Synthesis of Protocols and Discrete Controllers

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy by

Idress Mohammed Husien

November 2017

Dedication

To My family

Contents

Illustrations

List of Figures

List of Tables

Notations

The following notations and abbreviations are found throughout this thesis:

SA	Simulated Annealing
RSA	Rigid Simulated Annealing
FSA	Flexible Simulated Annealing
GP	Genetic Programming
HGP	Hybrid Genetic Programming
W/O	Genetic Programming without crossover
DCS	Discrete Controller Synthesis
W/C	Genetic Programming with crossover
MC	Model Checking
CTL	Computational Tree Logic
LTL	Linear Temporal Logic

Preface

I declare that this thesis is composed by myself and that the work contained herein is my own, except where explicitly stated otherwise, and that this work was undertaken by me during my period of study at the University of Liverpool, United Kingdom. This thesis has not been submitted for any other degree or qualification except as specified here.

Abstract

In this thesis, a number of search techniques are proposed as a solution for program and discrete controller synthesis (DCS). Classic synthesis techniques facilitate exhaustive search, while genetic programming has recently proven the potential of generic search techniques. But is genetic programming the right search technique for the synthesis problem? In this thesis we challenge this belief and argue in favor of simulated annealing, a different class of general search techniques. We show that, in hindsight, the success of genetic programming has drawn from what is arguably a hybrid between simulated annealing and genetic programming, and compare the fitness of classic genetic programming, the hybrid form, and pure simulated annealing. Our experimental evaluation suggests that pure simulated annealing offers better results for automated programming than techniques based on genetic programming.

Discrete Controller Synthesis (DCS) and Program Synthesis have similar goals: they are automated techniques to infer a control strategy and an implementation, respectively, that is correct by construction. We also investigate the application of the search techniques that we have been used for program synthesis for the computation of deterministic strategies solving symbolic Discrete Controller Synthesis (DCS) problems, where a model of the system under control is given along with desired objective behaviours. We experimentally confirm that relative performance results are similar to program synthesis, and give a complexity analysis of our simulated annealing algorithm for symbolic DCS. From the performance results we obtain, we draw the conclusion that simulated annealing, when combined with efficient model-checking techniques, is worth further investigating to solve symbolic DCS problems.

A tool is designed to explore the parameter space of different synthesis techniques. Besides using it to synthesise a discrete control strategies for reactive systems (controller synthesis) and for protocol adapters for the coordination of different threads (software synthesis), we can also use it to study the influence of turning various screws in the synthesis process. For simulated annealing, PranCS allows the user to define the behaviour of the cooling schedule. For genetic programming, the user can select the population size.

Acknowledgements

I would like to express my deepest gratitude to my first supervisor, Sven Schewe, specially for his support, constant patience and encouragement throughout my years of study. He has been very supportive in many different ways and has constantly encouraged me during all these years. His immense knowledge has always helped me to learn a lot from him. His friendly behaviour and positive attitude are exemplary and I regard it as my honour to have done my Ph.D. under his excellent supervision.

I would like to thank Alexei Lisitsa, and David Jackson, for being my academic advisors, Dominik Wojtczak, for being my second supervisor and Special thanks also go to Nicolas Berthier for his support. I thank them for all their advice, useful ideas and support.

I would like to extend my deepest gratitude to the Ministry of higher Education in Iraq especially the University of Kirkuk and Iraqi Cultural Attaché in London for their financial support and for providing me with the opportunity to conduct my Ph.D. study.

I would like to thank the Department of Computer Science at the University of Liverpool has been an excellent place to conduct research; all staff members and colleagues have been helpful whenever necessary. Finally, I thank my family, for all their support, trust, encouragement and prayers.

Chapter 1

Introduction

1.1 Overview

The development of correct code can be quite challenging, especially for concurrent systems. Classical software engineering methods, where the validation is based on testing, do not seem to provide the right way to approach this type of involved problems, as bugs easily elude predefined tests. Guaranteeing correctness for such programs is also not trivial. Manual proof methods for verifying the correctness of the code against a given formal specification were suggested in the late 60s.

The next step for achieving more reliable software has been to offer an automatic verification procedure through model checking [CGP99, BCM+90, AHM+98]. The holy grail of such techniques would be automatic synthesis: the automated construction of programs that are correct by construction.

Such synthesis techniques have long been held to be impossible due to complexity (which ranges between EXPTIME for CTL synthesis [CE82] and undecidable [PR90, FS05, SF06] for distributed systems. This line of thought has come under attack on many fronts.

On the theoretical side, bounded [FS13] and succinct [FPS15] synthesis techniques have leveled the playing field between verification and synthesis by shifting the focus from the input complexity to the cost measured in the minimal explicit and symbolic solution, respectively. One could argue that this is the theoretical foundation of successful approaches, including implementations of bounded synthesis [FJR09, Ehl11] and methods based on genetic algorithms [Joh07, KP08, KP09a].

In this work we suggest to use simulated annealing for program synthesis and compare it to similar approaches based on genetic algorithms. We use formal verification technique (model checking) as a way for assessing fitness in an inductive automatic programming system.

We have implemented a synthesising tool, which uses (multiple calls to) the model checker NuSMV to determine the fitness for a candidate program. The candidate programs exist in two forms. The main form is a simple imperative language. This form is subject to mutation, but it is translated to a secondary form, the modeling language

of NuSMV, for evaluating the fitness. We have implemented different selection and update mechanism to compare the performance of simulated annealing with genetic programming. Genetic programming represented in this work with and without applying crossover. A hybrid genetic programming method also applied in this work beside simulated annealing.

The remainder of this introductory chapter is organised as follows. Section 1.2 presents the motivation for the work presented in this thesis. Section 1.3 describes the main research question and the associated research issues to be addressed by the thesis. The adopted research methodology is presented in Section 1.4. Section 1.5 describes the contributions of the work presented. The organisation of the remainder of this thesis is presented in Section 1.6. Section 1.7 lists the publications resulting from the research presented in this thesis. Finally this chapter is concluded in Section 1.8 with a brief summary.

1.2 Motivation

The main aim of the work presented in this thesis is to investigate and evaluate effective algorithms that will be used for code generation especially for concurrent programming and for discrete controller synthesis. The motivation for this work is that the synthesis of programs and discrete controllers is desirable because it provides the following :

1. It can be used for generating correct code, which can be quite challenging especially for concurrent systems, that cannot be obtained efficiently by classical software engineering methods.

2. It allows to correct the errors in faulty code by verifying a desired specifications using model checking, which takes a model and properties as input and check if the model satisfies the objectives or not, and lets the implementation evolve into correct code.

3. It can also be used for the computation of deterministic strategies solving symbolic Discrete Controller Synthesis (DCS) problems, where a model of the system to control is given along with desired objective behaviours.

The challenge in automatic programming is synthesizing programs automatically from their set of requirements. Search techniques in which the fitness of each program is usually calculated by running the program on some test cases, and evaluating its performance. Orthogonally, model checking can be used to analyze a given program, verifying that it satisfies its specification, or providing a counterexample of that fact.

Discrete Controller Synthesis (DCS) and Program Synthesis have similar goals in that they are constructive methods for behaviour control. Discrete controller synthesis typically operates on the model of a plant, and seeks the automated construction of a strategy so that the plant controlled accordingly satisfies a specific set of specifications. Likewise, program synthesis operates by using some predefined rules, such as the

grammar and semantics of the target programming language, and seeks the automated construction of a program whose execution satisfies a specific set of specifications.

As noted above the search techniques can be used together the model checking as a way to synthesis both programs and discrete controllers. The motivation of the work described in this thesis can thus broadly be identified as the desire to develop a synthesis technique that has low cost and can be used for both programs and discrete controllers.

1.3 Research Question

From the research motivation presented in Section 1.2, the key objective of the work presented in this thesis is to research and investigate effective and efficient technique for program and controller synthesis. This objective can be formulated as a research question as follows:

What are the most appropriate search technique that can be used with model checking (as a fitness function) for program so as control synthesis

In order to answer this research question the resolution of a number of sub-question is required. These questions can be summarized as follows:

- **General Search Technique:** What is the best search technique that can be used for program synthesis? Is genetic programming, simulated annealing, or a hybrid of them better? Can we implement them in another way as hybrid methods?
- **Search Techniques Parameters:** What are the efficient parameters that can be used for genetic programming and simulated annealing? Is the application of crossover effective? How does initial population size affect the cost beside the success rate? How does the initial temperature and cooling schedule affect the results of simulated annealing? Are there 'best' parameters for our techniques?
- **Controller Synthesis:** Can we use the same techniques in controller synthesis? Are the results similar to those of program synthesis? What about the parameters do they have the same effect?
- **Synthesis Tool:** What is the scope of a synthesis tool based on generic search technique?

1.4 Research Methodology

As described in the previous sections, the adopted research methodology of the work described in this thesis is to investigate and evaluate a series of techniques used for program and controller synthesis. For this purpose, simulated annealing is used for program synthesis and compared it to similar approaches based on genetic programming.

We use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system. We have implemented a synthesis tool,

which uses multiple calls to the model checker NuSMV [CCG+02] to determine the fitness for a candidate. The candidates exist in two forms.

1. The main form is a simple imperative language. This form is subject to mutation, in which the program represented as a binary tree. The leaf nodes in the program tree is the variables or constant. The program tree translated to
2. a secondary form, the modeling language of NuSMV, for evaluating its fitness.

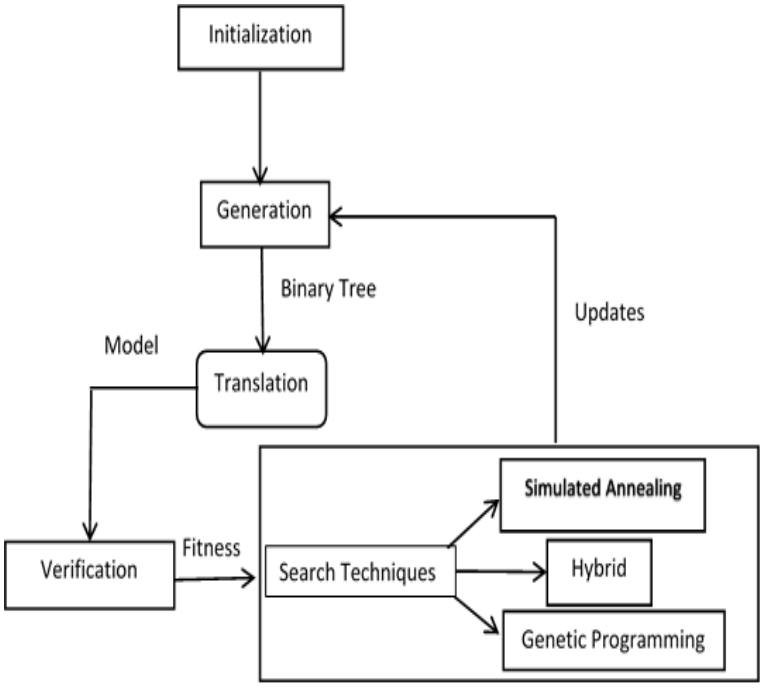


FIGURE 1.1: Work Flow Chart

While there has been further research on how to measure partial satisfaction [HO13], we have adopted an approach that retains with the previous attempts:

1. The best choice for us is to keep to the choices made for promoting genetic programming [KP08, KP09a, KP09b], as this is the only choice that is completely free of suspicion of being selected for being more suitable for simulated annealing than for genetic programming.
2. A second motivation for this selection is that it results in very simple specifications and, therefore, in fast evaluations of the fitness.

Noting that synthesis entails on average hundreds of thousands to millions of calls to a model checker, only simple evaluations can be considered. We have implemented six different combinations of selection and update mechanism to test our hypothesis:

1. Simulated annealing with two different fitness measure.
2. Genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover.

3. Hybrid genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover.

At the beginning all the parameters initialized, then the initial population generated by the generation part of the our software, initially the candidates represented as a binary trees. The candidates trees used as input for the translator, which convert them into a NuSMV models to evaluate those candidates. The model checker use as fitness function for the search techniques.

1.5 Research Contribution

1.5.1 Program Synthesis

In this part of our work we implement the following approaches:

1. We use simulated annealing for program synthesis and compare it to similar approaches based on genetic programming.

2. We use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system. For this purpose we give a score for a candidate depending on the satisfied specifications.

3. We have implemented a synthesis tool, which uses multiple calls to the model checker NuSMV [CCG$^+$02] to determine the fitness for a candidate program. The tool enables the user to input the requirements (as temporal logic specifications)of the programs (protocols) that (s)he wants to generate, and the input variables for the algorithms.

4. We have implemented six different combinations of selection and update mechanism to test our hypothesis: besides simulated annealing, we have used genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover and hybrid genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover. Simulated annealing also implemented in two forms depending on the sharing of specifications. If the safety specification considered first then we refer to simulated annealing as flexible otherwise it will be knowns as rigid simulated annealing.

The tests we have run confirmed that simulated annealing performs significantly better than genetic programming. As a side result, we found that the assumption of Katz and Peled [KP08, KP09a, KP09b] that crossover does not accelerate genetic programming did not prove to be entirely correct, but the advantages we observed were minor.

1.5.2 Controller synthesis

1. We first define a symbolic model and an associated class of DCS problems, for which deterministic strategies are sought.

2. Next, we adapt the aforementioned search techniques to obtain algorithmic solutions that avoid computing the unsafe portion of the state-space.

3. Then, we confirm the hypotheses that (i) general search techniques are as applicable to solve our DCS problem as they are for synthesising programs; and

   (ii) one obtains similar relative performance results for our DCS problem. Experimental results [HS16] for program synthesis, essentially that simulated annealing performs better than genetic programming.

4. To assess these hypotheses, we adapt the six different combinations of candidate selection and update mechanisms of our previous work [HS16], and execute them on a scalable example DCS problem.

5. We perform an experimental feasibility assessment.

From the performance results we obtain, we draw the conclusion that, even though for technical reasons our current experimental results do not compare favourably with existing symbolic DCS tools, simulated annealing, when combined with efficient model-checking techniques, is worth further investigating to solve symbolic DCS problems.

1.6 The Organization of Thesis

The rest of the thesis delves in detail into the work behind each of the contributions, and is organized as follows:

- In Chapter 2 We present a literature review of related research and some background material to the work on the search techniques explained in this thesis.

- Chapter 3 presents the adaptation of the general search techniques to solve the program synthesis problem and an extension of the quest for the best general search technique by studying the effect of the parameter settings for the individual search techniques: the influence of the selected temperature for simulated annealing and the crossover ratio for genetic programming.

- Chapter 4 summarizes how to use the general searching techniques for the computation of deterministic strategies solving symbolic Discrete Controller Synthesis (DCS) problems, where a model of the system to control is given along with desired objective behaviours.

- Chapter 5 present the analysis of complexity of the approaches applied in this work, the complexity analysis considered for both program and discrete controller synthesis aspects.

- Chapter 6 provides the description of our PranCS tool, which implements the simulated annealing based approach proposed in [HS16, HBS17] as well as similar genetic programming based approaches from [KP08] and [KP09a].

PranCS is based on quantitative measures for partial compliance with a specification, which serve as a measure for the fitness (or: quality) of a candidate solution.

- Chapter 7 begins by presenting some conclusions, then lists the main findings of the work presented in this thesis.

Note: Some concepts appear several times to make the technical chapters independently accessible.

1.7 Publication

Four papers, two published, one of them selected as the best paper among 25 papers on SEFM16, and two under review, have arisen out the work presented in this thesis, and these are listed and described in this section:

1. Journal Papers:

   - Idress Husien and Sven Schewe Program Generation Using Simulated Annealing and Model Checking. submitted to the Journal of Logical and Algebraic Methods in Programming (JLAMP).

     This article summarises the application of the synthesis tool on code generation. We extend the quest for the best general search technique by studying the effect of the parameter settings for the individual search techniques: the influence of the selected temperature for simulated annealing and the crossover ratio for genetic programming.

     We found that the advantage in the individual execution time between the classic and the hybrid version of genetic programming is in the range that is to be expected, as the number of calls to the model checker is reduced. It is interesting to note that simulated annealing, where the shift from rigid to flexible evaluation might be expected to have a similar effect, does not benefit to the same extent. It is also interesting to note that the execution time suggests that determining the fitness of programs produced by simulated annealing is slightly more expensive.

     This journal article comprising an extended, updated and revised version of a paper [HS16] that appeared in the 14^{th} International Conference on Software Engineering and Formal Methods (SEFM 2016) (see below). The work of this article included in Chapters 2, 3, and 5.

2. Conference Papers:

   - Idress Husien and Sven Schewe Program Generation Using Simulated Annealing and Model Checking, Software Engineering and Formal Methods - 14th International Conference, (SEFM 2016) (155-171), Springer 2016.

In this paper we challenge the belief that simulated annealing is the right search technique that can be used with model checking for synthesis and argue in favour of simulated annealing comparing with genetic programming, a different class of general search techniques. We show that, in hindsight, the success of genetic programming has drawn from what is arguably a hybrid between simulated annealing and genetic programming, and compare the fitness of classic genetic programming, the hybrid form, and pure simulated annealing.

Our experimental evaluation suggests that pure simulated annealing offers better results for automated programming than techniques based on genetic programming. In our view, the performance is naturally sensitive to the quality of the integration, the suitability of the model checker used, and hidden details, like how the seed is chosen or details of how the fitness is computed. The integrated comparison makes sure that all methods are on equal footage in these regards. The work of this paper included in Chapters 2, 3, and 5.

- Idress Husien, Nicolas Berthier and Sven Schewe. A Hot Method for Synthesising Cool Controllers, In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017, pages 122-131, New York, NY, USA, 2017. ACM.

  In this paper, we investigate the application of our searching techniques for the computation of deterministic strategies solving symbolic Discrete Controller Synthesis (DCS) problems, where a model of the system to control is given along with desired objective behaviours. We experimentally confirm that relative performance results are similar to program synthesis. our experimental results do not compare favourably with existing symbolic DCS tools. Yet, our implementations are proofs of concept, and one can think of numerous practical improvements that constitute inescapable ways to pursue investigating efcient symbolic DCS algorithms using simulated annealing. For instance, canonically representing symbolic candidate strategies using BDDs instead of syntactic trees would allow building a cache of ftness results, and thereby avoid re-evaluating the ftness of equivalent candidate strategies.

  At last, note that our search based algorithms do not require the computation of the unsafe region to produce deterministic strategies. The work of this paper is included in Chapters 2 and 4 and 6.

- Idress Husien, Nicolas Berthier and Sven Schewe. PranCS: protocol Controller Synthesis Tool. In Proceeding of Symposium on Dependable Software Engineering Theories, Tools and Applications SETTA 2017 Changsha, China, Pages(337-347), Springer.

  This paper summarises the synthesis tool. Our **Pr**otocol **an**d **C**ontroller **S**ynthesis (PranCS) tool is designed to explore the parameter space of different synthesis techniques. Besides using it to synthesise a discrete control strategies

for reactive systems (controller synthesis) and for protocol adapters for the coordination of different threads, we can also use it to study the influence of turning various screws in the synthesis process.

For simulated annealing, PranCS allows the user to define the behaviour of the cooling schedule. the evaluation of PranCS indicates that simulated annealing is faster than genetic programming, and that some temperature ranges are more useful than others. The work of this paper is included in Chapter 5

1.8 Summery

In summary, this chapter has provided an overview, and some background, for the research presented in the remainder of this thesis, including details concerning the motivations for the work and the research question and subsidiary questions. It has also provided a brief description of the research methodology and the contributions of the research. In the following chapter, a literature review, intended to provide more detail regarding the background concerning the research described in the thesis, is presented.

Chapter 2

Background

2.1 Introduction

Generating new programs—or correcting existing ones—for a specific problem can be a quite challenging, especially for concurrent systems. Classical software engineering methods, where the validation is based on testing, do not seem to suffice for this problem, as an error might depend on the order of context switches. In such cases, it might not occur in test cases, and even if it is caught, it might not be reproducible.

To strengthen tests, manual proof methods for verifying the correctness of the code against a given formal specification were suggested in the late 60s. The next step for achieving more reliable software has been to offer an automatic verification procedure through model checking [CGP99, BCM+90, AHM+98]. While this line of research has improved the power to *validate* the correctness of software and is useful for debugging existing code, it does not help to build correct code or protocols.

Synthesis—the automated construction of programs that are correct by construction—however, has long been held to be impossible due to complexity (which ranges between EXPTIME for CTL synthesis [CE82] and undecidable for distributed systems [PR90, KV01, FS05, SF06]).

This line of thought has been challenged both theoretically—through the introduction of bounded [FS13] and succinct [FPS15] synthesis techniques—and through the development of an increasing number of tools, including implementations of bounded synthesis [FJR09, Ehl11] and methods based on genetic algorithms [Joh07, KP08, KP09a].

2.2 Search Techniques

We investigate two general search techniques, namely *simulated annealing* and *genetic programming*, and derive a *hybrid* one. We present these techniques, and turn to their application in combination with model-checking to find deterministic strategies in the following section. Let us now elaborate on the each of the search techniques we shall experiment with.

2.2.1 Simulated Annealing

Simulated annealing [CJ01, HJJ03] is a general local search technique that is able to escape from local optima, is easy to implement, and has good convergence properties. When applied to an optimisation problem, the fitness function (objective) generates values for the quality of the solution constructed in each iteration.

The fitness of this newly selected solution is then compared with the fitness of the solution from the previous round. Improved solutions are always accepted, while some of the other solutions are accepted in the hope of escaping local optima in search of global optima.

The probability of accepting solutions with reduced fitness depends on a temperature parameter, which is typically falling monotonically with each iteration of the algorithm.

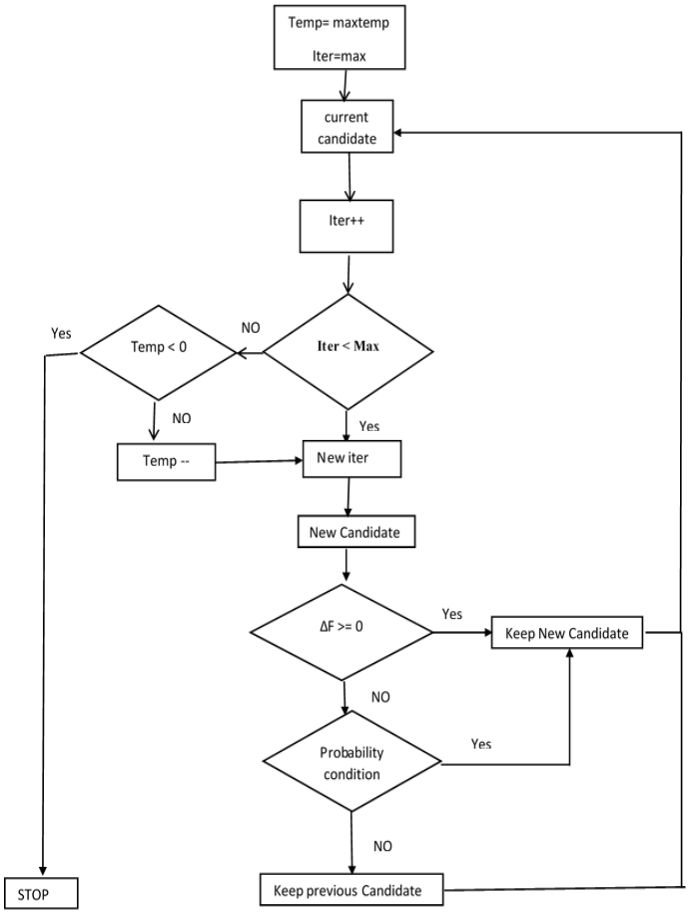


FIGURE 2.1: Simulated Annealing Flow Chart

Simulated annealing starts with an initial candidate solution. In each iteration, a neighboring solution is generated by mutating the previous solution. Let, for the i^{th} iteration, F_{i-1} be the fitness of the 'old' solution and F_i the fitness of its mutation constructed in the i^{th} iteration. If the fitness is not decreased ($F_i \geq F_{i-1}$), then the mutated solution is kept. If the fitness is decreased ($F_i < F_{i-1}$), then the probability p

that this mutated solution is kept is

$$p = e^{\frac{F_i - F_{i-1}}{T_i}} ,$$

where T_i is the temperature parameter for the i^{th} step. The chance of changing to a mutation with smaller fitness is therefore reduced with an increasing gap in the fitness, but also with a falling temperature parameter. The temperature parameter is positive and usually non-increasing ($0 < T_i \leq T_{i-1}$). The development of the sequence T_i is referred to as the *cooling schedule* and inspired by cooling in the physical world [HJJ03].

