesis for program inversion. In *ACM SIGPLAN Notices*, volume 46, pages 492–503. ACM, 2011.

[139] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.

[140] Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In *FMCAD*, volume 2517 of *LNCS*, pages 160–170. Springer, 2002.

[141] P. Tabuada. *Verification and control of hybrid systems: a symbolic approach.* 2009.

[142] F. Tadeo, O. P. Lopez, and T. Alvarez. Control of neutralization processes by robust loop shaping. *IEEE Transactions on Control Systems Technology*, 8(2):236–246, Mar 2000.

[143] R. H. G. Tan and L. Y. H. Hoo. DC-DC converter modeling and simulation using state space approach. In *Conference on Energy Conversion (CENCON)*, pages 42–47. IEEE, Oct 2015.

[144] Robert Thomson. *The evolutionary design of digital VLSI hardware*. PhD thesis, University of Edinburgh, UK, 2005.

[145] Marc Toussaint and Amos J. Storkey. Probabilistic inference for solving discrete and continuous state markov decision processes. In *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 945–952. ACM, 2006.

[146] G Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[147] Charles Van Loan. Computing integrals involving the matrix exponential. *IEEE transactions on automatic control*, 23(3):395–404, 1978.

[148] Timothy E. Wang, Pierre-Loïc Garoche, Pierre Roux, Romain Jobredeaux, and Eric Feron. Formal analysis of robustness at model and code level. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 125–134. ACM, 2016.

[149] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374. IEEE, 2009.

[150] Jun Wu, Gang Li, Sheng Chen, and Jian Chu. Robust finite word length controller design. *Automatica*, 45(12):2850–2856, 2009.

[151] Tina Yu. Polymorphism and genetic programming. In *EuroGP*, volume 2038 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2001.

[152] M. Zamani, M. Mazo, and A. Abate. Finite abstractions of networked control systems. In *Conference on Decision and Control (CDC)*, pages 95–100. IEEE, 2014.

[153] Luke S. Zettlemoyer and Michael Collins. Learning context-dependent mappings from sentences to logical form. In *ACL/IJCNLP*, pages 976–984. The Association for Computer Linguistics, 2009.

[154] Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, volume 10416 of *Lecture Notes in Computer Science*, pages 671–686. Springer, 2017.

[155] Zahra Zojaji, Behrouz Tork Ladani, and Alireza Khalilian. Automated program repair using genetic programming and model checking. *Appl. Intell.*, 45(4):1066–1088, 2016.