Algorithm 1: Simulated Annealing algorithm

```
Set the iteration value i := 0
Set the initial temperature T to very high value
loop cooling
loop Local search
Randomly derive initial solution x
repeat
  i := i + 1
  derive a neighbour solution x' of x
  ΔF := F(x') − F(x)
  if ΔF >= 0 then
    x := x'
  else
    derive random number p ∈ [0, 1]
    if p < e^(ΔF / T(i)) then
      x := x'
    end if
  end if
  end loop Local search
until the goal is reached or i = i_max
end loop Cooling
```

The effect of *cooling* on the simulation of annealing is that the probability of following an unfavorable move is reduced. In practice, the temperature is often decreased in stages. During each stage the temperature is kept constant until a balanced solution is reached. The set of parameters that determines how the temperature is reduced (i.e., the initial temperature, the stopping criterion, the temperature decrements between successive stages, and the number of transitions for each temperature value) is called the cooling schedule. We have used a simple cooling schedule, where the temperature is dropped by a constant in each iteration. The algorithm is described in Algorithm 1.

2.2.2 Genetic Programming

Genetic programming [Koz92] is a different general search technique that has been used for program synthesis in a similar setting [Joh07, KP08, KP09a, KP09b]. In genetic programming, a population of λ candidate programs is first generated randomly. In each

step, μ candidates (with $\mu \ll \lambda$) selected from the main popualtion according on their fitness value.

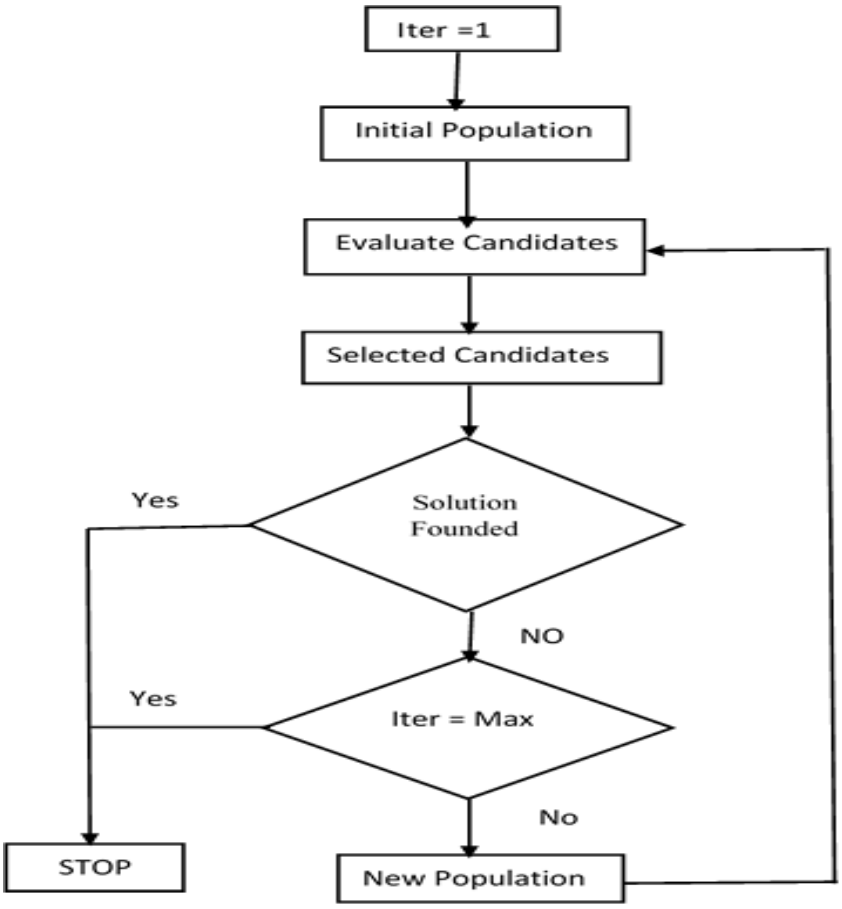


FIGURE 2.2: Genetic Programming Flow Chart

Genetic programming used with computer programs, which is represented as trees. Trees can be easly evaluated and represented, each node in the tree node has an operator function and every terminal node has an operand, making mathematical expressions easy to evolve and evaluate. Figure 2.3 shows an example of candidate tree.

We have implemented genetic programming as a comparison point, using the values $\lambda = 150$ and $\mu = 5$ from [KP08]. We also use the $2,000$ iterations suggested there as a cut-off point, where the algorithm is re-started. In its pure form, it uses the sum of the partial satisfaction values of all sub-specifications as a foundation of the fitness function.

We have additionally implemented a hybrid form that changes the selection technique over time. This technique works in layers: it first establish the safety properties, and then the liveness properties. Specifications with better values for the safety properties are always given preference, while liveness properties are—for equal values for the safety properties—used to determine the fitness. I.e., they are merely tie-breakers.

This approach has been used in [KP08, KP09a, KP09b]. We refer to it as a *hybrid approach* as it introduces a property known from simulated annealing: in the beginning, the algorithm is applying changes more flexibly, while it becomes more rigid later.

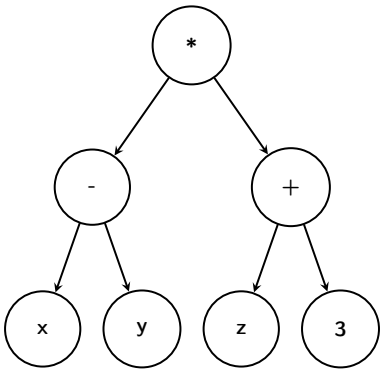


FIGURE 2.3: GP Candidate Tree

We have implemented the genetic approaches with and without crossover, and used both evaluation techniques for simulated annealing, where we refer to using the classic fitness function as a *rigid* evaluation, and to the hybrid approach as *flexible* evaluation.

2.2.3 Initialization

The generation of the initial population has a significant effect on the GP performance. Populations with poor diversity may have a negative effect on finding a correct solution. On the otherhand, the initial population size can increase the cost of finding a correct solution. Initial populations are typically randomly generated. Here, we will describe grow method, which we have used in our work. In the grow method, nodes are taken randomly from both function and terminal sets until the generator selects enough terminals to end the tree or it reaches the maximal depth. The grow method is known to produce trees of irregular shapes [Koz92]. Similar to the full tree generator, the problem with the grow method is that the shape of the trees with the grow method is directly influenced by the size of the function and the terminal sets.

2.3 Genetic Operators

2.3.1 Mutation

The main operators of genetic programming are mutation and crossover.

Mutation only work with one parent. A mutation point is selected randomlly, the sub-tree rooted by the selected node replaced by another sub-tree of the same type. The new sub-tree is generated randomly. Mutation can also be applied on one node randomly selected from the leaf or inner nodes. See Figure 2.4 and Figure 2.5.

2.3.2 Crossover

The main aim of crossover Figure 2.6 is to combine the nodes of two parents by exchanging some nodes from one parent tree with some from the other. The most commonly used type of crossover is sub-tree crossover. In sub-tree crossover, the GP system selects

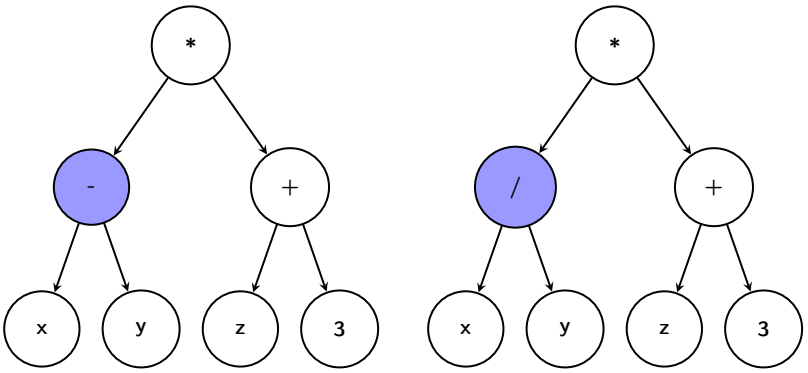


FIGURE 2.4: Candidate tree (left) with one node mutations (right)

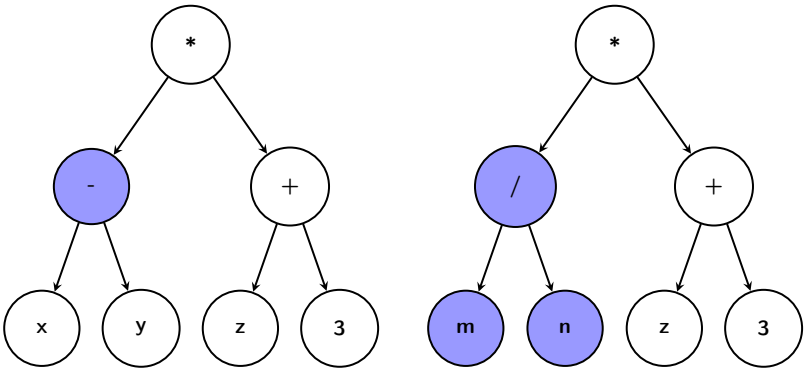


FIGURE 2.5: Program tree (left) with sub-tree mutations (right)

two trees. The system randomly selects two crossover points in each parent and swaps the sub-trees rooted there. Then, it generates a new offspring, which consists of parts of the two selected parents [Koz92]. Therefore, the crossover points are selected randomly and independently.

2.4 Hybrid Genetic Programming

A part from simulated annealing and pure genetic programming with and without crossovers as presented above, we additionally investigate a *hybrid* form introducing a property known from simulated annealing into the genetic programming algorithm:

by appropriately tuning the measures of fitness, changes are applied more flexibly in the beginning, while evolution becomes more rigid over time. This hybrid approach has already been used for program synthesis by [KP08, KP09a, KP09b] as well as [HS16].

In our case, it basically consists in changing the candidate selection technique over time, by first establishing the safety properties only, and then the liveness properties.

Just as for the genetic programming technique, crossovers are optional for this hybrid approach as well.

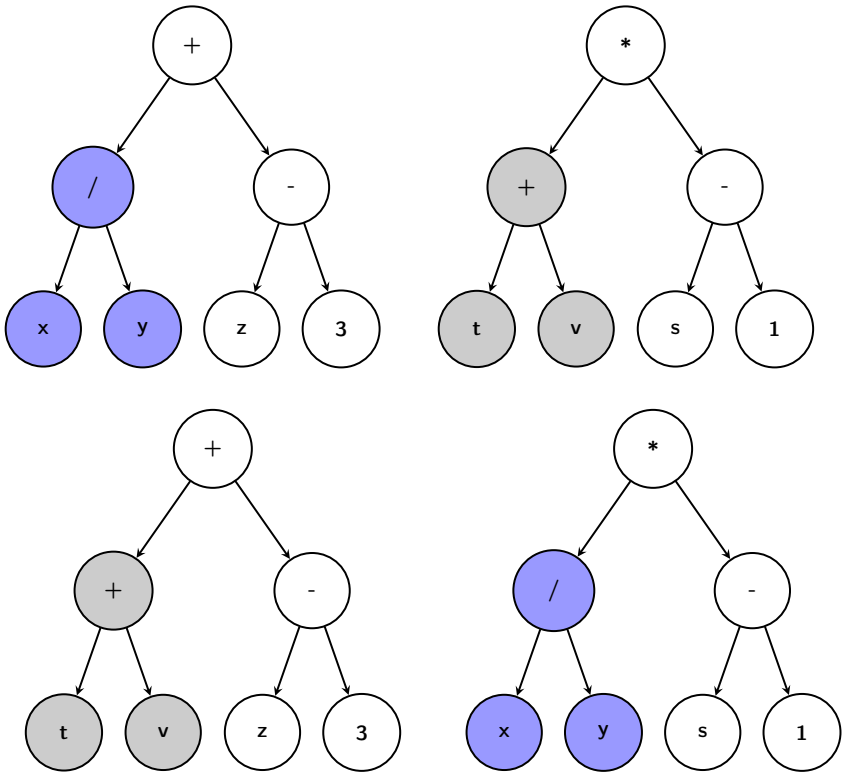


FIGURE 2.6: Crossover:two parents(above)and two offspring (below)

2.5 Model checking

Model checking [CGP99, BCM+90] is a technique used to determine whether a program satisfies a number of specifications. A model checker takes two inputs. The first of them, the specification, is a description of the temporal behavior a correct system shall display, given in a temporal logic. The second input, the model, is a description of the dynamics of the system that the user wants to evaluate. This might be a computer program, a communications protocol, a state machine, a circuit diagram, etc.

A model checker uses a symbolic representation of the model to decide efficiently if the model satisfies the specification. Standard temporal logics used in model checking are linear-time temporal logic (LTL) [Pnu77] and computation tree logic (CTL) [CE82, Eme90].

2.5.1 Computation Tree Logic (CTL)

In this work we are focus on CTL (Computation Tree Logic) [CE82, Eme90]. This consists a number of basic "atomic propositions" , that can be used with propositional logic connectives and a set of temporal connectives which act on propositions.

These temporal connectives consist of two components: a description of the scope over the future time paths (either A or E) and a description of when the proposition that

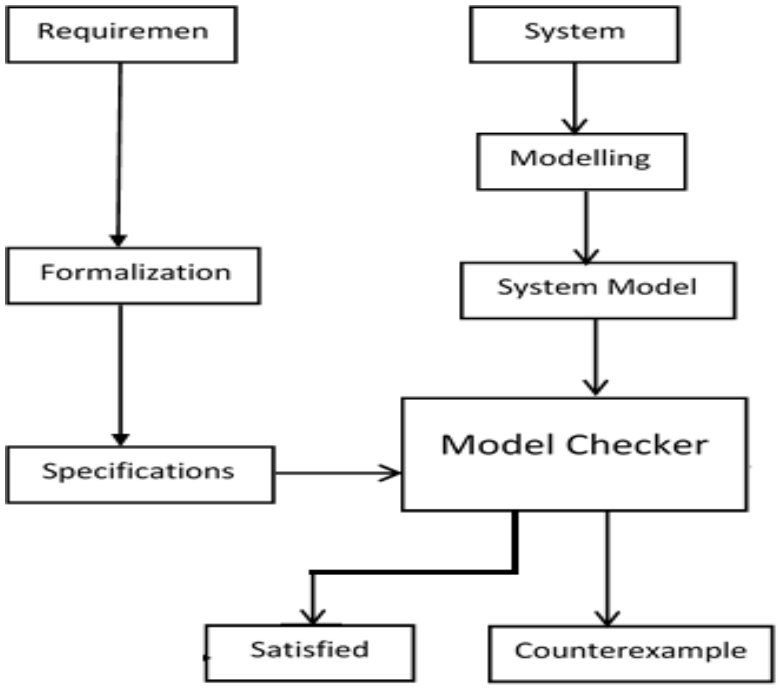


FIGURE 2.7: Model Checking

is the argument of the temporal operator holds within that scope (one of G,F,X or U). These have (in informal terms) the following meanings:

- A The proposition has to hold on All paths starting from the current point.
- E There Exists a path starting from current state on which the proposition will hold.
- G The proposition holds all states (Globally) along the path.
- F The proposition has to hold somewhere (in the Future) along the path.
- X The neXt state the proposition will be satisfied.
- U The proposition holds Until a second proposition holds.

(U is the only binary operator, the others are unary)

Given a finite set Π of atomic propositions, the syntax of a CTL formula is defined as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi,$$
$$\psi ::= X\phi \mid \phi U \phi \mid G\phi,$$

where $p \in \Pi$. For each CTL formula ϕ we denote the length of ϕ by $|\phi|$.

Let $T = (V, E)$ be an infinite directed tree, with all edges pointing away from the root. (In model checking, this is the unraveling of the model.) Let $l : V \to 2^{\Pi}$ be a labeling function. The semantics of CTL is defined as follows. For each $v \in V$ we have:

- $v \models p$ if, and only if, $p \in l(v)$.
- $v \models \neg\phi$ if, and only if, $v \not\models \phi$.
- $v \models \phi \vee \psi$ if, and only if, $v \models \phi$ or $v \models \psi$.
- $v \models A\psi$ if, and only if, for all paths π starting at v, we have $\pi \models \psi$.
- $v \models E\psi$ if, and only if, there exists a path π starting at v with $\pi \models \psi$.

Let $\pi = v_1, v_2, \ldots$ be an infinite path in T. We have:

- $\pi \models X\phi$ if, and only if, $v_2 \models \phi$.
- $\pi \models \phi U \phi'$ if, and only if, there exists an $i \in \mathbb{N}$ such that $v_i \models \phi'$ and, for all j in the range $1 \leq j < i$, we have $v_j \models \phi$.
- $\pi \models G\phi$ if, and only if, $v_i \models \phi$ for all $i \in \mathbb{N}$.

Note that the ϕ and ϕ' here are state formulas.

The pair (T, l), where T is a tree and l is a labeling function, is a *model* of ϕ if, and only if, $r \models \phi$, where $r \in V$ is the root of the tree. If (T, l) is a model of ϕ, then we write $T, l \models \phi$.

For the candidate programs in our work, the tree is the tree of all runs / interleaving of the programs under asynchronous composition, and the labels are the program states.

Chapter 3

Program Synthesis

This chapter is based on the results of [HS16]. We extend the quest for the best general search technique by studying the effect of the parameter settings for the individual search techniques: the influence of the selected temperature for simulated annealing and the crossover ratio for genetic programming. Note that we have already describe the general searching techniques and model checking on Chapter 1.

We start with our tool structure description in section 3.3,that simplify the main parts of the synthesis tool. Then on section 3.4 we discuss how the model checking used as a fitness for our searching methods. Section 3.5 shows how the programs represented in two forms for mutation purpose and as a pseudo code. Section 3.6 describes the case studies that we have solved using the synthesis tool. The investigation of the parameters variety shown on Section 3.7.1. The rest of the chapter consist of the evaluation section (3.8) and the discussion (section 3.9) of the tool results.

3.1 Abstract

Program synthesis can be viewed as an exploration of the search space of candidate programs in pursuit of an implementation that satisfies a given property. Classic synthesis techniques facilitate exhaustive search, while genetic programming has recently proven the potential of generic search techniques. But is genetic programming the right search technique for the synthesis problem? In this chapter we challenge this belief and argue in favour of simulated annealing, a different class of general search techniques. We show that, in hindsight, the success of genetic programming has drawn from what is arguably a hybrid between simulated annealing and genetic programming, and compare the fitness of classic genetic programming, the hybrid form, and pure simulated annealing. Our experimental evaluation suggests that pure simulated annealing offers better results for automated programming than techniques based on genetic programming.

3.2 Introduction

Model checking methods are used to verify the correctness of digital circuits and code against their formal specification. In case of design or programming errors, they provide counterexample evidence of erroneous behavior. Model checking techniques suffer from inherent high complexity. New model checking methods attempt to speed it up and reduce the memory requirement. Recently, the more ambitious task of converting the formal specification automatically into correct-by-design code has gained significant progress. In this chapter, automata-based techniques for model checking and automatic synthesis are described.

Concurrent systems are are very difficult to synthesize, where a specific task needs to be decomposed into many components, where each having limited visibility and control on the behaviour of the other components. This make the synthesis of concurrent systems is, in general, undecidable, so the classical software engineering methods in which testing is used for selection seems not suitable for this type of synthesis. The holy grail of such techniques would be synthesis: the automated construction of programs that are correct by construction. Such synthesis techniques have long been held to be impossible for reactive systems due to the complexity of synthesis, which ranges from EXPTIME for CTL synthesis [CE82, KV99] to undecidable for distributed systems [PR90, MT01, FS05, SF06].

This line of thought has come under attack on many fronts. On the theoretical side, bounded [FS13] and succinct [FPS15] synthesis techniques have levelled the playing field between the verification and synthesis of reactive systems by shifting the focus from the input complexity to the cost measured in the minimal explicit and symbolic solution, respectively. One could argue that this is the theoretical underpinning of successful approaches, including implementations of bounded synthesis [FJR09], and methods based on genetic programming [Joh07, KP08, KP09a, KP09b], also in [Ehl11] Unbeast, a tool for the synthesis of reactive systems from LTL specifications has presented

The success of genetic programming is also based on the observation that the neighborhood of good solutions are often ‘not bad’, and would often still display many sought after properties, such as satisfying a number of sub-specifications fully, and others partially. Such properties are translated to a high fitness of the candidate solution. Vice versa, the higher the fitness of a candidate, the more likely is it to find a full solution in its proximity. This observation is also at the heart of traditional engineering techniques: usually the elimination of a bug does not cause errors in other places. It is also the assumption used when applying program repair [JGB05, vEJ13] techniques. The successive development into correct programs is also distantly related to counter-example-guided inductive synthesis [Sol13] for inductive programs, where a genetic approach has also been discussed [DKL15].

Our work is at the same time inspired by the success of genetic programming and driven by the doubt if genetic programming is the right generic search technique to use. The success of genetic programming for synthesis is thoroughly documented by a series

of papers by Katz and Peled [KP08, KP09a, KP09b]. The doubts, on the other hand, are fueled by the general observation that genetic programming is often outperformed by simulated annealing [Dav87, LMST96, MS96].

On a conceptual level, the difference between simulated annealing and genetic programming techniques are rather minor. These difference are threefold. The first difference is in the number of candidates considered in each iteration. In genetic programming, these are many. In the Katz and Peled papers [KP08, KP09a, KP09b], for example, these are typically 150, 5 from the previous cycle and 145 mutated programmes—numbers we have copied for our own experiments with genetic programming. In simulated annealing, there is typically one new implementation in each iteration. The second difference is that genetic approaches may use crossovers, a proper mix of two candidate solutions, in addition to mutations, whereas simulated annealing only uses mutations[1]. The third difference is the way the selection takes place. The rules for selection is typically static for genetic programming, while the entropy falls over time in simulated annealing. It is important to note that crossovers are not always used in genetic programming, and we are not aware of any genetic programming approach that has tried to exploit crossovers for synthesis. Personal communication with the authors of [KP08, KP09a, KP09b] showed that they did not believe that crossover would be useful in the context of synthesis.

Simulated annealing has been reported [Dav87, LMST96, MS96] to outperform genetic programming when crossovers do not provide an advantage or are not used. Broadly speaking, this is because keeping only a single instance increases the update speed (where the factor is roughly the number of instances), whereas many instances reduce the search depth or increase the likelihood of success in a bounded search with a fixed number of iterations. Overall, the speed-up of the update tends to outweigh the increase in depth, or the reduction in the success rate, of a bounded search. This led us to the hypothesis that the same holds when these techniques are used in synthesis.

Finally, the paper series on genetic programming by Katz and Peled [KP08, KP09a, KP09b] has used a layered approach, where the evaluation of the search function differs over time, starting with establishing the safety then liveness properties. The effect of this difference is comparable to the effect of cooling when a stable level of quality is reached. We took this as another hint that simulated annealing is the more appropriate technique when implementing synthesis based on general search with model checking as a fitness measure. In this work we suggest to use simulated annealing for program synthesis and compare it to similar approaches based on genetic programming. We use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system.

We have implemented a synthesis tool, which uses multiple calls to the model checker NuSMV [CCG+02] to determine the fitness for a candidate program. The candidate programs exist in two forms. The main form is a simple imperative language. This form

[1]The changes are usually not referred to as mutations, but the rules of obtaining them are the same. We use the term mutations for simulated annealing, too, in order to ease the comparison between simulated annealing and genetic programming.

is subject to mutation, but it is translated to a secondary form, the modeling language of NuSMV, for evaluating its fitness. All choices of how exactly a program is represented and how exactly the fitness is evaluated are disputable.

Generic search techniques are, however, usually rather robust against changes in such details. While there has been further research on how to measure partial satisfaction [HO13], we believe that the best choice for us is to keep to the choices made for promoting genetic programming [KP08, KP09a, KP09b], as this is the only choice that is completely free of suspicion of being selected for being more suitable for simulated annealing than for genetic programming. A second motivation for this selection is that it results in very simple specifications and, therefore, in fast evaluations of the fitness.

Noting that synthesis entails on average hundreds of thousands to millions of calls to a model checker, only simple evaluations can be considered. We have implemented six different combinations of selection and update mechanism to test our hypothesis: besides simulated annealing, we have used genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover. The tests we have run confirmed that simulated annealing performs significantly better than genetic programming. As a side result, we found that the assumption of the authors of [KP08, KP09a, KP09b] that crossover does not accelerate genetic programming did not prove to be entirely correct, but the advantages we observed were minor.

3.3 The Approach in a Nutshell

Our tool consists of four main components: a modifier / seeder for programs (Program Generation), a compiler into a model checker format (Program Translation), a quantitative extension of a model checker, using NuSMV [CCG+02] as a back-end, and a selector that determines which program to keep (Generic Search Technique).

The structure of our synthesis tool is shown in Figure 3.1. In a nutshell, our synthesiser (cf. Figure 3.1) Via a user-interface, the user can select the problem parameters: (1) Search technique; (2) NuSMV specification (3) number, size, and type of variables; and (4) complex or simple conditional statements (if and while statement).

The parameters are sent to the Program Generator, which generates new programs according to the given parameters. Each program is first translated into NuSMV [CCG+02] and then analysed by NuSMV. The model checking results form the basis of a fitness function for the selected search technique. The specification is provided in form of a list of sub-specifications, which is then automatically extended to additional weaker specifications that are used to obtain a quantitative measure for partial satisfaction.

Broadly speaking, the extension takes partial satisfaction of a specification into account by giving different weights to different weaker versions of sub-specifications (cf. Section 3.4). The result can be manually modified, but the results reported in Section 3.8 refer to the automatically produced extension. The internal representation of a program is a tree. The seeder / modifier produces an initial seed. (Alternatively, one could

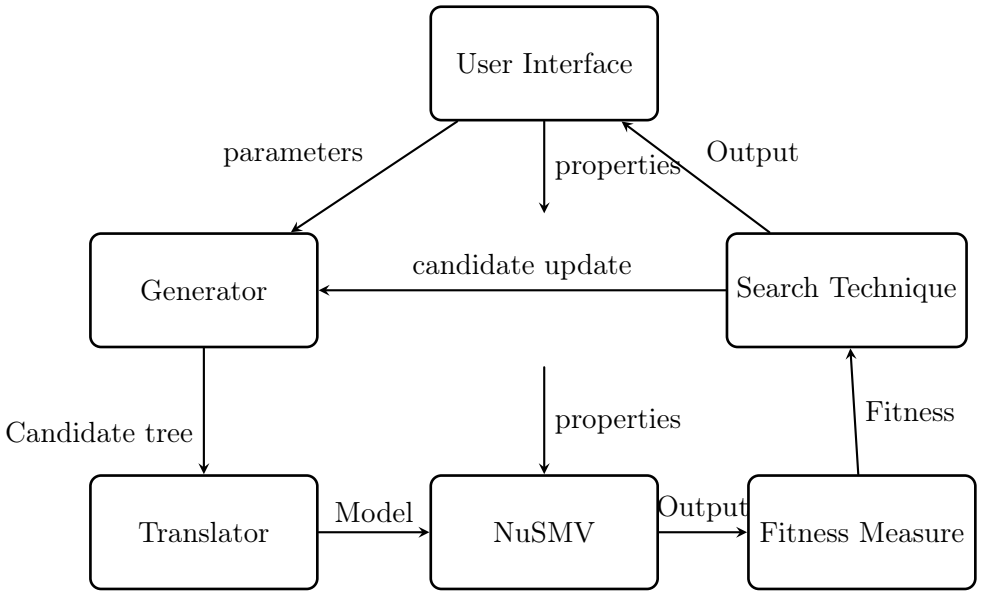


FIGURE 3.1: Synthes Tool

start with an initial program provided by the user.) The modifier / seeder also produces modifications of existing programs by changing sub-trees (cf. Section 3.5). The programs are then translated to the input language of a model checker (NuSMV in our case), which is then called several times to determine the level of satisfaction, which is the measure of the fitness (cf. Section 3.3) of a program.

Broadly speaking, the number of candidate programs kept depends on the search technique used. We have implemented both genetic programming approaches and simulated annealing in order to obtain a clean point of comparison. We use NuSMV [CCG+02] as a model checker. The translator therefore translates the abstract programs into the model language of NuSMV. The other parts of the tool are written in C++. Figure 3.1 gives an overview on the main components of our tool. When comparing simulated annealing to genetic programming, we merely replace the simulated annealing component by a similar component for the respective genetic programming variant and optionally add crossover to the available mutations.

The user provides specifications for the desired properties of a system in the form of a list of CTL specifications for the system dynamics that the program has to satisfy. The simulated annealing component then derives the intermediate specifications (full and partial compliance) that are used to determine the fitness of a candidate (cf. Section 3.3). If the candidate program satisfied all required properties, then the synthesiser returns it as a correct program.

Otherwise, it will compare the fitness of the current candidate with the (stored) fitness value of the program it is derived from by mutation. (This is the currently stored candidate.) If the fitness is lower, then the tool will update the stored candidate with the probability $e^{\Delta F/T(i)}$ defined by the loss $\Delta F = F_i - F_{i-1}$ in fitness and the current temperature $T(i)$ taken from the cooling schedule.

If the fitness is not lower, the tool will always replaces the stored candidate by the mutated one. When the end of the cooling schedule is reached, the tool aborts. The synthesis process is then re-started, either with a fresh cooling schedule (usually with a higher starting temperature or slower cooling) or with the same cooling schedule. We have implemented the latter.

3.4 Model Checking as a Fitness Function

We use model checking to determine the fitness of a candidate program in the same way as it has been used for genetic programming [KP08, KP09a, KP09b]. Based on the model checking results, we derive a quantitative measure for the fitness (as a level of partial correctness) of a program. This can be the share of properties that are satisfied so far, or mechanically produced simpler properties.

For example, if a property shall hold on all paths, it is better if it holds on some paths, and yet better if it holds almost surely. Our implementation considers the specification as a list of sub-specifications and assigns full marks for each sub-specification, which is satisfied by the candidate program. For cases where the sub-specification is not satisfied, we distinguish between different levels of partial satisfaction. We offer an automated translation of properties with up to two universal quantifiers that occur positively. To evaluate the candidates 100 points are assigned when the sub-specification is satisfied, 80 points if the specification is satisfied when replacing one universal path quantifier by an existential path quantifier, and 10 points are assigned if the specification is satisfied after replacing both universal path quantifiers by existential ones. Those evaluation numbers are given to weight the candidates and distinguish among three levels of them according to the satisfaction of the specifications. This means that it is possible to take another values (eg. 20, 60, and 100) the most important thing is keep the difference among them clear. (Existential quantifiers that occur negatively are treated accordingly.) Examples of this automated translation are shown in Section 3.6.

The output of the model checker is used to evaluate the fitness of the current candidate. The main part of the fitness is the average of the values for all sub-specifications in the rigid evaluation and the average of all liveness specifications in the flexible evaluation. Following [KP08], we apply a penalty for long programs by deducing the number of inner nodes of a program from this average when assigning the fitness of a candidate program. The resulting fitness value will be used by simulated annealing to compare the current candidate with the previous one when using rigid evaluation, and to make a decision whether the changes will be preserved or discarded. When using flexible evaluation, this only happens if the value for the safety specification is equal; falling resp. rising values for safety specifications always result in discarding resp. selecting the update when using flexible evaluation.

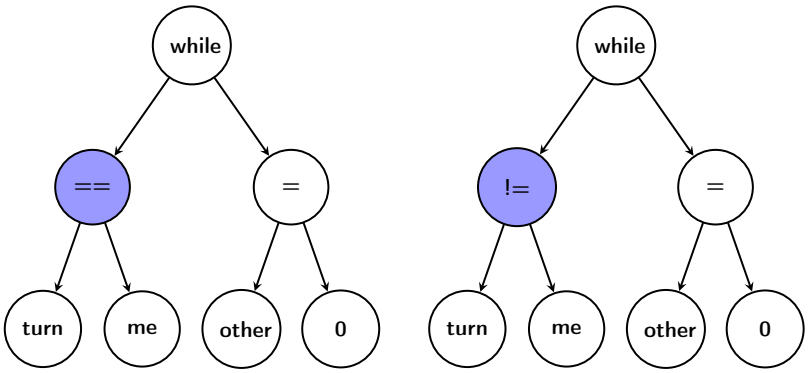


FIGURE 3.2: Program tree (left) with one node mutations (right)

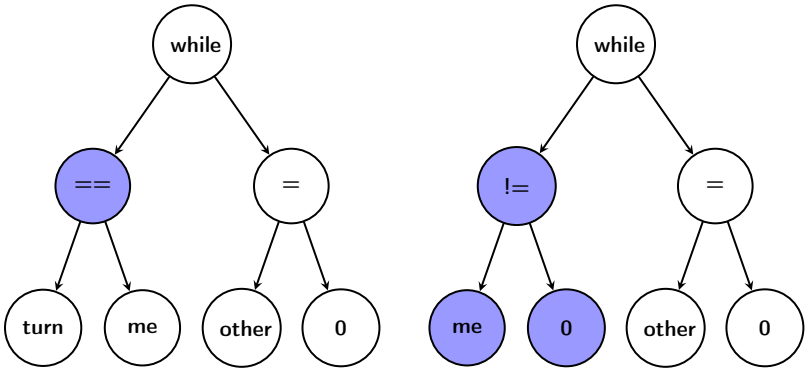


FIGURE 3.3: Program tree (left) with sub-tree mutations (right)

3.5 Programs as Trees

The main form of the programs is a tree, in which each *leaf node* represents a parameter or constant(eg. 0, 1, 2, *me*, *other*,*myturn*, and *otherturn*), while each *parent node* represents an operation like assignments, comparisons, or algorithm instruction like *if*, *while*, *empty while*, *and*, *or*. The candidate programs are built from the root down to the terminal nodes (cf. [Koz92, KP08]). Figure 3.2 shows the tree representation of the program

while (turn==me)
other=0

and two mutations of that program the first by replacing one node and another one by replacing sub-tree.

Mutations are changes in the program tree. Changes can be applied as follows:

1. Randomly select a node to be changed.
2. Apply one of the following changes:
   (a) Replace a boolean comparator by a different boolean comparator. E.g., the program from Figure 3.2 can result from the left program when '==' is replaced by '!='.
   (b) Replace a leaf by a different parameter or constant from a user defined set.

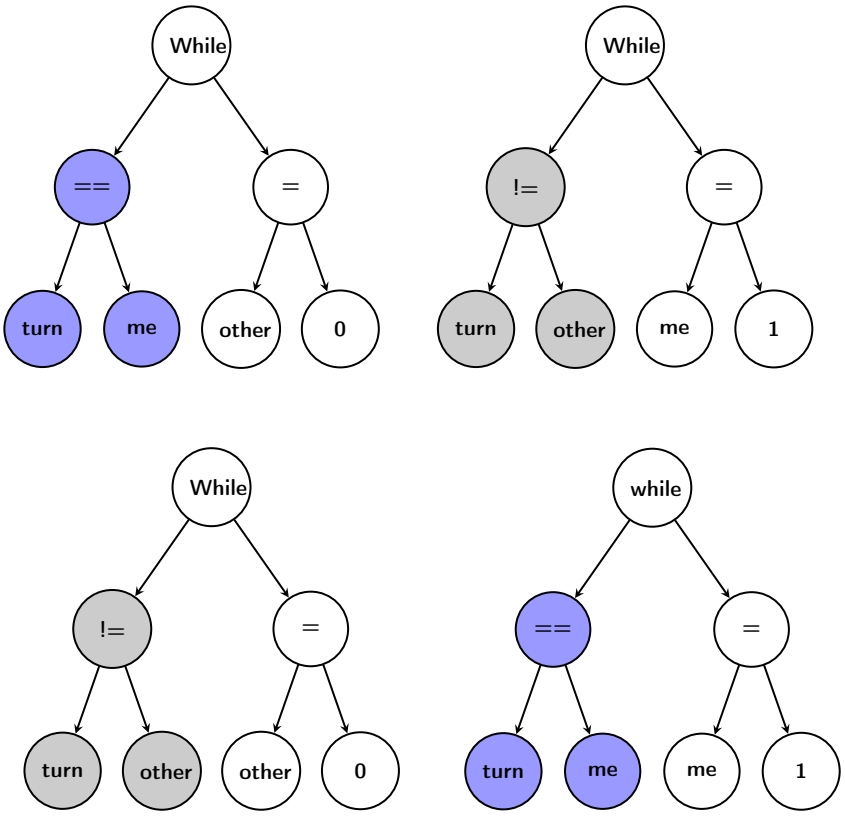


FIGURE 3.4: Crossover:two parents(above)and two offspring (below)

(c) Replace a sub-tree (which is no leaf) by a different sub-tree of size 3 with the same type. E.g., the program from Figure 3.3 can result from the left program when by replacing the left sub-tree.

(d) Add a new internal node, using the node that was there as one sub-tree and creating further offspring of minimal size (which is ≤ 3) to make the resulting tree well typed.

Crossovers between two programs P1 and P2 randomly select nodes N1 of P1 and N2 of P2, and swap the sub-trees rooted in N1 and N2. This way, they produce a proper mix of the two programs, see Figure3.4.

Besides standard commands—'while', 'if', assignments, boolean connectives and comparators—there are also variable names and constants. They have to be provided by the user. The user also needs to specify, which variables are local and which are global. She can provide an initial tree with nodes that the modifier is not allowed to alter. Examples of this are provided in Section 3.6.

To evaluate the fitness of the produced program, it is first translated into the language of the model checker NuSMV [CCG+02]. We have used the translation method suggested by Clarke, Grumberg, and Peled [CGP99].

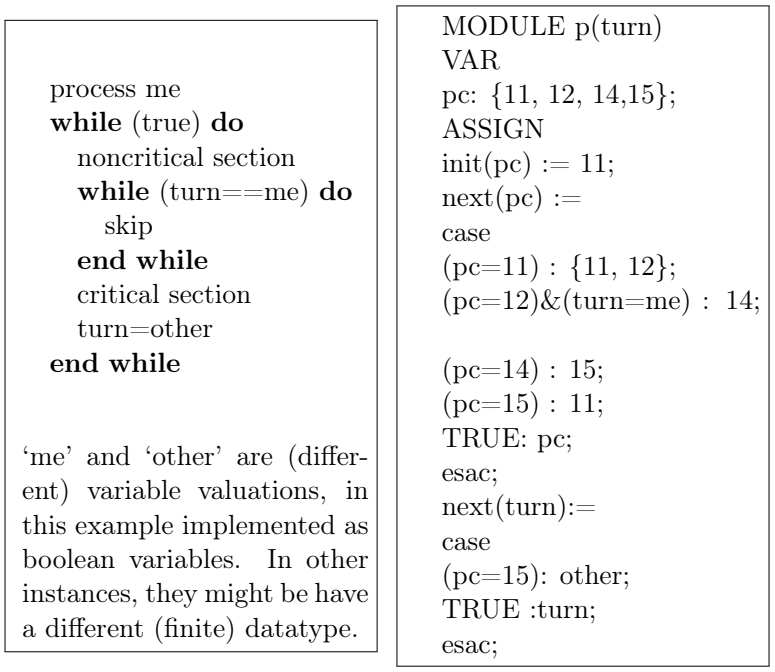


FIGURE 3.5: Translation example – source(left) and target (right)

In this translation, the program is converted into very simple statements (similar to assembly language). To simplify the translation, the program lines are first labeled, and this label is then uses as a pointer that represents the program counter *(PC)*. From this intermediate language, the NuSMV model is then built by creating *(case)* and *(next)* statements that use the *PC*. Figure 3 shows the translation of a mutual exclusion algorithm. At first, each line in the source algorithm labelled, then a variable pc (which is local for each MODULE) is added to represent the control state.

In the first step label each statement in the algorithm, the labels will used to build the model.

First step label the statements

10 while (true)
11 noncritical section
12 while (turn==0)
13 skip
14 critical section
15 turn=1

The next step is adding a pointer variable pc, and define all the variables as global variables.

Second add pc variable
VAR
turn:boolean;

p0:process(turn,myturn);
p1:process(turn,myturn);
....
MODULE P(turn,myturn)
VAR
pc: {11, 12, 14,15};
10 while (true)
11 noncritical section
12 while (turn==0)
13 skip
14 critical section
15 turn=1

Finally build the case and next state expression according the values of pc variable.

Thired Initialize all the variables properly, and start building their next-state expressions:

```
MODULE p(turn, myturn)
VAR
pc: {11, 12, 14,15};
ASSIGN
init(pc) := 11;
next(pc) :=
case
(pc=11) : {11, 12};
(pc=12)&(turn=myturn) : 14;
(pc=14) : 15;
(pc=15) : 11;
TRUE: pc;
esac;
next(turn):=
case
(pc=15): !myturn;
TRUE :turn;
esac;
```

And so on until build all the case statements by adding clauses according to the algorithm fragments, til create the program.

3.6 Case Studies

We have selected mutual exclusion [Dij65] and leader election [LP85, KP09b] as case studies, because these are the examples, for which genetic programming has been successfully attempted. In mutual exclusion, the programming language is set to use two and three shared bits, and for leader election 3 and 4 nodes ring can be used.

3.6.1 Mutual Exclusion

In mutual exclusion, no two processes are allowed to be in the *critical section* at the same time. In addition, there are liveness properties that essentially require non-starvation.

For the mutual exclusion example, we consider programs that progress through four sections, a 'non-critical section', an 'entry section', a 'critical section', and an 'exit section'. The 'non-critical section' and 'critical section' parts are not targets of the synthesis process. In this example, we start with a small program tree that includes the non-critical section and the critical section as privileged commands that cannot be changed by the modifier. Neither can any of their ancestors in the program tree. The entry and exit sections, on the other hand, are standard parts of the tree that can be changed.

The modifier is also provided with the vocabulary it can use. Besides the standard commands and the privileged commands for the critical and non-critical sections, these are the variables 'me' and 'other' that identify the two processes involved and, depending on the benchmark, two or three global / shared boolean variables see Figure 3.6.

The mutual exclusion example uses one safety specification: only one process can be in the critical sections at a time. This is represented by the CTL formula

$$!EF(P0 \text{ in critical section } \& \ P1 \text{ in critical section }).$$

When using this sub-specification for determining the fitness, we assign

100 points when the sub-specification is satisfied, and

80 points when $!AF(P0$ in critical section & $P1$ in critical section) holds.

In addition, there is a non-starvation property that, whenever a process enters its entry section, it will eventually enter the critical section. For one process ($P1$) this is

$$AG(P1 \text{ in entry section} \rightarrow AFP1 \text{ in critical section}).$$

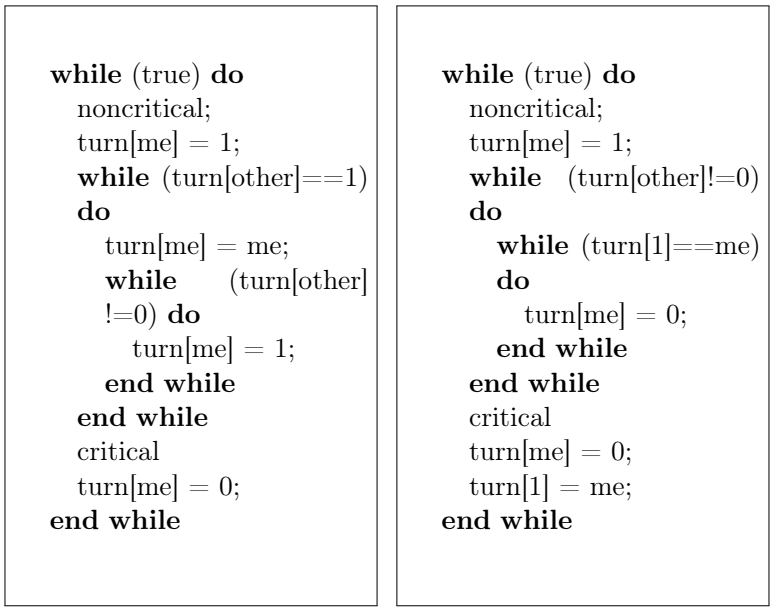
When using this sub-specification for determining the fitness, we assign

100 points when the sub-specification is satisfied,

80 points when $EG(P1$ in entry section $\rightarrow AFP1$ in critical section) holds,

80 points when $AG(P1$ in entry section $\rightarrow EFP1$ in critical section) holds, and

10 points when $EG(P1$ in entry section $\rightarrow EFP1$ in critical section) holds.



```
while (true) do
  noncritical;
  turn[me] = 1;
  while (turn[other]==1)
  do
    turn[me] = me;
    while (turn[other]
    !=0) do
      turn[me] = 1;
    end while
  end while
  critical
  turn[me] = 0;
end while
```

```
while (true) do
  noncritical;
  turn[me] = 1;
  while (turn[other]!=0)
  do
    while (turn[1]==me)
    do
      turn[me] = 0;
    end while
  end while
  critical
  turn[me] = 0;
  turn[1] = me;
end while
```

FIGURE 3.6: Synthesized Programs

3.6.2 Leader Election

As a second case study, we consider synthesising a solution for the leader election problem [LP85, KP09b]. For that purpose, we use clockwise unidirectional ring networks with two different sizes, three or four nodes, respectively.

For leader election, we do not consider any privileged commands. Again, the modifier needs to be provided with vocabulary. Besides the standard commands, this includes

- *id:* a specific integer value for each node in the ring, which have the values $1, \ldots, i$ for rings of size i.
- *myval,other,leaderID:* local variables; leaderID is initialized to 0.
- *Send (myval):* a command that refers to sending the value of 'myval' to the next node in the ring. (It is placed in a variable the next process can read using the following command.)
- *Receive (other):* a command that reads the last value sent by the previous node.

The specification for leader election requires the safety specification that there is never more than one leader, and the liveness requirement that a leader will eventually be elected. For both requirement, we assign

100 points when the sub-specification is satisfied on all paths, and

80 points when the sub-specification is satisfied on some path.

3.7 Synthesis Approach

When using the tool, the user starts with determining the search technique s/he would like to use from a list of three types of the techniques:

genetic programming, hybrid, and simulated annealing. Both the genetic programming and the hybrid method can be used with or without crossover. Hybrid [KP09a] is a method, where the fitness is viewed as a pair of 'safety-fitness' and 'liveness-fitness', where the latter is only used if the safety fitness is equal. For simulated annealing there are also two flavours to choose from: rigid (where the classic fitness function is used as described above) and *flexible*, which uses the two-step fitness approach from 'hybrid'. The tool allows the user to input parameters that control the dynamics of the synthesis process. These parameters depend on the selected search technique.

For genetic programming and the hybrid approach, the parameters include the population size, the number of selected candidates, and the number of iterations. For simulated annealing, the user chooses the initial temperature and the cooling schedule. The input specification should be a list of NuSMV specifications. The tool will produce weaker specifications for the purpose of producing a fitness measure.

Finally, the user defines the signature of the program by selecting input variables that are used to build the pseudo code of the program that the user want to synthesize. The model checker NuSMV is used for the individual model checking tasks. When the candidate program satisfies all required properties, the synthesiser returns it as a correct program. Otherwise, it will compare the fitness of the current candidate/s with the (stored) fitness value of the program/s it is derived from by mutation. (This is the currently stored candidate or population.) The selection of the candidate / population to be kept is determined by the selected generic search technique.

For simulated annealing, if the fitness is lower, then the tool will update the stored candidate with the probability defined by the loss in fitness and the current temperature taken from the cooling schedule. If the fitness is not lower, the tool will always replaces the stored candidate by the mutated one. When the end of the cooling schedule is reached, the tool aborts.

3.7.1 Parameters Setting

The tool allows the user to select the generic search technique and to influence the parameter used for fine-tuning the respective technique. The parameters of the search technique determine the likelihood of finding a correct program in each iteration as well as the expected running time for each iteration. The selected parameters therefore heavily influence the expected overall speed of the search.

The type of available parameters differs between genetic programming and simulated annealing. Figure 3.7 shows the graphical user interface of the tool, which the user can use to set the parameters or use the default setting.

First, the search technique is selected. The tool then provides the input variables to input their values, like population size and the number of the selected candidates for genetic programming and the hybrid technique, or the initial temperature and the cooling schedule for simulated annealing.

The user can also select the ratio of crossover vs mutation in genetic techniques. As a default setting, this ratio is 20% and the initial population size is 150 and 5 selected candidates.

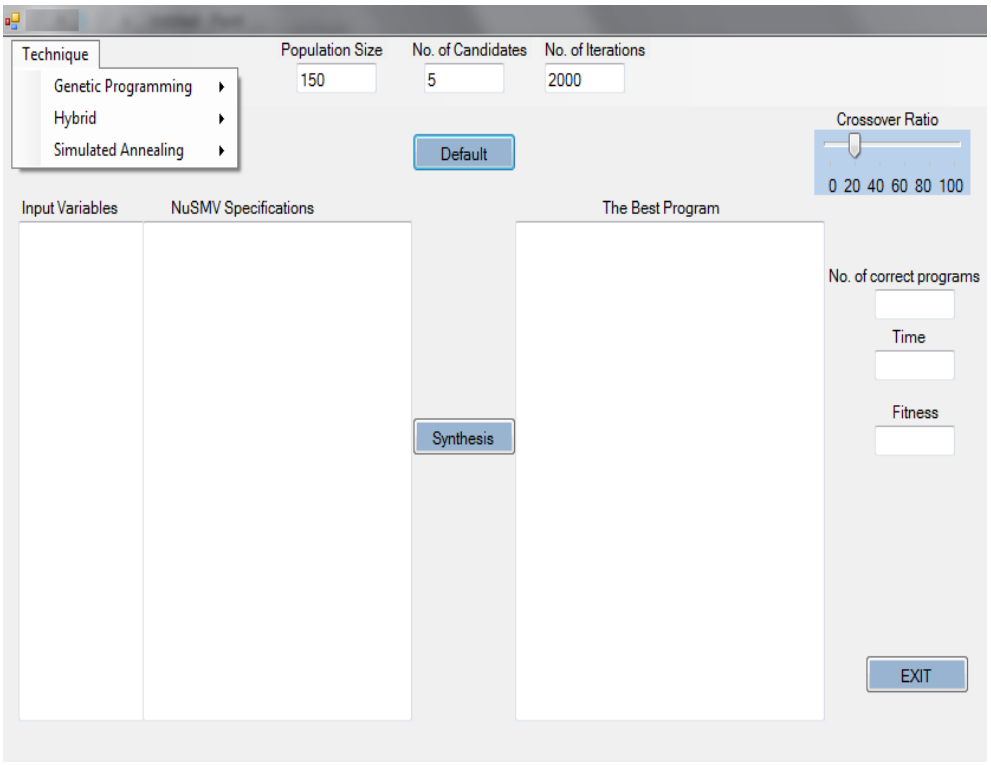


FIGURE 3.7: Graphical User Interface

The user should input her list of specifications and define the search space by selecting the input language (variables and their range). When the tool has found a correct program, it is provided to the user.

3.7.2 Temperature Range

Besides serving as a synthesis tool, the main intention of the tool is to allow for a comparison of different generic search techniques. In [HS16], we have used our technique to generate correct solutions for mutual exclusion and leader election and found that simulated annealing is much (1.5 to 2 orders of magnitude) faster than genetic programming.

In order to test if the hypothesis from [Con90] that simulated annealing does most of its work during the middle stages—while being in a good temperature range—holds for our application, we have developed the tool to allow for 'cooling schedules' that do not cool at all, but use a constant temperature. In order to be comparable to the default strategy, we use up to $25,001$ iterations in each attempt.

We have run 100 attempts to create a correct program using different constant temperatures, with the success rates, average running times per iteration, and inferred expected overall running times[2] shown in Tables 3.1 and 3.2.

	const temp	t	suc. %	expec. time
3 nodes	0.7	316	00	∞
	400	285	00	∞
	4000	196	11	1,781.81
	7000	97	14	692.85
	10000	**73**	**21**	**347.61**
	13000	78	**22**	354.54
	16000	83	20	415.00
	20000	87	19	457.89
	25000	94	17	494.73
	30000	108	15	720.00
	40000	117	15	780.00
	50000	129	13	992.30
	100000	193	12	1,608.33
4 nodes	0.7	521	00	∞
	400	493	00	∞
	4000	368	10	3,680.00
	7000	314	13	2,415.38
	10000	**138**	**18**	**766.66**
	13000	146	**19**	768.42
	16000	150	17	882.35
	20000	153	15	1,020.00
	25000	167	13	1,284.61
	30000	184	11	1,672.72
	40000	193	11	1,754.54
	50000	201	10	2,010.00
	100000	287	9	3,188.88

TABLE 3.1: Comparison for search temperature for Leader Election

The findings support the hypothesis that some temperatures are much better suited than others: low temperatures provide a very small chance of succeeding, and the chances also go down at the high temperature end. While the values for low temperatures are broadly what we had expected, the high end performed better than we had thought.

This might be because some small guidance is maintained even for infinite temperature, as a change that is decreasing the fitness is taken with an (almost) 50% chance in this case, while increases are always selected. The figures, however, are much worse than the figures for the good temperature range of $10,000$ to $16,000$.

The best results have been obtained at a temperature of $10,000$, with an expected time of 68 and 71 seconds for two and three shared bits, respectively. Notably, these results are better than the running time for the cooling schedule that uses a linear decline

[2] In all tables, execution times are in seconds; t is the mean execution time of single executions (succeeding or failing), and columns "expec. time" extrapolate t based on the success rate obtained in 100 single executions ("suc. %") The best values for each comparison printed in bold.

	const temp	t	suc. %	expec. time
2 shared bits	0.7	147	00	∞
	400	143	00	∞
	4000	129	3	4300
	7000	77	12	641.6
	10000	**15**	22	**68.18**
	13000	16	**23**	69.56
	16000	17	21	80.95
	20000	21	20	105
	25000	23	19	121.05
	30000	28	18	155.55
	40000	31	16	193.75
	50000	37	15	246.66
	100000	52	11	472.72
3 shared bits	0.7	155	00	∞
	400	148	00	∞
	4000	121	4	3025
	7000	81	11	252
	10000	**17**	**24**	**70.83**
	13000	18	**24**	75
	16000	19	22	86.36
	20000	23	22	104.54
	25000	25	21	191.04
	30000	30	19	157.89
	40000	34	17	200
	50000	41	16	256.25
	100000	58	13	446.15

TABLE 3.2: Comparison for search temperature for Mutual Exclusion

in the temperature. Starting at $20,000$ and terminating at 0, it has the same average temperature.

In the light of the results from the second experiment, it seems likely that the last third of the improvement cycles in this cooling schedule had little avail.

3.7.3 Crossover Ratio

Similarly, we have studied the effect of changing the share between crossover and mutation in genetic programming.We considered a range from a 0% to 60% share of crossovers. Country to our expectation, we did not observe any relevant effect of in- or decreasing the share of cross over for either classic or hybrid genetic programming. Interestingly, the running time per instance increased with the share of crossovers, which might point to a production of more complex programs. See Tables 3.3 and 3.4.

3.7.4 Initial Population Size Cost

The population size affect the algorithm in two sides, firstly large size could provide a better diversity than the small one, but besides that it could make the algorithm make

	ratio %	t	suc. %	expec. time
2 shared bits	0	**583**	7	8,328.57
	20	589	**9**	**6,544.44**
	40	602	**9**	6,688.88
	60	614	8	7,657.00
	80	613	8	7,662.50
	100	652	2	32,600.00
3 shared bits	0	**615**	7	8,785.71
	20	620	**9**	**6,888.88**
	40	637	**9**	7,077.77
	60	658	8	8,225.00
	80	669	4	16,725.00
	100	682	2	34,100.00
3 nodes	0	**1120**	3	37,333.33
	20	1123	**6**	**18,716.66**
	40	1137	5	22,740.00
	60	1149	5	22,980.00
	80	1154	3	38,466.66
	100	1167	2	58,350.00
4 nodes	0	**1,311**	3	43,700.00
	20	1,314	**5**	**26,280.00**
	40	1,325	4	33,125.00
	60	1,336	3	44,533.33
	80	1,345	3	44,833.33
	100	1,353	2	67,650.00

TABLE 3.3: Crossover ratio for GP (Program Synthesis)

more computation to find a good solution. In order to investigate how the population size effect on our synthesis approach and check the cost of the large size, we initiate the population size to different sizes see Tables 3.5 and 3.6.

We found that the the cost of the initial population size increase the cost of finding a good solution dramatically as shown on Figure 3.8.

3.8 Evaluation

We have implemented the simulated annealing and genetic programming algorithms as described, using NuSMV [CCG+02] as a solver when deriving the fitness of candidate programs. For simulated annealing, we have set the initial temperature to $20,000$. The cooling schedule decreases the temperature by 0.8 in each iteration.

The schedule ends after $25,000$ iterations, when the temperature hits 0. In a failed execution, this leads to determining the fitness of $25,001$ candidate programs.

As described in Section 3.2, we have taken the values suggested in [KP08] for genetic programming: $\lambda = 150$ candidate programs are considered in each step, $\mu = 5$ are kept, and we abort after $2,000$ iterations. In a failed execution, this leads to determining the fitness of $290,150$ candidate programs.

	ratio %	t	suc. %	expec. time
2 shared bits	0	**113**	31	364.51
	20	115	**33**	**348.48**
	40	123	**33**	372.72
	60	134	**33**	406.06
	80	142	21	676.19
	100	151	5	3,020.00
3 shared bits	0	**171**	17	1,005.88
	20	175	**19**	**921.05**
	40	187	**19**	984.21
	60	196	**19**	1,031.57
	80	207	11	1,881.81
	100	223	3	7,433.33
3 nodes	0	**418**	15	2,786.66
	20	421	**16**	**2,631.25**
	40	427	**16**	2,668.75
	60	453	13	3,484.61
	80	469	9	5,211.11
	100	487	4	12,175.00
4 nodes	0	**536**	11	4,872.72
	20	541	**14**	**3,864.28**
	40	557	13	4,284.61
	60	569	13	4,376.92
	80	581	9	6,455.55
	100	593	3	17,966.66

TABLE 3.4: Crossover ratio for Hybrid (Program Synthesis)

For the mutual exclusion benchmark, we distinguish between programs that use two and three shared bits, respectively. For the leader election benchmark we use ring networks with three and four nodes, respectively. The results are shown in Figures 3.9 to 3.11 and summarised in Table 3.7.

The experiments have been conducted using a machine with an Intel core i7 3.40 GHz CPU and 16GB RAM. Figure 3.9 shows the average time needed for synthesising a correct program. The two factors that determine the average running time are the success rate and the running time for a full execution, successful or not. These values are shown in Figure 3.11.

An individual execution of simulated annealing ends when a correct program is found or when the stopping temperature is reached after $25,000$ iterations. Similarly, the genetic programming approaches stop when they have found a solution or when the number or iterations has reached its maximum of $2,000$ iterations.

Note that, while simulated annealing incurs more iterations before reaching its termination criterion, it needs to perform only a fraction of the model checking tasks in each iteration. While the number of iterations is slightly more than an order of magnitude higher, the number of programs, for which the fitness needs to be calculated, is slightly more than an order of magnitude lower ($25,001$ vs. $290,150$).

GP w/o crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	583	7	8328.57
	7	583	7	8328.57
	9	584	7	8342.85
250	5	1024	12	8533.33
	7	1024	12	8533.33
	9	1024	12	8533.33
350	5	1435	15	9566.66
	7	1435	15	9566.66
	9	1435	15	9566.66
GP with crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	589	9	6544.44
	7	589	9	6544.44
	9	588	9	6533.33
250	5	1057	15	7046.66
	7	1057	15	7046.66
	9	1057	15	7046.66
350	5	1451	18	8061.11
	7	1451	18	8061.11
	9	1451	19	7636.84

TABLE 3.5: Population size vs cost for Program synthesis (2-shared bit mutual exclusion)

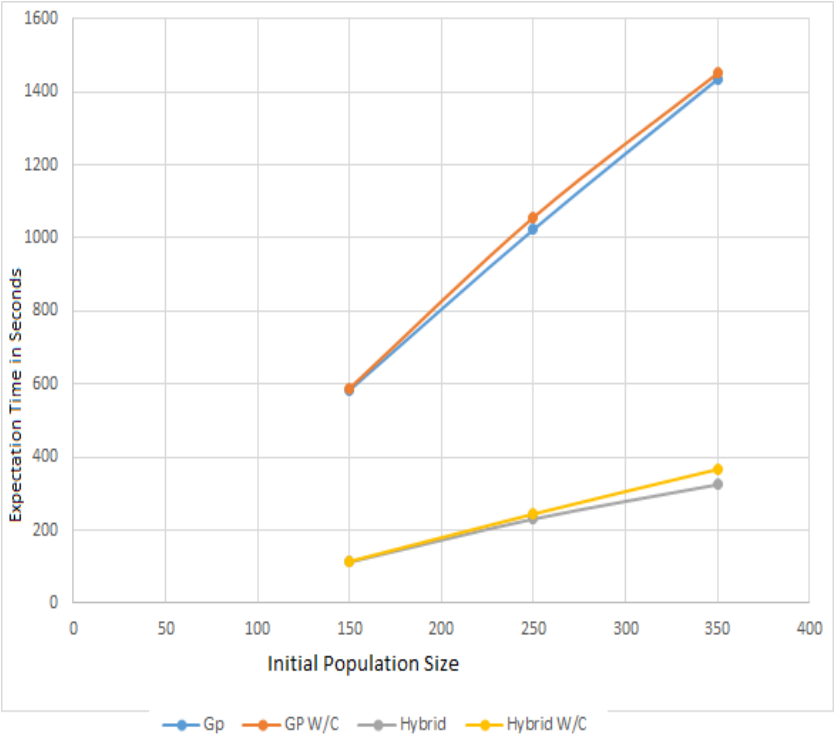


FIGURE 3.8: Initial Population Size vs Cost for SW Synthesis

Hybrid w/o crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	113	31	364.51
	7	113	31	364.51
	9	113	31	364.51
250	5	230	46	500.00
	7	230	46	500.00
	9	231	46	502.17
350	5	325	63	515.87
	7	325	63	515.87
	9	325	64	507.81
Hybrid with crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	115	33	348.48
	7	115	33	348.48
	9	114	33	345.45
250	5	245	49	500.00
	7	245	49	500.00
	9	245	49	500.00
350	5	367	67	547.76
	7	366	67	546.26
	9	367	67	547.76

TABLE 3.6: Population size vs cost for Program synthesis (2-shared bit mutual exclusion)

The success rates of around 20% may sound very low, but when we compare it with the genetic programming synthesis work [KP08, KP09a] we can find that this is the appropriate range for such techniques. Note that it is very simple to drive the success rate up: one can decrease the cooling speed for simulated annealing and increase the number of iterations for genetic programming, respectively.

However, this also increases the running time for individual full executions. A very high success rate is therefore not the goal when devising these algorithms, but a low expected overall running time. A 20% success rate is in a good region for achieving this goal.

Table 3.7 shows the average running time for single executions in seconds, the success rate in %, and the resulting overall running time.

Both simulated annealing and the hybrid approach significantly outperform the pure genetic programming approach. The low success rate for pure genetic programming suggests that the number of iterations might be too small. However, as the individual execution time is already ways above the average time simulated annealing needs for constructing a correct program, we did not increase the number of iterations.

The advantage in the individual execution time between the classic and the hybrid version of genetic programming is in the range that is to be expected, as the number of calls to the model checker is reduced.

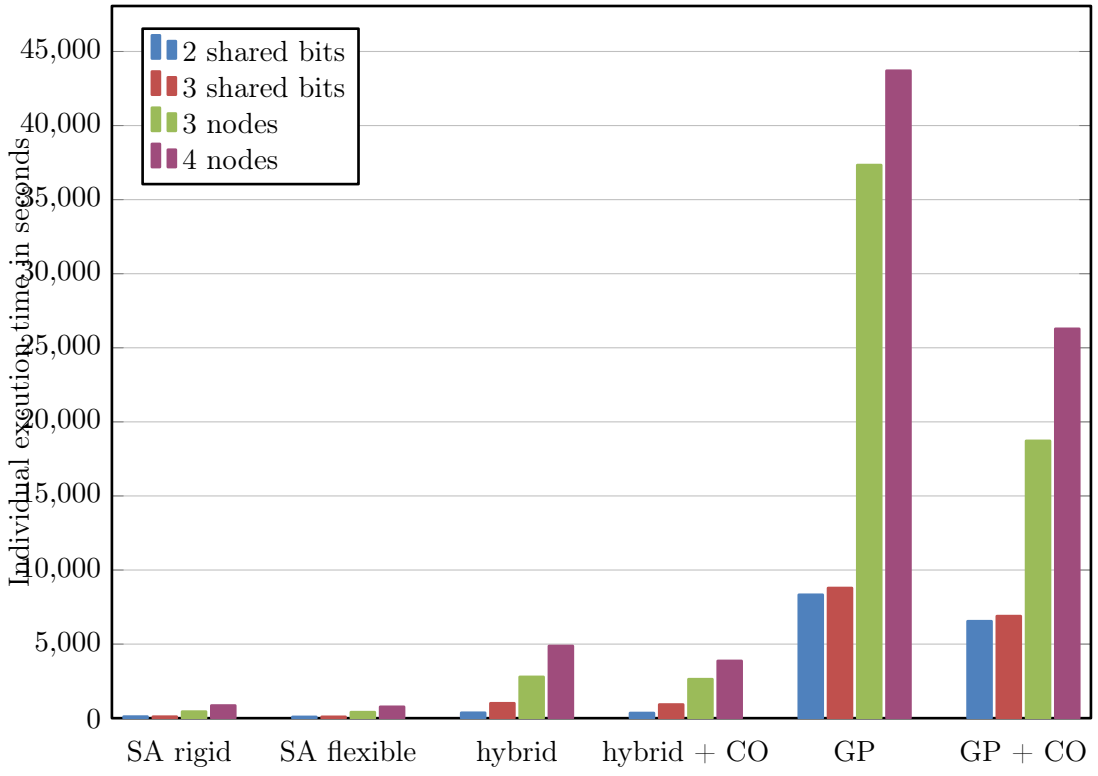


FIGURE 3.9: Average time required for synthesising a correct program

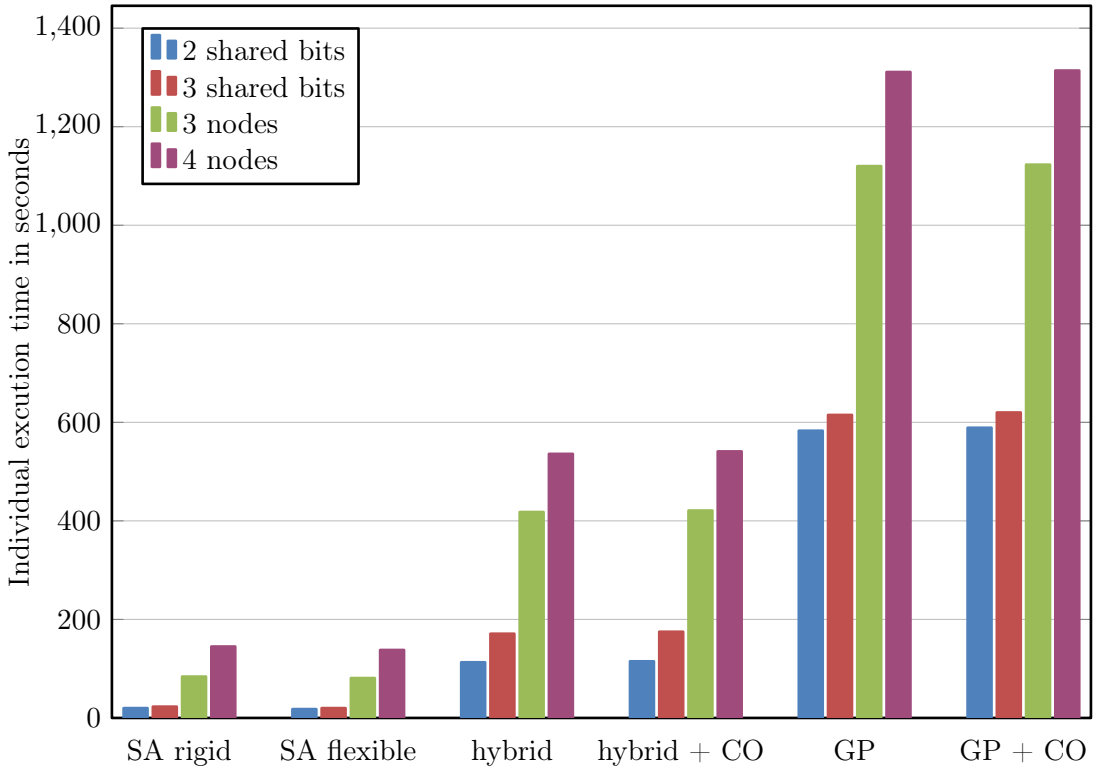


FIGURE 3.10: Average running time of an individual execution

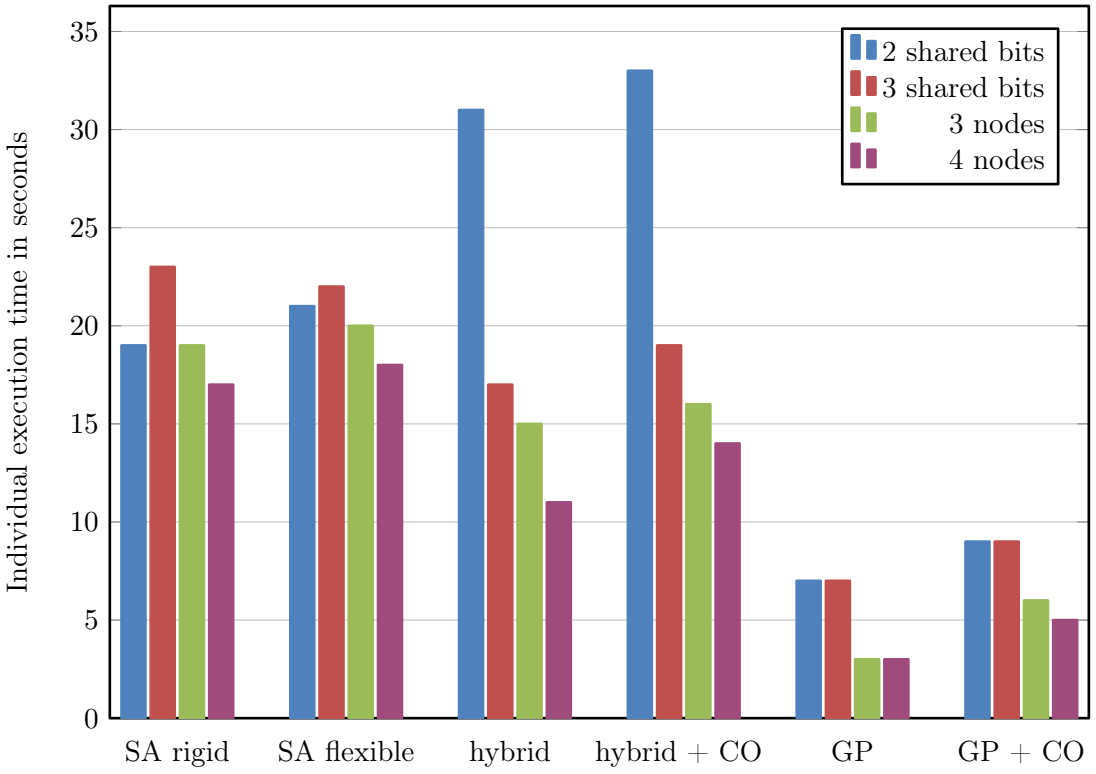


FIGURE 3.11: success rate of individual executions

It is interesting to note that simulated annealing, where the shift from rigid to flexible evaluation might be expected to have a similar effect, does not benefit to the same extent. It is also interesting to note that the execution time suggests that determining the fitness of programs produced by simulated annealing is slightly more expensive.

This was to be expected, as the average program length grows over time. The penalty for longer programs reduces this effect, but cannot entirely remove it. (This potential disadvantage is the reason why an occasional re-start provides better results than prolonging the search.)

The advantage in running of simulated annealing compared to the hybrid approach reach from factor 4 to factor 10, and the comparison to pure genetic programming reach from factor 35 to factor 76. It is interesting to note that both the pure and the hybrid approach to genetic programming benefit from crossovers, but while the benefit for the pure approach is significant, almost halving the average time for synthesising a program in one case, the benefit for the superior hybrid approach is small.

3.9 Discussion

We have implemented an automated programming technique based on simulated annealing and genetic programming, both in the pure form of [Joh07] and the arguably hybrid form of [KP08, KP09a]. The implementations from these papers were unavailable for comparison, but this is, in our view, a plus:

TABLE 3.7: Search Techniques Comparison

	Search Technique	t	suc. %	expec. time
2 shared bits	SA rigid	20	19	105.26
	SA flexible	**18**	21	**85.71**
	Hybrid w/o crossover	113	31	364.51
	Hybrid with crossover	115	**33**	348.48
	GP w/o crossover	583	7	8,328.57
	GP with crossover	589	9	6,544.44
3 shared bits	SA rigid	23	**23**	100
	SA flexible	**20**	22	**90.9**
	Hybrid w/o crossover	171	17	1,005.88
	Hybrid with crossover	175	19	921.05
	GP w/o crossover	615	7	8,785.71
	GP with crossover	620	9	6,888.88
3 nodes	SA rigid	84	19	442.1
	SA flexible	**81**	**20**	**405**
	Hybrid w/o crossover	418	15	2,786.66
	Hybrid with crossover	421	16	2,631.25
	GP w/o crossover	1120	3	37,333.33
	GP with crossover	1123	6	18,716.66
4 nodes	SA rigid	145	17	852.94
	SA flexible	**138**	**18**	**766.66**
	Hybrid w/o crossover	536	11	4,872.72
	Hybrid with crossover	541	14	3,864.28
	GP w/o crossover	1311	3	43,700.00
	GP with crossover	1314	5	26,280.00

The performance is naturally sensitive to the quality of the integration, the suitability of the model checker used, and hidden details, like how the seed is chosen or details of how the fitness is computed. The integrated comparison makes sure that all methods are on equal footage in these regards.

The results are very clear and in line with the expectation we had drawn from the literature [Dav87, LMST96, MS96]. When crossovers are not used, the main difference between the established genetic programming techniques and simulated annealing is the search strategy of using many and using a single instance, respectively. The data gathered confirms that an increase of the number of iterations can easily overcompensate the broader group of candidates kept in genetic programming.

In our experiments, we have used an increase that fell short of creating the same expected running time for a single full execution (with or without success), and yet outperformed even the hybrid approach w.r.t. the success rate on three of our four benchmarks.

We have also added variations of genetic programming that include crossover to validate the assumption that crossovers do not lead to an annihilation of the advantage, but it proved that the hybrid approach, and thus the stronger competitor, does not benefit much from using crossover.

The double advantage of shorter running time and higher success rate led to an improvement of 1.5 to 2 orders of magnitude compared to pure genetic programming (with and without crossover), and between half an order and one order of magnitude when compared to the hybrid approach (with or without crossover).

The results from in this chapter indicate that simulated annealing is the better general search technique to use. It will be interesting to see if that these factors are essentially constant, or if they depend heavily on the circumstances.

Together with the later extensions, the evaluation with the tool indicates that there are good temperature ranges. To developing the tool further will be an integration of these results into the cooling schedule. One way to do this could, for example, be to chose a temperature, where many improvements have clustered in the observed runs, or just to progress more slowly through areas where this has been the case during the observed runs in the past. Another possibility to use the observation that some temperatures provide better results than others could be to re-heat a bit after observing an improved fitness, or to staying in the temperature where this improvement was realised for a while.

Chapter 4

Discrete Controller Synthesis

This chapter is mainly based on the results from [HBS17]. In this chapter we formally define the symbolic model and DCS problems, and detail the particular kind of solutions we seek in Section 4.3.

We then turn to a description of the general search techniques that we investigate in Section 4.5.1, and further detail how we adapted them for solving our symbolic DCS problems in Section 4.6.

We detail our experiments of our simulated annealing algorithm in Sections 4.7. We eventually summarise and discuss our results in Section 4.8.4.

4.1 Abstract

Several general search techniques such as genetic programming and simulated annealing have recently been investigated for synthesising programs from specifications of desired objective behaviours.

In this context, these techniques explore the space of all candidate programs by performing local changes to candidates selected by means of a measure of their fitness wrt the desired objectives. Previous performance results advocated the use of simulated annealing over genetic programming for such problems.

In this chapter, we investigate the application of these techniques for the computation of deterministic strategies solving symbolic Discrete Controller Synthesis (DCS) problems, where a model of the system to control is given along with desired objective behaviours. We experimentally confirm that relative performance results are similar to program synthesis.

4.2 Introduction

Discrete Controller Synthesis (DCS) and *Program Synthesis* not only share a common noun, but also similar goals in that they are constructive methods for behaviours control.

The former typically operates on the model of a plant, and seeks the automated construction of a strategy so that the plant controlled accordingly fulfils a set of given objectives [RW89, AMP95].

Likewise, program synthesis operates by using some predefined rules, such as the grammar and semantics of the target programming language, and seeks the automated construction of a program whose execution fulfils given objectives.

Apart from their numerous applications to manufacturing systems [RW89, ZD12, KH91], DCS algorithms have also successfully been used to enforce deadlock avoidance in multi-threaded programs [WLK+09], enforce fault-tolerance [GR09], or for global resource management in embedded systems [ACMR03, BMM13]. A closely related algorithm was also applied by [RCK+09] for device driver synthesis.

Foundations of DCS and program synthesis are similar to principles of actually stem from the same theoretical foundations. They can be seen as complementary to model checking [CGP99, BCM+90], that determines whether a system satisfies a number of specifications.

In that respect, traditional DCS algorithms are highly inspired by model checking techniques.

Given a model of the plant, they first *exhaustively* compute an unsafe portion of the state-space to avoid for the desired objectives to be satisfied, and a strategy is then derived that avoids entering the unsafe region. A controller is built that alters the behaviour of the plant according to this strategy so that it is guaranteed to always behave as required.

Just as for model checking, *symbolic* approaches for solving DCS problems have been successfully investigated [AMP95, CKN98, MBLL00, BM14].

4.2.1 General Search Techniques

[CJ01, HJJ03, Joh07, KP08, KP09a, KP09b], as well as [HS16] in previous work, explored the use of *general search techniques* for program synthesis.

Instead of performing an exhaustive search, these techniques proceed by exploring the search space in pursuit of a program satisfying the objectives.

Among these techniques are *genetic programming* [Koz92] and *simulated annealing* [CJ01, HJJ03].

When applied to program synthesis, both search techniques work by successively mutating candidate programs that are deemed "good" by using some measure of their fitness wrt the desired objectives (eg using a model checker to measure the share of objectives satisfied by the candidate program, as done by [KP08, KP09a, KP09b] and [HS16]).

The genetic programming algorithm maintains a population of candidate solutions over a high number of iterations, generating new ones by mutating or mixing candidates randomly selected based on their fitness.

Simulated annealing on the other hand, produces one new candidate program per iteration, and does so by mutation only; the probability of survival of a candidate program depends on both its fitness and a temperature parameter that monotonically decreases from "hot" values favouring audacious mutations, to "cool" values preventing them. according to a "cooling schedule" inspired by cooling in the physical world.

By investigating and comparing these search techniques for program synthesis using their proof of concept implementation, [HS16] found that simulated annealing performs significantly better than genetic programming for synthesising programs.

4.2.2 Contributions

We first define a symbolic model and an associated class of DCS problems, for which deterministic strategies are sought. Next, we adapt the aforementioned search techniques to obtain algorithmic solutions that avoid computing the unsafe portion of the state-space.

Then, we confirm the hypotheses that:

(i) general search techniques are as applicable to solve our DCS problem as they are for synthesising programs; and (ii) one obtains similar relative performance results for our DCS problemexperimental results [HS16] for program synthesis, essentially that simulated annealing performs better than genetic programming.

To assess these hypotheses, we adapt the six different combinations of candidate selection and update mechanisms of our previous work [HS16], and execute them on a scalable example DCS problem.

Perform an experimental feasibility assessment.

From the performance results we obtain, we draw the conclusion that, even though for technical reasons our current experimental results do not compare favourably with existing symbolic DCS tools, simulated annealing, when combined with efficient model checking techniques, is worth further investigation to solve symbolic DCS problems.

4.3 Symbolic Model Checking & Controller Synthesis

4.3.1 Predicates

We denote by $V = \langle v_1, \ldots, v_n \rangle$ a vector of Boolean variables (*i.e.,* taking their values in the domain $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$; $\mathcal{D}_V = \mathbb{B}^n$ is the domain of V.

$V \cup W$ is the concatenation of two vectors of variables, defined iff they contain distinct sets of variables ($V \cap W = \varnothing$).

A *valuation* $v \in \mathcal{D}_V$ for each variable in V can be seen as the mapping $v \colon V \to \mathcal{D}_V$. We denote valuations accordingly: $v = \{v_1 \mapsto \mathsf{ff} ... v_n \mapsto \mathsf{tt}\}$.

Further, given an additional vector of variables W such that $V \cap W = \varnothing$, and corresponding valuations $v \in \mathcal{D}_V$ and $w \in \mathcal{D}_W$, the union of v and w is $(v, w) \in \mathcal{D}_{V \cup W}$.

$\mathcal{P}_V$ denotes the set of all *propositional predicates* to as *characteristic functions*) over variables in V, consisting of all formulae φ that can be generated according to the following grammar:

$$\varphi ::= \mathsf{ff} \mid \mathsf{tt} \mid v_i \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

where $v_i \in V$.

Let $P \in \mathcal{P}_V$ be such a predicate, and $v \in \mathcal{D}_V$ a valuation for variables in V.

$\mathcal{V}\colon V \to \mathcal{D}_V$)

Let $v \in \mathcal{D}_V$ be a valuation for each variable in V, and $P \in \mathcal{P}_V$ a predicate on V, we denote by $P(v)$ the

traditional evaluation of P after substituting every variable from V with its corresponding value in v. For each $\mathcal{V} \in \mathcal{D}_V$,

One has:

- $v \not\models \mathsf{ff}$ and $v \models \mathsf{tt}$;
- $v \models v_i$ iff $v(v_i)$, for $v_i \in V$;
- $v \models \neg\varphi$ iff $v \not\models \varphi$;
- $v \models \varphi \vee \psi$ iff $v \models \varphi$ or $v \models \psi$;
- $v \models \varphi \wedge \psi$ iff $v \models \varphi$ and $v \models \psi$.

($\varphi \Rightarrow \varphi'$ denotes the logical implication, equivalent to $\neg\varphi \vee \varphi'$.)

Lastly, $P(\mathcal{V}) \stackrel{\text{def}}{=} \mathsf{tt}$ if $\mathcal{V} \models P$, ff otherwise.

4.3.2 Symbolic Transition Systems

A Symbolic Transition System (STS) comprises a finite set of (internal and input) variables, and evolves at discrete points in time.

An update function indicates the new values for each internal variable according to the current values of the internal and input variables. Each transition is guarded on the system variables, and has an update function indicating the variable changes when a transition is fired.

Definition 4.1 (Symbolic Transition System)**.** A *symbolic transition system* is a tuple $S = \langle X, I, T, A, x^0 \rangle$ where:

- $X = \langle x_1, \ldots, x_n \rangle$ is a vector of state variables encoding the memory necessary for describing the system's behaviour;
- $I = \langle i_1, \ldots, i_m \rangle$ is a vector of input variables;
- $T = \langle T_1\colon \mathcal{P}_{X \cup I}, \ldots, T_n\colon \mathcal{P}_{X \cup I} \rangle$ is the update function of S, and encodes the evolution of all state variables based on n predicates involving variables in $X \cup I$;

- $A \in \mathcal{P}_{X \cup I}$ is a predicate encoding an assertion on the possible values of the inputs depending on the current state;
- $x^0 \in \mathcal{D}_X$ is the initial valuation for the state variables.
- $X_0 \in \mathcal{P}_X$ is a predicate encoding the set of initial states.

We give an illustrative STS in Example 4.1 below.

Remark 4.2 (Parallel Composition). Consider given the set of STSs $S_1, \ldots, S_N$, each involving distinct sets of variables.

The system obtained by concatenating all their state vectors altogether, as well as all their input vectors and update functions, combined with the conjunction of their respective assertions A_i, and initial states x_i^0, is an STS behaving as their *synchronous product* $S_1 \| \ldots \| S_N$: *i.e.,*

$$\|_{i \in \{1,\ldots,N\}} S_i \stackrel{\text{def}}{=} \left\langle \bigcup X_i, \bigcup I_i, \bigcup T_i, \bigwedge A_i, (x_1^0, \ldots, x_N^0) \right\rangle .$$

Semantics To each STS, one can make correspond a Finite State Machine (FSM):

Definition 4.3 (Finite State Machine corresponding to an STS)**.** Given an STS $S = \langle X, I, T, A, x^0 \rangle$, we make correspond an *FSM* $[S] = \langle \mathcal{X}, \mathcal{I}, \mathcal{T}, \mathcal{A}, x^0 \rangle$ where:

- $\mathcal{X} = \mathcal{D}_X$ is the state space of $[S]$;
- $\mathcal{I} = \mathcal{D}_I$ is the input alphabet of $[S]$;
- $\mathcal{T} \colon \mathcal{X} \times \mathcal{I} \to \mathcal{X} = \lambda(x, \iota).\, (T_j(x, \iota))_{j \in \{1,\ldots,n\}}$;
- $\mathcal{A} \subseteq \mathcal{X} \times \mathcal{I} = \{(x, \iota) \in \mathcal{X} \times \mathcal{I} \,|\, (x, \iota) \models A\}$;
- x^0 is the initial state of $[S]$.

The behaviour of an FSM $[S]$ is as follows:

Assuming that $[S]$ is in a state $x \in \mathcal{X}$. Then, upon the reception of an input $\iota \in \mathcal{I}$ such that $(x, \iota) \in \mathcal{A}$ (*i.e.,* ι is an admissible valuation for all variables of I in state x), $[S]$ evolves to the state $x' = \mathcal{T}(x, \iota)$.

Let $(x^0, \iota^0) \cdot (x^1, \iota^1) \cdot (x^2, \iota^2) \cdots$ be an infinite sequence of states and inputs of $[S]$ starting from a given state $x^0 \in \mathcal{X}$, that can be constructed according to the preceding rule ($\forall j \in \mathbb{N}$, $x^{j+1} = \mathcal{T}(x^j, \iota^j)$ and $(x^j, \iota^j) \in \mathcal{A}$). *Suff* $[S]\left(x^0\right)$ denotes the set of all such sequences, and *XSuff* $[S]\left(x^0\right)$ is the sequences of states that are obtained from *Suff* $[S]\left(x^0\right)$ by removing the input component of each tuple of the sequences.

Further, *Suff* $[S](x)$ is the set of all suffixes of execution traces reaching a state $x \in \mathcal{X}$; *i.e.,* $x \cdot y^0 \cdot y^1 \cdots \in$ *Suff* $[S](x)$ iff $\exists x^0 \cdot x^1 \cdot x \cdot y^0 \cdot y^1 \cdots \in XTrace[S]$. every sequence $\pi = x^0 \cdot x^1 \cdots \in XTrace[S]$ such that $\exists i \in \mathbb{N}$, $(x = x^i)$ and $x^0 \cdots x^i \cdot x^{i+1} \cdots \in) \wedge (x = x^i)$.

All *execution traces* of $[S]$ start in the initial state x^0, and $XTrace[S] \stackrel{\text{def}}{=}$ *XSuff* $[S]\left(x^0\right)$ denotes all such sequences of states.

Further, the set of all *execution paths* of $[S]$ is the set of all suffixes of any execution trace of $[S]$: $XPath[S] \stackrel{\text{def}}{=} \{\pi_s \,|\, \exists \pi_p, \pi_p \cdot \pi_s \in XTrace[S]\}$.

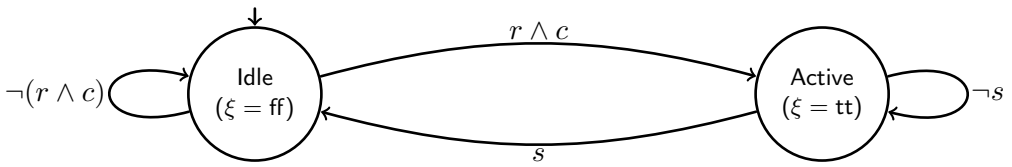


FIGURE 4.1: STS S_{Task} (Example 4.1) as a guarded automaton.

Example 4.1 (Task STS)**.** *Let us now give a small illustrative example STS modelling a two-state task. We build upon this example throughout this chapter.*

*We model the behaviour of the system under consideration, called "*Task*", using the STS, where* $X = \langle \xi \rangle$, $I = \langle r, s, c \rangle$, $T = \langle (\neg\xi \wedge r \wedge c) \vee (\xi \wedge \neg s) \rangle$, $A = \mathsf{tt}$, *and* $x^0 = \{\xi \mapsto \mathsf{ff}\}$. *The example represents a monitor in which* r *means run,* s *means stop, and* c *means control. An automaton representation of* Task *is given in Figure 4.1, where the* Idle *(resp.* Active*) location represents states where* $\xi = \mathsf{ff}$ *(resp.* $\xi = \mathsf{tt}$*).*

With successive inputs $\iota^0 = \{r \mapsto \mathsf{tt}, s \mapsto \mathsf{ff}, c \mapsto \mathsf{ff}\}$, *and* $\iota^1 = \{r \mapsto \mathsf{tt}, s \mapsto \mathsf{ff}, c \mapsto \mathsf{tt}\}$, *one obtains a prefix of execution traces* $\{\xi \mapsto \mathsf{ff}\} \cdot \{\xi \mapsto \mathsf{ff}\} \cdot \{\xi \mapsto \mathsf{tt}\} \cdots$ *belonging to* $XTrace[S_{\mathsf{Task}}]$.

4.3.3 Model Checking STSs

Model checking [CGP99, BCM+90] is a technique used to determine whether a system satisfies a number of specifications. A model-checker takes two inputs.

The first one of them, the *specification*, is a description of the temporal behaviour a correct system shall display, given in a temporal logic.

The second input, the *model*, is a description of the dynamics of the system that the user wants to assess, be it a computer program, a communications protocol, a state machine, a circuit diagram, etc.

Model-checkers usually use symbolic representations of the model to decide efficiently if it satisfies the specification.

Typical symbolic representations for predicates involve Binary Decision Diagrams (BDDs) [Bry86], because they yield good time and memory performance in practice.

BDDs also enjoy an often useful canonicity property, as all functionally equivalent predicates lead to a unique diagram.

Standard temporal logics used for model checking are *Linear-time Temporal Logic* (LTL) [Pnu77] and *Computation Tree Logic* (CTL) [CGP99].

As we focus on the latter, we now define CTL in terms of STSs.

CTL *w.r.t.* STSs Consider given an STS $S = \langle X, I, T, A, x^0 \rangle$, and its corresponding FSM $[S]$ with $\mathcal{X} = \mathcal{D}_X$. The syntax of a CTL formula ϕ relating to S is defined as

$$\phi ::= \varphi \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi$$
$$\psi ::= X\phi \mid \phi U \phi \mid G\phi$$

where $\varphi \in \mathcal{P}_X$.

For each CTL formula ϕ, we denote the length of ϕ by $|\phi|$.

For each $x \in \mathcal{X}$, we have (state formulae):

- $x \models \varphi$, for $\varphi \in \mathcal{P}_X$ (*cf.* Section 4.3.1);
- $x \models \neg\phi$ iff $x \not\models \phi$;
- $x \models \phi \vee \phi'$ iff $x \models \phi$ or $x \models \phi'$;
- $x \models A\psi$ iff $\forall \pi \in \mathit{XSuff}[S](x), \pi \models \psi$;
- $x \models E\psi$ iff $\exists \pi \in \mathit{XSuff}[S](x), \pi \models \psi$.

Let $\pi = x^0 \cdot x^1 \cdot x^2 \cdots \in \mathit{XPath}[S]$ be an (infinite) execution path of S. We have (trace formulae):

- $\pi \models X\phi$ iff $x^1 \models \phi$;
- $\pi \models \phi U \phi'$ iff $\exists i \in \mathbb{N}$, $x^i \models \phi'$ and $\forall j < i, x^j \models \phi$;
- $\pi \models G\phi$ iff $\forall i \in \mathbb{N}, x^i \models \phi$.

Note that ϕ and ϕ' here are state formulae. (The shortcut $F\phi$ denotes the "finally" construct, equivalent to $\mathtt{tt}\, U\phi$.)

$[S]$ is a *model* of ϕ iff $x^0 \models \phi$; if $[S]$ is a model of ϕ, then we write $[S] \models \phi$.

4.4 Symbolic Discrete Controller Synthesis

The theoretical framework for Discrete Controller Synthesis (DCS) algorithms was first introduced in [RW89] in a language-theoretic setting. The general goal of DCS algorithms is, given a system to be controlled S and a *control objective* ϕ, to obtain a *controller* that alters the behaviour of S so that it fulfils ϕ.

In terms of the STSs as defined above, DCS algorithms involve partitioning the input space (*i.e.,* the vector of input variables I) into *non-controllable* U and *controllable* inputs C. In practice, the former set typically corresponds to measures performed on the controlled system (*aka* plant), whereas the latter provides means for the controller to influence the behaviours of the model (and thereby on the controlled system itself).

A control objective can typically be expressed in the form of a CTL formula. A *desired invariant* for S, for instance, can be expressed as a CTL property of the form $AG\varphi$, for some $\varphi \in \mathcal{P}_X$.

4.4.1 Principles of Traditional DCS Algorithms

The traditional approach for solving DCS problems is inspired by classical model checking algorithms: as follows:

(i) a portion of the state-space $\mathcal{F}_\phi \subseteq \mathcal{X}$ that must be avoided for the desired control objective ϕ to hold whatever the valuations of the non-controllable inputs is first computed; then,

(ii) a strategy $\sigma_\phi \subseteq \mathcal{X} \times \mathcal{I}$ is derived that avoids entering $\mathcal{F}_\phi$;

(iii) the resulting controller operates according to σ_ϕ.

The synthesis fails if initial states belong to the forbidden area, *i.e.,* $\mathcal{F}_\phi \cap \mathcal{X}_0 \neq \varnothing$.

The synthesis fails if, starting from the initial state, there does not exist a strategy that avoids $\mathcal{F}_\phi$; in other words, it fails if the initial state belongs to the forbidden area of the state-space, *i.e.,* $x^0 \in \mathcal{F}_\phi$.

4.4.2 Symbolic DCS

Symbolic DCS algorithms targeting various models were investigated, by [AMP95, CKN98] for instance.

Regarding models close to STSs, symbolic DCS algorithms and tools were developed by [MS00, MBLL00], and later extended by [BM14] to deal with logico-numerical (infinite-state) systems, involving variables defined on numerical domains.

These algorithms operate on STSs (possibly extended with variables defined on numerical domains), with predicates represented using BDDs. They model checking approaches: they are based on a fixpoint computation of a symbolic representation of $\mathcal{F}_\phi$, the portions of the state-space $\mathcal{F}_\phi \subseteq \mathcal{X}$ that must be avoided for the desired control objective to hold, and the strategy σ_ϕ takes the form of a predicate K_ϕ restricting the admissible values of the controllable input variables *w.r.t.* the values of the state and non-controllable input ones; *i.e.,* $K_\phi \in \mathcal{P}_{X \cup I}$.

Then, given valuations for all state and non-controllable input variables, values for all controllable inputs are chosen so that K_ϕ is satisfied.

The STS resulting from these algorithms is then One requirement regarding STSs is that K must be at least as restrictive as A, *i.e.,* $\forall (x,i) \in \mathcal{D}_X \times \mathcal{D}_I, (x,i) \models K \Rightarrow (x,i) \models A$.

Example 4.2 (Controlling S_{Task})**.** *Building up on the STS S_{Task} introduced in Example 4.1, we consider that the input variable c is actually a controllable input variable: it is a lever given to the controller to be synthesised, to prevent the modelled task from entering the* Active *state if this behaviour may lead to a violation of desired control objectives.*

The resulting STS we use for DCS is then $S'_{\mathsf{Task}} = \langle X, U \uplus C, T, A, x^0 \rangle$, with $U = \langle r, s \rangle$ and $C = \langle c \rangle$.

Consider for instance the control objective expressed as the CTL formula $\phi = AG((\neg s) \vee X \neg \xi)$ expressing that S'_{Task} should not enter the Active *state while non-controllable input s holds.*

This objective can be manually attained by following the strategy represented by the predicate $K_\phi = (\neg\xi \wedge s \wedge r \Rightarrow \neg c)$.

Given now first non-controllable inputs $u^0 = \{r \mapsto \mathsf{tt}, s \mapsto \mathsf{tt}\}$*, according to the strategy* K_ϕ *we must choose* $c^0 = \{c \mapsto \mathsf{ff}\}$ *as values for the controllable inputs (as* c^0 *is the only valuation for* C *s.t* $(x^0, u^0, c^0) \models K_\phi$*, and* $c^{0\prime} = \{c \mapsto \mathsf{tt}\}$ *would lead to a violation of* ϕ*). The controlled system* S'_{Task} *can then evolve into state* $x^1 = \{\xi \mapsto \mathsf{ff}\}$ *(staying in* Idle*).*

With further inputs $u^1 = \{r \mapsto \mathsf{tt}, s \mapsto \mathsf{ff}\}$*, we can either choose* $c^1 = \{c \mapsto \mathsf{ff}\}$ *or* $c^{1\prime} = \{c \mapsto \mathsf{tt}\}$*, that both fulfil the strategy* K_ϕ *and respectively lead to state* $x^2 = \{\xi \mapsto \mathsf{ff}\}$ *and* $x^{2\prime} = \{\xi \mapsto \mathsf{tt}\}$.

4.4.3 Controlled Execution of STSs

As exemplified above, σ_ϕ might be non-deterministic, and its symbolic representation K_ϕ describes a relation: σ_ϕ is a subset of $\mathcal{X} \times \mathcal{I}$.

Given a state $x \in \mathcal{X}$ and a valuation for all non-controllable inputs $u \in \mathcal{D}_U$, the set of all valuations $\{c \in \mathcal{D}_C \mid (x, u, c) \models K_\phi\}$ might not be a singleton.

Therefore, traditional DCS algorithms require further processing steps to eventually produce a deterministic, *executable* controlled system.

Two approaches exist to this end:

(i) using an on-line solver to randomly pick values $c \in \mathcal{D}_C$ for the variables in C, given values for non-controllable inputs $u \in \mathcal{D}_U$ and state $x \in \mathcal{X}$, *s.t* $(x, u, c) \models K_\phi$ holds (as we did manually in Example 4.2); or (ii) translating the predicate K_ϕ into a function assigning values for each controllable variable based on values for state and non-controllable input variables.

In the remainder of this chapter, to obtain deterministic, easily implementable controlled STSs, we seek algorithms that give results similar to those obtained after applying the translation of on solving symbolic DCS problems combined with the translation of option (ii).

4.4.4 Obtaining a Deterministic Controlled STS

Option (ii) above amounts to refining the non-deterministic strategy σ_ϕ into a *deterministic strategy* σ'_ϕ.

Note that this translation may have an impact on the kind of control objectives that can effectively be enforced using the traditional DCS algorithms (that operate by computing $\mathcal{F}_\phi$), as this determinisation procedure implicitly entails "removing" transitions from σ_ϕ. Notably, the resulting predicates highly depend on both the order of variable eliminations, and their default values.

A triangulation technique similar to the one described by [HRL08] may be used to translate K_ϕ into a set of assignments.

This translation operates by using successive variable substitutions and partial evaluations of K_ϕ. It requires ordering (prioritising) the controllable input variables, and assigning "default" values for them, or introducing additional non-controllable input "phantom" variables (i.e. preferable constant inputs).

Essentially, the symbolic representation for σ'_ϕ obtained by triangulation for an STS $S = \langle X, U \uplus C, T, A, x^0 \rangle$, with $C = \langle c_1, \ldots, c_k \rangle$, is a vector Γ_ϕ of k predicates giving values for each controllable variable of the system based on state and non-controllable inputs only, *i.e.,*

$$\Gamma_\phi = \langle \gamma_1 \colon \mathcal{P}_{X \cup U}, \ldots, \gamma_k \colon \mathcal{P}_{X \cup U} \rangle. \tag{4.1}$$

Every occurrence of a controllable variable in the update function T can then be substituted with its corresponding assignments in Γ_ϕ, thereby providing a *Deterministic Controlled STS* (DCSTS), denoted $S/_{\Gamma_\phi}$, satisfying the desired objective (*i.e.,* $[S/_{\Gamma_\phi}] \models \phi$).

Example 4.3 (Deterministic Controller for S'_{Task})**.** *Consider again the controller obtained in Example 4.2. A triangulation of K_ϕ with default value* tt *for c gives $\Gamma_\phi^{c?\mathsf{tt}} = \langle (\xi \vee \neg r \vee \neg s) \rangle$. The alternative choice for the default value for c leads to $\Gamma_\phi^{c?\mathsf{ff}} = \langle (\mathsf{ff}) \rangle$, always assigning the value* ff *to c (and incidentally prevents S'_{Task} from ever reaching the* Active *state).*

The resulting DCSTS in the first case is $S'_{\mathsf{Task}}/_{\Gamma_\phi^{c?\mathsf{tt}}} = \langle X, U, T[C/\Gamma_\phi^{c?\mathsf{tt}}], A, x^0 \rangle$ where

$$\begin{aligned} T[C/\Gamma_\phi^{c?\mathsf{tt}}] &= \langle (\neg\xi \wedge r \wedge (\xi \vee \neg r \vee \neg s)) \vee (\xi \wedge \neg s) \rangle \\ &= \langle (\neg\xi \wedge r \wedge \neg s) \vee (\xi \wedge \neg s) \rangle. \end{aligned}$$

Note that the triangulation has an impact on the kind of control objectives that can effectively be enforced using traditional DCS algorithms (that operate by computing $\mathcal{F}_\phi$), as this determinisation procedure implicitly entails "removing" transitions from σ_ϕ.

This translation is also computationally expensive when performed on BDDs, and may incur a non-negligible increase in the number of nodes involved to represent the resulting functions. The controller in this work is memory free.

4.5 Contribution *w.r.t.* Symbolic DCS

Our contribution is an original algorithm for the construction of correct DCSTSs solving symbolic DCS problems when multiple control objectives are desired: given an STS S and a set ω of desired control objectives specified using CTL formulae over variables of S, its goal is to construct a *deterministic strategy* σ'_ω so that S controlled according to σ'_ω fulfils every objective belonging to ω.

Accordingly, the resulting deterministic strategy shall take the form of a vector of predicates over state and non-controllable input variables of S (as in Equation (4.1)),

and the goal of our algorithm is to find a "good" solution Γ_ω so that $\forall \phi \in \omega, [S/\Gamma_\omega] \models \phi$. To this end, we rely on:

(i) *general search techniques* to amongst explore the set of all potential deterministic strategies; (ii) well-established *model checking* techniques for assessing the fitness of such potential solutions.

4.5.1 General Search Techniques

The contribution we investigate two general search techniques, namely *simulated annealing* and *genetic programming*, and derive a *hybrid* one. We present these techniques, and turn to their application in combination with model checking to find deterministic strategies in the following Section.

In the search of deterministic controllers solving symbolic DCS problems for the construction of correct DCSTSs solving symbolic DCS problems. Let us now elaborate on the each of them search techniques we shall experiment with.

Simulated annealing [CJ01, HJJ03, HS16] is a general local search technique that is able to escape from local optima.

Algorithm 2: Simulated Annealing for Discrete Controller Synthesis

$i := 0$
randomly generate a first candidate c
repeat
  $i := i + 1$
  derive a new candidate c' of c
  $\Delta F := F(c') - F(c)$
  if $\Delta F >= 0$ **then**
    $c := c'$
  else
    derive random number $p \in [0, 1]$
    if $p < e^{\frac{\Delta F}{\theta_i}}$ **then**
      $c := c'$
    end if
  end if
until the goal is reached or $i = i_{\mathsf{max}}$

The algorithm, described in Algorithm 2, is easy to implement and has good convergence properties.

When applied to an optimisation problem, the fitness function (objective) generates values for the quality of the solution constructed in each iteration.

The fitness of this newly selected solution is then compared with the fitness of the solution from the previous round.

Improved solutions are always accepted, while some of the other solutions are accepted in the hope of escaping local optima in search of global optima.

The probability of accepting solutions with reduced fitness depends on a temperature parameter, which is typically falling monotonically with each iteration of the algorithm.

Simulated annealing starts with an initial, *randomly generated*, candidate solution. In each iteration, a neighbouring candidate x' is generated by mutating the previous candidate x.

Let, for the i^{th} iteration, $F(x)$ be the fitness of the "old" solution and $F(x')$ the fitness of its mutation newly constructed.

If the fitness is not decreased ($F(x') \geqslant F(x)$), then the mutated solution x' is kept.

If the fitness is decreased ($F(x') < F(x)$), then the probability p that this mutated solution is kept is

$$p = e^{\frac{F(x')-F(x)}{\theta_i}}$$

where θ_i is the temperature parameter for the i^{th} step. The chance of changing to a mutation with smaller fitness is therefore reduced with an increasing gap in the fitness, but also with a falling temperature parameter.

The temperature parameter is positive and usually non-increasing ($0 < \theta_i \leqslant \theta_{i-1}$).

The development of the sequence θ_i is referred to as the *cooling schedule* and inspired by cooling in the physical world [HJJ03].

The effect of *cooling* on the simulation of annealing is that the probability of following an unfavorable move is reduced. In practice, the temperature is often decreased in stages.

The cooling schedule is given as a set of parameters that determines how the temperature is reduced in each iteration (*i.e.,* the initial temperature, the stopping criterion, the temperature decrements between successive stages, and the number of transitions for each temperature value).

For our investigations, we have used a simple cooling schedule, where the temperature is dropped by a constant in each iteration.

4.5.2 Random Generation of Candidates

Genetic programming has already been used for program synthesis in [Joh07, KP08, KP09a, KP09b], and [HS16]. In genetic programming, a population of λ candidate solutions (in our case, deterministic strategies) is first randomly generated.

Then at each iteration, a small share of the population consisting of μ candidates, with $\mu \ll \lambda$, is selected based on its fitness; usually, a random function that makes it more likely for fitter candidate solutions to be selected for spawning the next generation is applied.

The selected candidates are then mated using some *crossover* operation to make up a population of λ, and mutations are applied to a high share of the resulting candidates (*e.g.,* on all duplicates). Mutations of selected candidates are used to obtain λ candidates at the end of each iteration. Crossovers are optional.

To randomly initialise the population of strategies, we need to generate vectors of as many trees as controllable variables in C. We use the "grow" method suggested by [Koz92] in order to build each tree; the method starts from the root, and potential children nodes are generated until the maximum depth of the tree is reached.

Algorithm 3: $\mathsf{grow}_{maxdepth}\,(depth)$

```
if depth < maxdepth then
  node ← random ({∨, ∧, ¬, tt, ff} ∪ X ∪ U)
  for each children child required for node do
    node.child ← grow_maxdepth (depth + 1)
  end for
else
  node ← random ({tt, ff} ∪ X ∪ U)
end if
return node
```

$\mathsf{grow}_{maxdepth}\,(depth)$ is shown as a recursive function in Algorithm 3, generating trees of maximum depth maxdepth. It takes the depth of the current node to be generated as argumentpredicate represented by the tree can be expressed. Initially depth=0.

If depth is less than the maximum tree depth, a node is chosen randomly from the set of terminals and binary operators. Then, depending on whether the node to generate is a terminal (leaf) or internal node, as many recursive calls as needed are performed to create the required number of children for the node.

If depth equals the maximum tree depth, then a node is chosen from the set of terminals.

k) calls to grow1X∪U shall then be used to generate one candidate deterministic strategy Γ.

Apart from simulated annealing and pure genetic programming with and without crossovers as presented above, we additionally investigate a *hybrid* form introducing a property known from simulated annealing into the genetic programming algorithm: by appropriately tuning the measures of fitness, changes are applied more flexibly in the beginning, while evolution becomes more rigid over time.

This hybrid approach has already been used for program synthesis in [KP08, KP09a**?**] as well as [HS16].

In our case, it basically consists in changing the candidate selection technique over time, by first establishing the safety properties only, and then the liveness properties.

Just as for the genetic programming technique, crossovers are optional for this hybrid approach as well.

4.6 Principles of our DCS Algorithms

In this Section, we assume given an STS $S = \langle X, U, C, T, A, x^0 \rangle$ to be controlled, with $C = \langle c_1, \ldots, c_k \rangle$, and a set of desired objective CTL formulae $\omega = \{\phi_1, \ldots, \phi_w\}$.

4.6.1 Representing Deterministic Strategies

Recall that the candidate deterministic strategies are vectors of predicates involving state and non-controllable variables (*cf.* Section 4.5), and such candidates are subject to

mutations (and possibly crossovers) for genetic programming and simulated annealing algorithms to be applicable. Therefore, one needs to find a suitable representation for such vectors of predicates.

Usual symbolic representations for predicates involve Binary Decision Diagrams (BDDs) [Bry86], for they yield good time and memory performance in practice. BDDs also enjoy an often useful canonicity property, as every functionally equivalent predicates lead to a unique BDD.

Usual symbolic representations for predicates involve BDDs (*cf.* Section 4.3.3). Yet, implementing efficient random generation, mutations, and crossovers on such diagrams appears to be challenging, and more importantly, unnecessary *w.r.t.* the purpose of this chapter whose goal is only to perform a our goal of performing a preliminary feasibility assessment for the use of general search techniques to solve symbolic DCS problems. The canonicity of candidates is not required by the algorithms we investigate either.

Therefore, we have opted for the simpler solution of using trees built according to the grammar of predicates introduced in Section 4.3.1:

- each leaf is labelled with either a variable belonging to $X \cup U$, or a constant in $\{\mathsf{tt}, \mathsf{ff}\}$;
- each node with one children is labelled with $\neg$;
- each node with two children is labelled with a binary operator $\vee$ or $\wedge$.

In the end, we represent candidate deterministic strategies as fixed-sized vectors $\Gamma = \langle \gamma_i \rangle_{i \in \{1,\ldots,k\}}$ of k trees γ_i as defined above, one per controllable variable in C.

Predicate consist of two parts, first a predicate P describes a relation or property, and variables (x, y) can take arbitrary values from some domain,P still have two truth values for statements ($TandF$), When we assign values to x and y, then P has a truth value. This can be easily reprehend as a binary tree, in which mutations can be applied in a simple way the using BDDs.

4.6.2 Performing Mutations and Crossovers

Mutations are changes applied on each candidate deterministic strategy Γ. Recall that the latter are represented as vectors of predicates involving state and non-controllable variables (*cf.* Section 4.5). Such changes can be applied using a random walk on one tree of Γ as follows:

1. Randomly select a predicate γ to be changed in Γ;
2. Perform a random walk on γ from its root, randomly choosing to stop or visit one of its children nodes (picked with uniform probability);
3. Perform a random walk on γ from its root, randomly choosing to stop or visit one of its children nodes (picked with probabilities weighted by the number descendants);

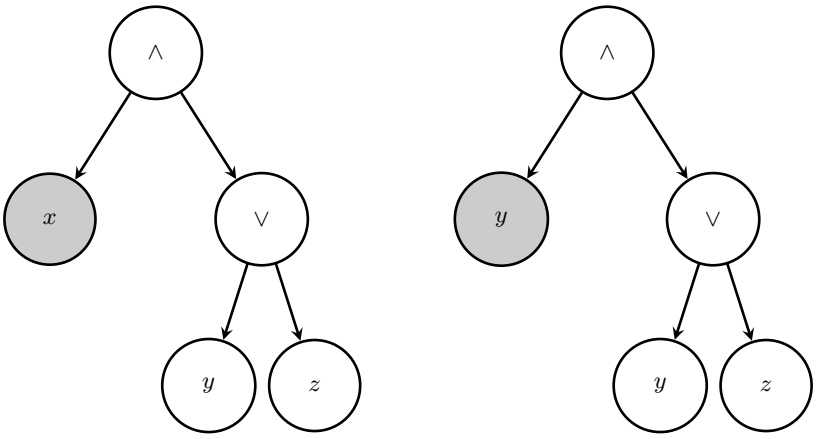


FIGURE 4.2: Candidate predicate (left) with one node mutation (right)

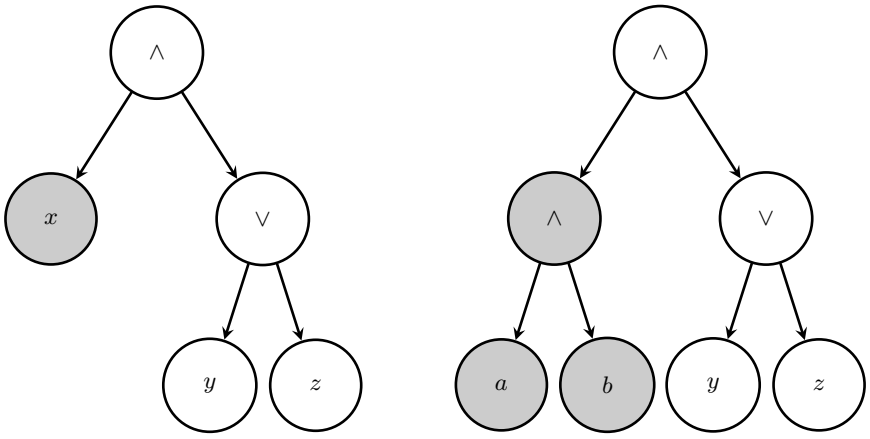


FIGURE 4.3: Candidate predicate (left) with sub-tree mutation (right)

4. Apply the one change applicable from the following:

   - When on a node n labelled with a binary operator $\vee$ or $\wedge$, replace it with a different binary operator or insert a negation node $\neg$ with child n;
   - When on a node labelled with a unary operator $\neg$, remove it;
   - When on a leaf l, insert a negation node $\neg$ with child l, or replace l with a randomly generated sub-tree built by using $\mathsf{grow}_{maxdepth}(1)$; the latter case is illustrated in Figure 4.3 and Figure 4.2.

The principle for performing the crossover between two candidates $\Gamma_1 = \langle \gamma_{1,i} \rangle_{i \in \{1,\ldots,k\}}$ and $\Gamma_2 = \langle \gamma_{2,i} \rangle_{i \in \{1,\ldots,k\}}$ consists in selecting and index $j \in \{1, \ldots, k\}$ (*i.e.,* a controllable variable), and swapping two randomly selected sub-trees t_1 from $\gamma_{1,j}$ and t_2 from $\gamma_{2,j}$.

As each predicate involved is defined on the same set of variables ($X \cup U$), a proper mix of the two candidates is always produced. We show in Figure 4.4 an example crossover between two trees.

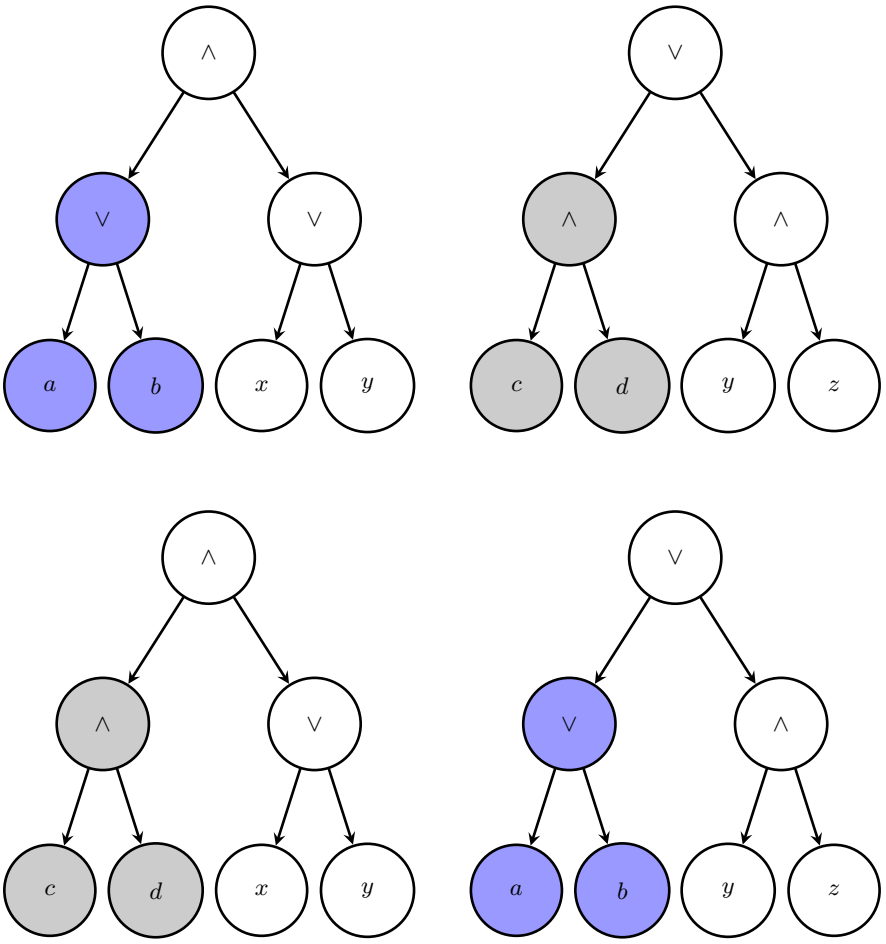


FIGURE 4.4: Crossover: two parents (above) and two offspring (below)

4.6.3 Model checking as a Fitness Function

We use model checking to determine the fitness of a candidate deterministic strategy in a way similar to that of [KP08, KP09a, KP09b] and [HS16] for program synthesis using genetic programming.

The model checking results are used to derive a quantitative measure for the fitness (as a level of partial correctness) of a deterministic strategy Γ *w.r.t.* the objectives ω.

To design a fitness measure for candidates, we make the hypothesis that the share of objectives that are satisfied so far by a candidate is a good indication of its pertinence.

We additionally observe that candidate solutions that satisfy weaker objectives—that can be mechanically derived from those belonging to ω—may be good candidates worth selecting for the generation of further potential solutions.For example, if a property shall hold on all paths (*i.e.,* CTL formula of the form $A\psi$) , it is better if it holds on some paths, and yet better if it holds almost surely.

Taking these observations into account, we first automatically translate the objectives with up to two universal path quantifiers occurring positively into weaker objectives:

- A first set of weaker objectives ω' is obtained by selecting every objective from ω featuring universal path quantifiers (*i.e.,* $A\psi$), and replacing

- Repeating this process once again on eligible objectives of ω' gives ω''.

(We shall give example partial objectives as built by the above procedure in Section 4.7.1 below.)

Then, given a candidate deterministic strategy Γ and a set of objectives ω, a model checking algorithm is used to check whether $[S/\Gamma] \models \phi$, for each objective ϕ in $\omega \cup \omega' \cup \omega''$

The fitness of Γ is computed based on the number of objectives of each set ω, ω' and ω'' that it satisfies: m points are assigned per objective of ω that is satisfied, m' points (with $m' < m$) per objective of ω', and m'' points (with $m'' < m'$) per objective of ω''.

We offer an automated translation of properties with up to two universal quantifiers that occur positively. 100 points are assigned when the sub-specification is satisfied, 80 points if the specification is satisfied when replacing one universal path quantifier by an existential path quantifier, and 10 points are assigned if the specification is satisfied after replacing both universal path quantifiers by existential ones. (Existential quantifiers that occur negatively are treated accordingly.) Examples of this automated translation are shown in Chapter 3. Eventually, the main part of the fitness is the average of the values for all considered objectives ω in the rigid evaluation, and the average of all liveness objectives ω_l in the flexible evaluation.

Following the works of [KP08] and [HS16], we also apply a penalty for "large" strategies by deducing the number of inner nodes of all trees from this average when assigning the fitness of a candidate strategy.

Let us denote $F_\omega(\Gamma)$ the fitness value obtained as described above for the candidate Γ and objectives ω.

4.6.4 Variants for Improved Search Techniques

To derive improved variants of the general search algorithms of Section 4.5.1, we note that a subset ω_s of given target objectives ω are *safety* ones, while the others ω_l $(= \omega \backslash \omega_s)$ are considered *liveness* objectives[1].

At each iteration of the simulated annealing algorithm, the first decision to select a new candidate Γ' over an old one Γ (condition $\Delta F < 0$ in Algorithm 2) is taken according to one of two distinct policies, giving two versions of the simulated annealing algorithm:

rigid Γ' is selected at this stage whenever $F_\omega(\Gamma') > F_\omega(\Gamma)$;

flexible Γ' is selected at this stage whenever $F_\omega(\Gamma') > F_\omega(\Gamma)$ or $F_{\omega_s}(\Gamma') > F_{\omega_s}(\Gamma)$; Γ' is always discarded when $F_{\omega_s}(\Gamma') < F_{\omega_s}(\Gamma)$.

The resulting fitness value will be used by simulated annealing to compare the current candidate with the previous one when using *rigid* evaluation, and to make a decision whether the changes will be preserved or discarded.

[1] One can use a simple syntactical criterion for deciding that an objective surely belongs to ω_s; *e.g.,* some safety CTL formulae can be rewritten as $AG\varphi$ with $\varphi \in \mathcal{P}_X$.

When using *flexible* evaluation, this only happens if the value for the safety specification is equal; falling (resp. rising) values for safety specifications always result in discarding (resp. selecting) the update when using flexible evaluation.

Our implementation of the hybrid algorithm presented in Chapter 2 basically consists in tuning the evaluation of fitness to first favour safety objectives: in this algorithm, the fitness $F_\omega(\Gamma)$ of a candidate Γ is actually made of the pair $(F_{\omega_s}(\Gamma), F_{\omega_l}(\Gamma))$, and comparisons of such fitness measures are performed according to their lexical ordering.

This way, candidates with better values for the safety objectives ω_s are always given preference, while the fitness computed using liveness objectives ω_l only are merely tie-breakers for equal values of $F_{\omega_s}(\Gamma)$.

selection technique over time. It first evaluates the fitness of each candidate Γ using safety objectives only $(F_{\omega_s}(\Gamma))$. Then, over a second phase, it continues by taking all objectives ω into account to measure the fitness $(F_\omega(\Gamma))$.

properties, and then the liveness properties. Specifications with better values for the safety properties are always given preference, while liveness properties are—for equal values for the safety properties—used to determine the fitness. *i.e.,* they are merely tie-breakers.

This approach has been used in [HS16] as well as [KP08, KP09a, KP09b]. We refer to it as a *hybrid approach* as it introduces a property known from simulated annealing: in the beginning, the algorithm is applying changes more flexibly, while it becomes more rigid later.

crossover, and used both evaluation techniques for simulated annealing, where we refer to using the classic fitness function as a *rigid* evaluation, and to the hybrid approach as *flexible* evaluation.

In the end, we investigate and compare six algorithms involving various search techniques:

1. pure genetic programming without crossovers (GP); and
2. with crossovers (GP w. CO);
3. rigid simulated annealing (SA rigid);
4. flexible simulated annealing (SA flexible);
5. hybrid search without crossovers (Hybrid); and
6. with crossovers (Hybrid w. CO).

4.7 Experimental Feasibility Assessment

4.7.1 Problem Instances

In order to get a preliminary feasibility assessment of our approach, we have generated several problem instances based on the parallel composition of STSs as per Remark 4.2. Synthesis objectives were then derived as CTL formulae in a scalable manner.

Regarding the STSs, each problem N–Tasks was built using composition of N instances of the STS S'_{Task} described in Example 4.2 (modulo renaming of variables to

ensure disjointness); *i.e.,* the STSs involved were built as

$$S_{N\text{–}\mathsf{Tasks}} = \|_{i\in\{1,\ldots,N\}}\langle X_i, U_i \uplus C_i, T_i, A_i, x_i^0\rangle,$$

where $X_i = \langle \xi_i \rangle$, $U = \langle r_i, s_i \rangle$, $C_i = \langle c_i \rangle$, $T_i = \langle (\neg\xi_i \wedge r_i \wedge c_i) \vee (\xi_i \wedge s_i) \rangle$, $A_i = \mathsf{tt}$, and $x_i^0 = \{\xi_i \mapsto \mathsf{ff}\}$.

Objective CTL formulae for each problem N–Tasks consist of both safety $\omega_{s,N}$ and liveness $\omega_{l,N}$ objectives.

Regarding safety, mutual exclusion properties suit our need for scalable, CTL formulae that are relatively complex to represent: *i.e.,* no two tasks should be in their Active state (with $\xi_i = \mathsf{tt}$) at the same time.

For each problem instance N–Tasks, one obtains:

$$\omega_{s,N} = \left\{ AG\left(\bigwedge_{i\in\{1,\ldots,N-1\}} \left(\bigwedge_{j\in\{i+1,\ldots,N\}} \neg\left(\xi_i \wedge \xi_j\right) \right) \right) \right\},$$

with notable special case $\omega_{s,1} = \varnothing$. As a concrete illustration, one obtains $\omega_{s,2} = \{AG(\neg(\xi_1 \wedge \xi_2))\}$.

Regarding liveness objectives $\omega_{l,N}$, we want to ensure that every task in its Idle state should eventually reach its Active state. Formally, one obtains:

$$\omega_{l,N} = \bigcup_{i\in\{1,\ldots,N\}} \{AG\left(\neg\xi_i \Rightarrow AF\xi_i\right)\}.$$

$AG(\neg\xi_1 \Rightarrow AF\xi_1)$
$AG(\neg\xi_2 \Rightarrow AF\xi_2)$

Those two are the liveness properties (any task on idle state will definitely be on the active state in the future)

In the end, solving each problem N–Tasks consists in finding a deterministic strategy Γ_{ω_N} so that $\forall\phi \in \omega_N, [S_{N\text{–}\mathsf{Tasks}}/\Gamma_{\omega_N}] \models \phi$, with $\omega_N = \omega_{s,N} \cup \omega_{l,N}$.

Partial Objectives The partial objectives automatically generated from the $\omega_{s,N}$'s and $\omega_{l,N}$'s above, according to the procedure explained in Section 4.6.3, are:

$$\omega'_{s,N} = \left\{ EG\left(\bigwedge_{i\in\{1,\ldots,N-1\}} \left(\bigwedge_{j\in\{i+1,\ldots,N\}} \neg\left(\xi_i \wedge \xi_j\right) \right) \right) \right\},$$

and $\omega''_{s,N} = \varnothing$.

Informally, partial objectives in $\omega'_{s,N}$ state that there exists execution paths where the mutual exclusion property is met.

On the other hand, the $\omega_{l,N}$'s lead to:

$$\omega'_{l,N} = \bigcup_{i\in\{1,\ldots,N\}} \{AG\left(\neg\xi_i \Rightarrow EF\xi_i\right), EG\left(\neg\xi_i \Rightarrow AF\xi_i\right)\},$$

and

$$\omega''_{l,N} = \bigcup_{i\in\{1,\ldots,N\}} \{EG\left(\neg\xi_i \Rightarrow EF\xi_i\right)\}.$$

4.7.2 Experimental Setup

We have implemented the simulated annealing, and genetic programming, and hybrid algorithms as described, using NuSMV [CCG+02] as a solver to derive the fitness of candidate deterministic strategies.

For simulated annealing, we have set the initial temperature to 20,000. The cooling schedule decreases the temperature by 0.8 in each iteration. Thus, the schedule ends after 25,000 iterations, when the temperature hits 0. In a failed execution, this leads to determining the fitness of 25,001 candidates.

We have taken the values suggested in [KP08] and [HS16] for genetic programming: $\lambda = 150$ candidates are considered in each step, $\mu = 5$ are kept, and we abort after 2,000 iterations. In a failed execution, this leads to determining the fitness of 290,150 candidates.

Points assigned for fitness measures are arbitrarily set to $m = 100$ for target objectives, and $m' = 80$ and $m'' = 10$ for partial objectives Those evaluation numbers are given to weight the candidates and distinguish among three levels of them according to the satisfaction of the specifications. This means that it is possible to take another values (eg. 20, 60, and 100) the most important thing is keep the difference among them clear (*cf.* Section 4.6.3).

The experiments have been conducted using a machine with an Intel core i7 3.40 GHz CPU and 16GB RAM.

4.7.3 Experimental Results

[2]The results are shown in Figures 4.5 and 4.7 and summarised in Table 4.1. Figure 4.5 shows the average time needed for synthesising a correct candidates.

The two factors that determine the average running time are the success rate and the running time for a full execution, successful or not.

These values are shown in Figure 4.7. Table 4.1 shows the average running time for single executions in seconds, the success rate in %, and the resulting overall running time.

[2]In all tables, execution times are in seconds; t is the mean execution time of single executions (succeeding or failing), and columns "expec. time" extrapolate t based on the success rate obtained in 100 single executions ("suc. %").

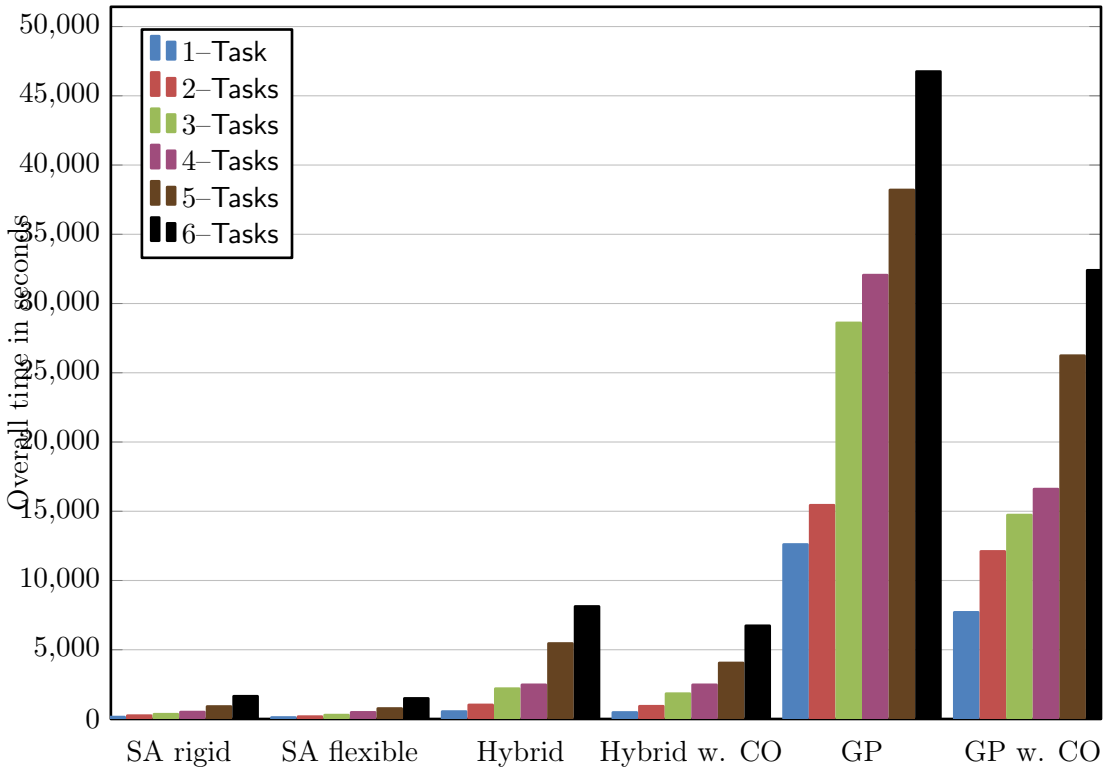


FIGURE 4.5: Overall time required for synthesising a correct candidate

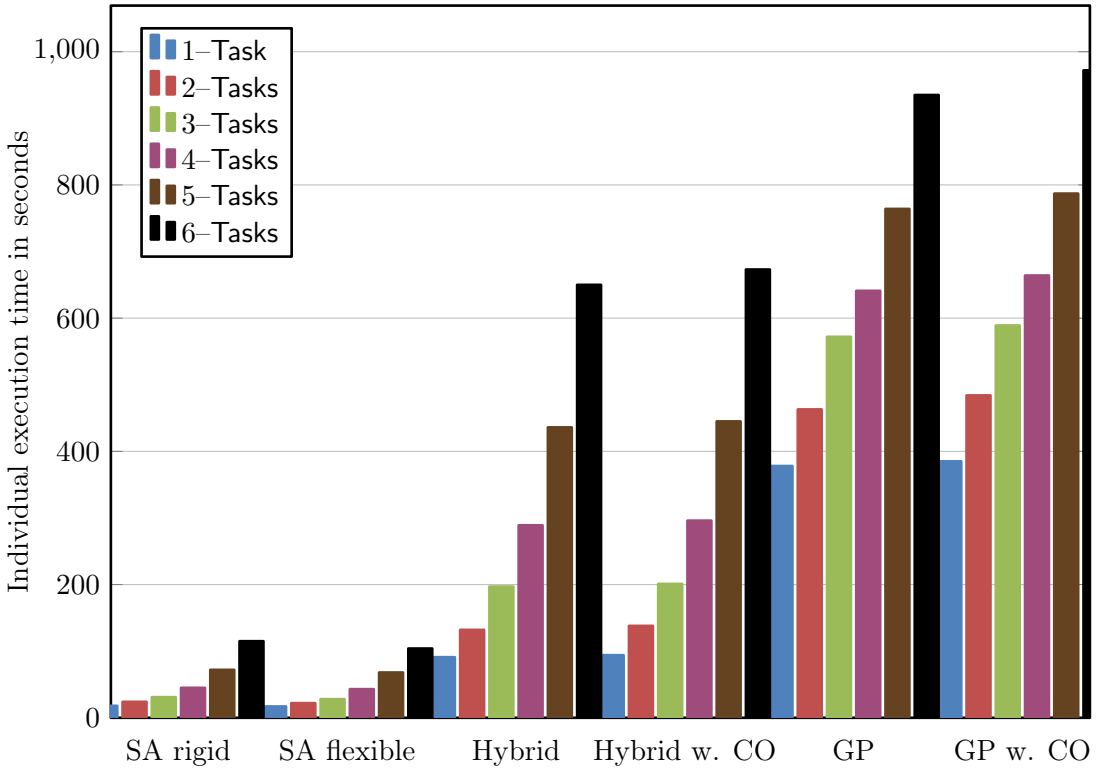


FIGURE 4.6: Average running time of an individual execution

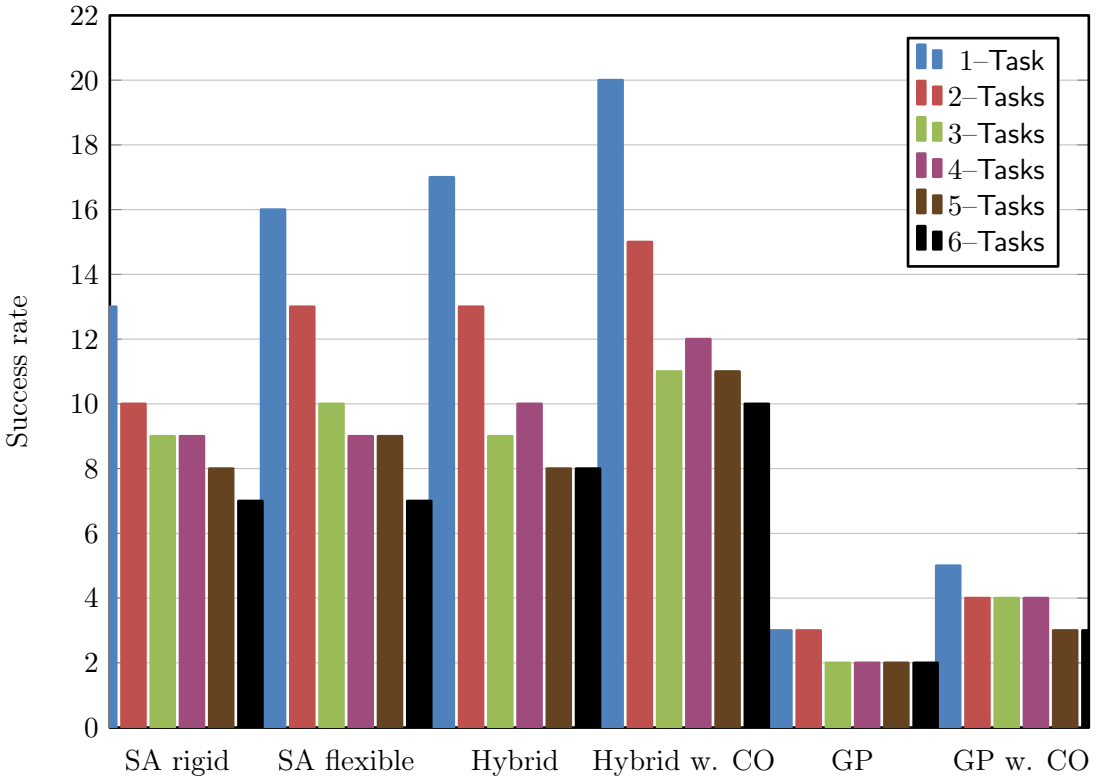


FIGURE 4.7: Success rate of individual executions

The best values (shortest expected running time or highest success rate) for each comparison printed in bold. Both simulated annealing and the hybrid approach significantly outperform the pure genetic programming approach. The low success rate for pure genetic programming suggests that the number of iterations might be too small.

However, as the individual execution time is already ways above the average time simulated annealing needs for constructing a correct candidate, we did not increase the number of iterations.

The advantage in the individual execution time between the classic and the hybrid version of genetic programming is in the range that is to be expected, as the number of calls to the model-checker is reduced. It is interesting to note that simulated annealing, where the shift from rigid to flexible evaluation might be expected to have a similar effect, does not benefit to the same extent.

It is also interesting to note that the execution time suggests that determining the fitness of candidates produced by simulated annealing is slightly more expensive. This was to be expected, as the average candidate size grows over time.

The penalty for longer candidates reduces this effect, but cannot entirely remove it. (This potential disadvantage is the reason why an occasional re-start provides better results than prolonging the search.)

As was the case in our previous work [HS16], it is interesting to note that both the pure and the hybrid approach to genetic programming benefit from crossovers.

TABLE 4.1: Search Techniques Comparison

	Search Tech.	t	suc. %	expec. time
1–Task	SA rigid	18	13	138.46
	SA flexible	**17**	16	**106.25**
	Hybrid	91	17	535.29
	Hybrid w. CO	94	**20**	470.00
	GP	378	3	12,600.00
	GP w. CO	385	5	7,700.00
2–Tasks	SA rigid	24	10	240.00
	SA flexible	**22**	13	**169.23**
	Hybrid	132	13	1,015.38
	Hybrid w. CO	138	**15**	920.00
	GP	463	3	15,433.33
	GP w. CO	484	4	12,100.00
3–Tasks	SA rigid	31	9	344.44
	SA flexible	**28**	10	**280.00**
	Hybrid	197	9	2,188.88
	Hybrid w. CO	201	**11**	1,827.27
	GP	572	2	28,600.00
	GP w. CO	589	4	14,725.00
4–Tasks	SA rigid	45	9	500.00
	SA flexible	**43**	9	**477.77**
	Hybrid	289	10	2,890.00
	Hybrid w. CO	296	**12**	2,466.66
	GP	641	2	32,050.00
	GP w. CO	664	4	16,600.00
5–Tasks	SA rigid	72	8	900.00
	SA flexible	**68**	9	**755.55**
	Hybrid	436	8	5,450.00
	Hybrid w. CO	445	**11**	4,045.45
	GP	764	2	38,200.00
	GP w. CO	787	3	26,233.33
6–Tasks	SA rigid	115	7	1,642.85
	SA flexible	**104**	7	**1,485.71**
	Hybrid	650	8	8,125.00
	Hybrid w. CO	673	**10**	6,730.00
	GP	935	2	46,750.00
	GP w. CO	972	3	32,400.00

Lastly, observe that in absolute terms, the expected execution times that we obtain for all our general search-based algorithms are several orders of magnitude higher than that obtained on similar examples when using a traditional symbolic DCS tool such as ReaX [BM14] (that is also able to enforce some restricted class of liveness objectives, when the controller is not triangulated). We discuss this aspect and its implications further in Section 4.8.4.

4.8 Parameters Setting

4.8.1 Crossover Ratio

We have studied the effect of changing the share between crossover and mutation in genetic programming.We considered a range from a 0% to 60% share of crossovers. Country

to our expectation, we did not observe any relevant effect of in- or decreasing the share of cross over for either classic or hybrid genetic programming. Interestingly, the running time per instance increased with the share of crossovers, which might point to a production of more complex programs. See Tables 4.2 and 4.3.

	ratio %	t	suc. %	expec. time
1 Task	0	378	4	9450
	20	385	5	7700
	40	403	5	8060
	60	418	4	10450
	80	425	3	14166.67
	100	438	1	43800
2 Tasks	0	475	3	15833.33
	20	484	4	12100
	40	491	4	12275
	60	501	3	16700
	80	509	2	25450
	100	521	1	52100
3 Tasks	0	571	3	19033.33
	20	589	4	14725
	40	597	3	19900
	60	606	3	20200
	80	613	1	61300
	100	627	1	62700
4 Tasks	0	658	3	21933.33
	20	664	4	16600
	40	679	4	16975
	60	687	3	22900
	80	693	2	34650
	100	711	1	71100
5 Tasks	0	776	1	77600
	20	787	3	26233.33
	40	792	3	26400
	60	799	2	39950
	80	804	2	40200
	100	815	1	81500
6 Tasks	0	961	2	48050
	20	972	3	32400
	40	981	2	49050
	60	989	2	49450
	80	997	2	49850
	100	1011	1	101100

TABLE 4.2: Crossover ratio for GP (Discrete Controller Synthesis)

	ratio %	t	suc. %	expec. time
1 Task	0	89	17	523.52
	20	94	20	470
	40	101	19	531.57
	60	109	19	573.68
	80	116	12	966.66
	100	124	5	2480
2 Tasks	0	127	13	976.92
	20	138	15	920
	40	146	15	973.33
	60	158	13	1215.38
	80	169	11	1536.36
	100	181	4	4525
3 Tasks	0	189	9	2100
	20	201	11	1827.27
	40	209	11	1900
	60	217	8	2712.5
	80	225	7	3214.28
	100	239	3	7966.66
4 Tasks	0	288	9	3200
	20	296	12	2466.66
	40	303	11	2754.54
	60	313	10	3130
	80	321	8	4012.5
	100	333	4	8325
5 Tasks	0	438	7	6257.14
	20	445	11	4045.45
	40	451	8	5637.5
	60	459	7	6557.14
	80	467	5	9340
	100	479	2	23950
6 Tasks	0	659	6	10983.33
	20	673	10	6730
	40	679	10	6790
	60	695	7	9928.57
	80	703	4	17575
	100	718	2	35900

TABLE 4.3: Crossover ratio for Hybrid (Discrete Controller Synthesis)

4.8.2 Temperature

Besides serving as a synthesis tool, the tool provides the user with the ability to compare the different generic search techniques. In [HS16] and [HBS17], we have used our technique to generate correct solutions for mutual exclusion, leader election and discrete controllers, the results show that simulated annealing is much (1.5 to 2 orders of magnitude) faster than genetic programming.

GP w/o crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	463	3	15433.33
	7	463	3	15433.33
	9	464	3	15466.67
250	5	943	5	18860.00
	7	943	5	18860.00
	9	943	5	18860.00
350	5	1517	9	16855.56
	7	1517	9	16855.56
	9	1518	9	16866.67
GP with crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	484	4	12100.00
	7	485	4	12125.00
	9	485	4	12125.00
250	5	969	7	13842.86
	7	969	7	13842.86
	9	969	7	13842.86
350	5	1557	10	15570.00
	7	1557	10	15570.00
	9	1557	10	15570.00

TABLE 4.4: Population size vs cost for GP (Discrete Controller synthesis 2-Tasks)

In order to test if the hypothesis from [Con90] that simulated annealing does most of its work during the middle stages—while being in a good temperature range—holds for our application, we have developed the tool to allow for 'cooling schedules' that do not cool at all, but use a constant temperature. In order to be comparable to the default strategy, we use up to $25,001$ iterations in each attempt.

We have run 100 attempts to create a correct candidates using different constant temperatures, with the success rates, average running times per iteration, and inferred expected overall running times shown in Table 4.7 and 4.6.

The findings support the hypothesis that some temperatures are much better suited than others: low temperatures provide a very small chance of succeeding, and the chances also go down at the high temperature end.

While the values for low temperatures are broadly what we had expected, the high end performed better than we had thought.

This might be because some small guidance is maintained even for infinite temperature, as a change that is decreasing the fitness is taken with an (almost) 50% chance in this case, while increases are always selected.

The figures, however, are much worse than the figures for the good temperature range of $10,000$ to $16,000$.

The best results have been obtained at a temperature of $10,000$, with an expected time of 68 and 71 seconds for two and three shared bits, respectively.

Hybrid w/o crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	132	13	1015.38
	7	132	13	1015.38
	9	131	13	1007.69
250	5	241	18	1338.88
	7	241	18	1338.88
	9	242	18	1344.44
350	5	403	24	1679.16
	7	403	24	1679.16
	9	403	24	1679.16
Hybrid with crossover				
population size	slctd cand.	t	suc. %	expec. time
150	5	138	15	920.00
	7	139	15	926.66
	9	139	14	992.85
250	5	218	19	1147.36
	7	218	19	1147.36
	9	218	19	1147.36
350	5	340	24	1416.66
	7	340	24	1416.66
	9	340	24	1416.66

TABLE 4.5: Population size vs cost for Hybrid (Discrete Controller synthesis 2-Tasks)

Notably, these results are better than the running time for the cooling schedule that uses a linear decline in the temperature. Starting at $20,000$ and terminating at 0, it has the same average temperature.

In the light of the results from the second experiment, it seems likely that the last third of the improvement cycles in this schooling schedule had little avail.

4.8.3 Initial Population vs Cost

One of the important parameters of genetic programming is the initial population size using our synthesis tool. In order to investigate how the population size effect on our synthesis approach and check the cost of the large size, we initiate the population size to different sizes see Tables 4.4 and 4.5. We found that increasing the size of the initial population increase the cost of finding a good solution dramatically as shown on Figure 4.8.

Broadly speaking, increasing the population size reduces the number of iterations and increase the success rate, but it also increase the computation time each iteration required. We have compared the effect of varying the population size.

4.8.4 Discussion

Inspired by our previous investigations for program synthesis using general search techniques [HS16], and in an effort to inquire further applications of such techniques, we have

	const temp	t	suc. %	expec. time
1-Task	0.7	163	0	∞
	400	93	0	∞
	4000	54	7	771.42
	7000	39	12	325
	10000	18	19	94.73
	13000	22	20	110
	16000	29	19	152.63
	20000	37	17	217.64
	25000	43	15	286.66
	30000	49	15	326.66
	40000	53	13	407.69
	50000	59	12	491.66
	100000	72	11	654.54
2-Task	0.7	177	0	∞
	400	99	0	∞
	4000	58	6	966.66
	7000	47	9	522.22
	10000	29	14	207.14
	13000	33	15	220
	16000	39	13	300
	20000	47	11	427.27
	25000	56	10	560
	30000	67	10	670
	40000	75	9	833.33
	50000	82	7	1171.42
	100000	94	7	1342.85
3-Task	0.7	192	0	∞
	400	163	0	∞
	4000	88	6	1466.66
	7000	45	9	500
	10000	26	11	236.36
	13000	31	11	281.81
	16000	37	10	370
	20000	42	10	420
	25000	47	9	522.22
	30000	56	8	700.00
	40000	63	9	700.00
	50000	79	7	1128.57
	100000	98	7	1400.00

TABLE 4.6: Comparison for search temperature for Discrete Controller synthesis

defined a class of DCS problems where deterministic strategies are sought.

We adapted our algorithms to seek solutions for these problems, and conducted an experimental evaluation using a scalable instance of a DCS problem.

Results proved to be consistent with our assumption that the relative performance of the algorithms would match our previous results for program synthesis.

	const temp	t	suc. %	expec. time
4-Task	0.7	332	0	∞
	400	167	0	∞
	4000	98	3	3266.66
	7000	65	6	1083.33
	10000	39	9	433.33
	13000	43	11	390.90
	16000	58	9	644.44
	20000	67	9	744.44
	25000	81	7	1157.14
	30000	89	6	1483.33
	40000	95	6	1583.33
	50000	103	5	2060
	100000	118	4	2950
5-Task	0.7	298	0	∞
	400	153	0	∞
	4000	98	4	2450
	7000	79	6	1316.66
	10000	61	9	677.77
	13000	67	10	670
	16000	73	8	912.5
	20000	81	6	1350
	25000	89	6	1483.33
	30000	102	4	2550
	40000	116	3	3866.66
	50000	128	4	3200
	100000	178	3	5933.33
6-Task	0.7	613	0	∞
	400	598	0	∞
	4000	278	3	9266.66
	7000	125	5	2500
	10000	99	8	1237.5
	13000	115	9	1277.77
	16000	127	9	1411.11
	20000	134	7	1914.28
	25000	152	6	2533.33
	30000	159	6	2650
	40000	168	6	2800
	50000	192	5	3840
	100000	253		6325

TABLE 4.7: Comparison for search temperature for Discrete Controller

As noted in Section 4.7, our experimental results do not compare favourably with existing symbolic DCS tools.

Yet, our implementations are proofs of concept, and one can think of numerous practical improvements that constitute inescapable ways to pursue investigating efficient symbolic DCS algorithms using simulated annealing

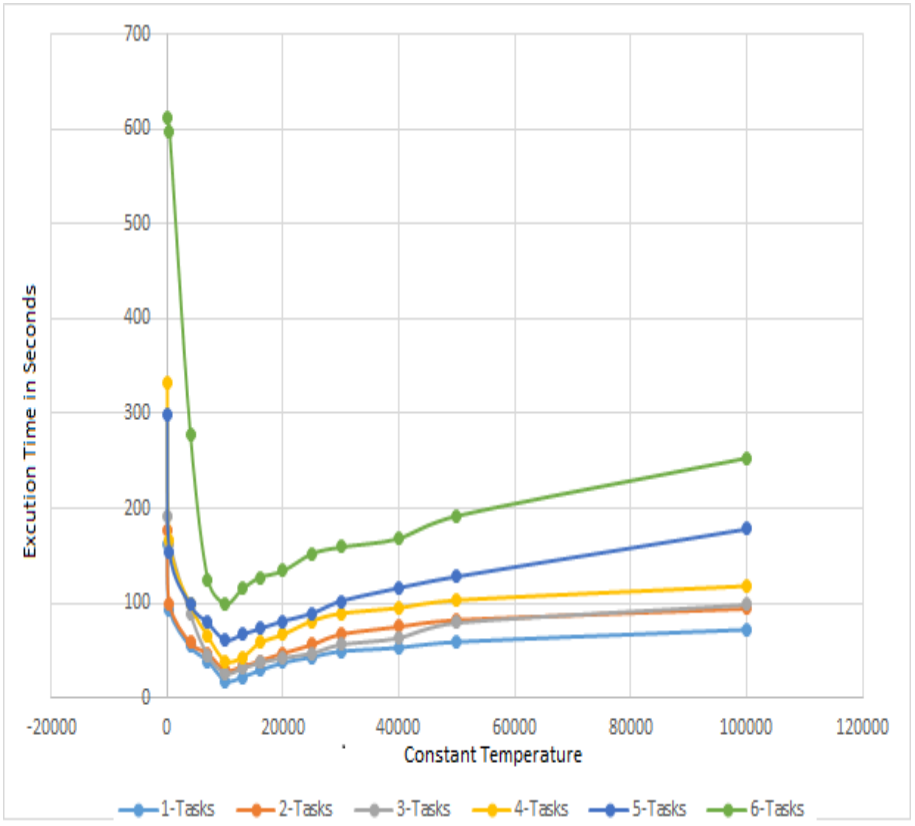


FIGURE 4.8: Temperature Range for Controller Synthesis

For instance, canonically representing symbolic candidate strategies using BDDs instead of syntactic trees would allow building a cache of fitness results, and thereby avoid re-evaluating the fitness of equivalent candidate strategies.

Similarly, implementing the algorithms in a symbolic model-checker could permit to avoid the very costly—and often performance bottleneck in case of model-checkers using BDDs—reconstruction of symbolic representations at every iteration of the algorithms.

At last, note that our search based algorithms do not require the computation of the unsafe region to produce deterministic strategies. Hence, considering the current advances in model checking technologies, our algorithms might constitute ways to solve DCS problems on some classes of infinite-state systems for which the unsafe region cannot be computed.

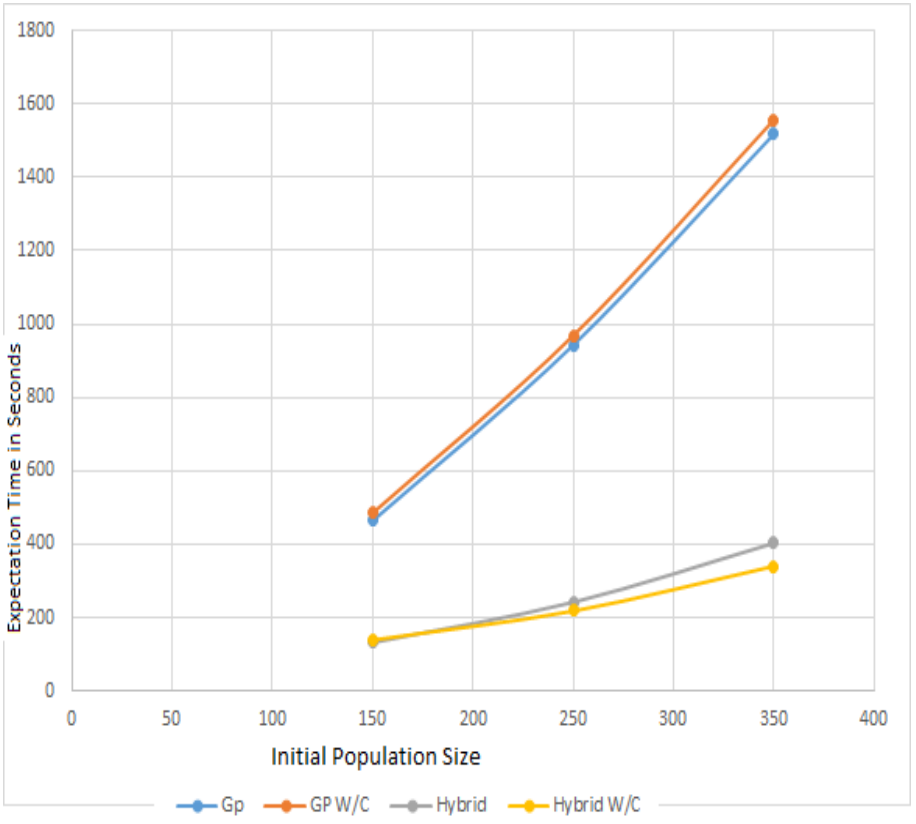


FIGURE 4.9: Initial Population Size vs Cost for Discrete Controller synthesis

Chapter 5

Complexity

5.1 Introduction

Discrete Controller Synthesis (DCS) and *Program Synthesis* have a similar theoretical background: they are produce a control strategy and an implementation automatically, and also correct by construction.

The main differences between these two classes of problems, that DCS typically operates on the model of a plant. It seeks the automated construction of a strategy to control the plant, such that its runs satisfy a set of given objectives [RW89, AMP95].

Similarly, program synthesis seeks to infer an implementation, often of a reactive system, such that the runs of this system satisfy a given specification. While the main argument supporting this technique is practical, it is interesting to consider the complexity of this approach, too. For the estimation of the complexity, we look at recurrent application of simulated annealing, as, from the figures from the previous Section, this seems to be the most promising approach.

Also, applying the procedure repeatedly is fairly normal, as not 100% of the attempts lead to a deterministic strategy that satisfies all required objectives.

For our complexity consideration, we consider cooling schedules that change slowly.

The reason for this is that the search space for a given cooling schedule is finite, as only a finite set of deterministic strategies can be constructed with a fixed cooling schedule.

We naturally can only refer to probabilistic complexity here:

it is always possible to only consider two neighbouring candidate strategies during all runs (this is the case when candidate two is always selected as a mutation of candidate one and, vice versa, candidate one is always chosen as a mutation of candidate two), even when re-starting infinitely often.

This chapter provides a summary of the complexity analysis of the approach used in this thesis. The rest of this section is organised as follows:

5.2.1 presents the complexity analysis of the program synthesis approach. Section 5.2.2 provides a summary he complexity analysis of the discrete controller synthesis approach.

5.2 Complexity Analysis

5.2.1 Program Synthesis

Theorem 5.1. *Let s be the length of the specification and m the minimal size of the correct program. We show that there is a recurrent cooling schedule such that, with very high probability,*

- *the space required is the space required for model checking programs of length $O(m)$ against specifications of length s and*
- *the time requirement is $m^{O(m)}$ times the time required for model checking programs of length $O(m)$ against specifications of length s.*

With very high probability means that, for all $p < 1$, one gets a result with probability greater than p in the mentioned time and space class. Note that this refers to the same cooling schedule.

As a preparation for the proof of Theorem 5.2, we estimate the chance of producing a correct program in exactly m steps by a particular derivation of the program tree. The chance is the product of the m transitions being made.

An individual transition can be described as the product of the chance that it is selected by the mutation algorithm and the chance that the algorithm continues with the mutated form.

The chance of selecting a particular position for the mutation is $\Omega\frac{1}{i}$ in the i^{th} step, and the chance of attempting a particular mutation when this position is selected is a constant. (For the argument, it suffices that it is at least $\frac{1}{O(\mathsf{poly}(m))}$.)

If the chance is bounded from below by a constant – or even by a polynomial $\frac{1}{O(\mathsf{poly}(m))}$, then the chance is bounded from below by $\frac{1}{O(m)!}$.

For a given number n of steps (e.g., $n = m$ if we have an oracle that tells us the size of the program we are looking for), it is easy to construct a cooling schedule with this property: any cooling schedule that keeps the temperature sufficiently high for at least n steps.

To avoid an unduly high time or space consumption of one iteration, the cooling after this should proceed quickly, such that the overall number of mutation attempts considered, n_{attempts}, is in $O(n)$.

As we would normally not know a suitable $n \geq m$, we suggest to adjust the cooling schedule over time by increasing it slowly: in the i^{th} iteration, we could use, for some constant $c \in \mathbb{N}$, the cooling scheme for $n_i = c + \max\{k \in \mathbb{N} \mid k! \leq i\}$. Let us choose $c = 0$ for the proof.

of Theorem 5.1. For $n_i < m$, we estimate the chance of creating a correct program with 0. Note that $n_{m!} \geq m$. The time and space consumption of each of these steps can be estimated by the cost of model checking a program of size $m_{\mathsf{attempts}} \in O(m)$ against a specification of size s. There are less than $m!$ of these attempts.

For $n_i \geq m$, the chance of creating a correct program is at least $\frac{1}{O(m)!}$. To create a program with an arbitrary (but fixed) chance of at least $p \in [0, 1[$ therefore requires $O(m)!$ such steps.

The time and space consumption of each these steps can be estimated by the cost of model checking a program of size $O(m)$ against a specification of size s. □

Note that this technique does not qualify as a decision procedure, as it cannot provide a negative answer. (What it can be used to provide is an answer that, for a given $m \in \mathbb{N}$ and $p \in [0, 1[$ there is no program of size at most m with a chance of at least p.)

Note further that the proof does not refer to a particular specification language. For space requirements, the complexity is, for relevant languages like LTL or CTL, as good as one can hope.

With regard to time complexity, the complexity of synthesis is exponential for CTL and doubly exponential for LTL. If the expected time complexity of this algorithm is higher depends on the question of whether or not PSPACE equals EXPTIME [FPS15].

5.2.2 Discrete Controller Synthesis

For the estimation of the complexity, we look at recurrent application of simulated annealing, as, from the figures from the previous Section, this seems to be the most promising approach.

Also, applying the procedure repeatedly is fairly normal, as not 100% of the attempts lead to a deterministic strategy that satisfies all required objectives.

For our complexity consideration, we consider cooling schedules that change slowly.

The reason for this is that the search space for a given cooling schedule is finite, as only a finite set of deterministic strategies can be constructed with a fixed cooling schedule.

We naturally can only refer to probabilistic complexity here:

it is always possible to only consider two neighbouring candidate strategies during all runs (this is the case when candidate two is always selected as a mutation of candidate one and, vice versa, candidate one is always chosen as a mutation of candidate two), even when re-starting infinitely often.

Theorem 5.2. *Let s be the length of the specification (some of the lengths of the target objectives in ω) and m the minimal size of the correct deterministic strategy. We show that there is a recurrent cooling schedule such that, with very high probability,*

- *the space required is the space required for model-checking STSs of length $O(m)$ against specifications of length s and*
- *the time requirement is $m^{O(m)}$ times the time required for model-checking STSs of length $O(m)$ against specifications of length s.*

With very high probability means that, for all $p < 1$, one gets a result with probability greater than p in the mentioned time and space class. Note that this refers to the same cooling schedule.

As a preparation for the proof of Theorem 5.2, we estimate the chance of producing a correct strategy in exactly m steps by a particular derivation of the corresponding predicate trees. The chance is the product of the m transitions being made.

An individual transition can be described as the product of the chance that it is selected by the mutation algorithm and the chance that the algorithm continues with the mutated form.

The chance of selecting a particular position for the mutation is $\Omega\left(\frac{1}{i}\right)$ in the i^{th} step—this lower bound stems from taking the weight of subtrees into account in the random walk, which can be used to select all nodes with equal probability—and the chance of attempting a particular mutation when this position is selected is a constant.

(For the argument, it suffices that it is at least $\frac{1}{O(\mathsf{poly}(m))}$.) If the chance is bounded from below by a constant – or even by a polynomial $\frac{1}{O(\mathsf{poly}(m))}$, then the chance is bounded from below by $\frac{1}{O(m)!}$.

For a given number n of steps (*e.g.,* $n = m$ if we have an oracle that tells us the size of the predicate trees we are looking for), it is easy to construct a cooling schedule with this property: any cooling schedule that keeps the temperature sufficiently high for at least n steps.

To avoid an unduly high time or space consumption of one iteration, the cooling after this should proceed quickly, such that the overall number of mutation attempts considered, n_{attempts}, is in $O(n)$.

As we would normally not know a suitable $n \geq m$, we suggest to adjust the cooling schedule over time by increasing it slowly: in the i^{th} iteration, we could use, for some constant $c \in \mathbb{N}$, the cooling scheme for $n_i = c + \max\{k \in \mathbb{N} \mid k! \leq i\}$. Let us choose $c = 0$ for the proof.

Proof of Theorem 5.2. For $n_i < m$, we estimate the chance of creating a correct strategy with 0. Note that $n_{m!} \geq m$.

The time and space consumption of each of these steps can be estimated by the cost of model-checking a strategy represented by predicate trees of size $m_{\mathsf{attempts}} \in O(m)$ against a specification of size s.

There are less than $m!$ of these attempts.

For $n_i \geq m$, the chance of creating a correct strategy is at least $\frac{1}{O(m)!}$.

To create a strategy with an arbitrary (but fixed) chance of at least $p \in [0, 1]$ therefore requires $O(m)!$ such steps. The time and space consumption of each of these steps can be estimated by the cost of model-checking a strategy represented by predicate trees of size $O(m)$ against a specification of size s.

□

Note that this technique does not qualify as a decision procedure, as it cannot provide a negative answer. (What it can be used to provide is an answer that, for a given $m \in \mathbb{N}$ and $p \in [0, 1[$ there is no strategy of size at most m with a chance of at least p.)

Note further that the proof does not refer to a particular specification language.

For space requirements, the complexity is, for relevant languages like LTL or CTL, as good as one can hope.

With regard to time complexity, the complexity of synthesis is exponential for CTL and doubly exponential for LTL.

If the expected time complexity of this algorithm is higher depends on the question of whether or not PSPACE equals EXPTIME [FPS15].

Chapter 6

Implementation

6.1 Abstract

PranCS [HSB17] is a tool for protocol and control synthesis. It is based on exploiting general search techniques, simulated annealing and genetic programming, that use model checking for the fitness function. Based on this fitness, simulated annealing and genetic programming is used as a general search technique for homing in on an implementation. We use NuSMV as a back-end for the individual model checking tasks and a simple candidate mutator to drive the search.

Our **Pr**octocol **an**d **C**ontrol **S**ynthesis (PranCS) tool is designed to exploring the parameter space of different synthesis techniques. Besides using it to synthesise a discrete control strategy for systems on chips and for protocol adapters for the coordination of different threads, it also allows for researching the influence turning various screws in the synthesis process.

For simulated annealing, it PranCS allows the user to define the behaviour of the cooling schedule, and the population size for genetic programming can be selected by the user. Additionally, PranCS offers different automated techniques for the inference of to quantify partial compliance with a specification and to infer the fitness of a candidate implementation based on this partial compliance.

PranCS also allows for choosing small random seeds as well as user defined seeds to allow the user to define a starting point for the search, e.g. for candidate repair.

In this Chapter we summarize the synthesis tool structure, use and the capabilities that enables the user to select the parameters and set the tools initial values. An introduction for the chapter will be on Section 6.2, an overview of the tool Section 6.3, exploring of the parameter space on Section 6.4, and conclusion on Section 6.5.

6.2 Introduction

Discrete Controller Synthesis (DCS) and *Program Synthesis* have similar goals: they are automated techniques to infer a control strategy and an implementation, respectively, that is correct by construction.

There are mild differences between these two classes of problems. DCS typically operates on the model of a plant. It seeks the automated construction of a strategy to control the plant, such that its runs satisfy a set of given objectives [RW89, AMP95].

Similarly, program synthesis seeks to infer an implementation, often of a reactive system, such that the runs of this system satisfy a given specification. Program synthesis is particularly attractive for the construction of protocols that govern the intricate interplay between different threads; we use mutual exclusion and leader election as examples.

DCS algorithms have been used to avoid deadlock in multi-threaded programs [WLK+09], for resource management correction in embedded systems [ACMR03, BMM13], and to enforce fault-tolerance [GR09]. A closely related algorithm was also applied for device driver synthesis [RCK+09].

DCS and program synthesis depend on a concept similar to principles of model-checking [CGP99, BCM+90]. Model-checking refers to automated techniques that determines whether or not a system satisfies a number of properties. Traditional DCS algorithms are inspired by this approach. Given a model of the plant, they first *exhaustively* compute an unsafe portion of the state-space to avoid for the desired objectives to be satisfied, and then derive a strategy that avoids entering the unsafe region.

Finally, a controller is built that restricts the behaviour of the plant according to this strategy, so that it is guaranteed to always comply with its specification. Just as for model-checking, *symbolic* approaches for solving DCS problems have been successfully investigated [AMP95, CKN98, MBLL00, BM14].

Techniques based on genetic programming [CJ01, HJJ03, Joh07, KP08, KP09a, HS16, HBS17], as well as on simulated annealing [HS16, HBS17], have been tried for program synthesis. Instead of performing an exhaustive search, these techniques proceed by using a measure of the fitness—reflecting the question '*How close am I to satisfying the specification?*'—to find a short path towards a solution.

Among the generic search techniques that look promising for this approach, we focus on *genetic programming* [Koz92] and *simulated annealing* [CJ01, HJJ03]. When applied to program synthesis, both search techniques work by successively mutating candidate programs that are deemed 'good' by using some measure of their fitness.

We obtain their fitness for meeting the desired objectives by using a model-checker to measure the share of objectives that are satisfied by the candidate program, *cf.* [KP08, KP09a, HS16, HBS17].

Simulated annealing keeps one candidate solution. In a sequence of iterations, it mutates the current candidate and compares the fitness of the old and new candidate. If the fitness increases, the new candidate is always maintained. If it decreases, a random process decides if the new candidate replaces the old one in the next iteration.

The chances of the new candidate to replace the old one then decrease with the gap in the fitness and increase with the temperature. Thus, a lower temperature makes the system 'stiffer'. Genetic programming maintains a population of candidate programs over

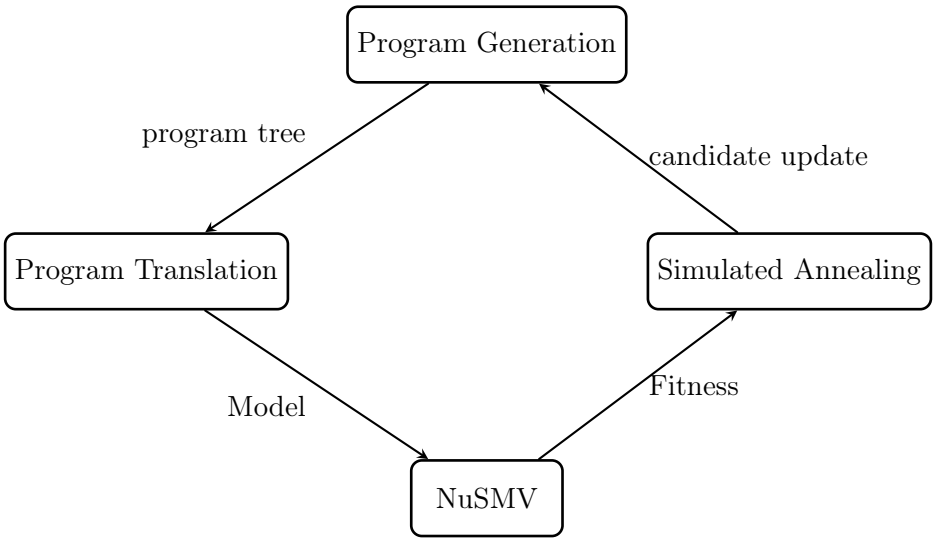


FIGURE 6.1: PranCS Structure

a number of iterations, generating new ones by mutating or mixing candidates randomly selected based on their fitness.

We describe the tool PranCS, which implements the simulated annealing based approach proposed in [HS16, HBS17] as well as approaches based on similar genetic programming from [KP08] and [KP09a]. PranCS uses quantitative measures for partial compliance with a specification, which serve as a measure for the fitness (or: quality) of a candidate solution.

Furthering on the comparison of simulated annealing with genetic programming [HS16, HBS17], we extend the quest for the best general search technique by: (i) looking for good cooling schedules for simulated annealing; and (ii) investigating the impact of the population size for genetic programming.

6.3 Overview of PranCS

PranCS implements several generic search algorithms that can be used for solving DCS problems as well as for synthesising programs.

6.3.1 Representing Candidates

The representation of candidates depends on the kind of problems to solve. Candidate programs are represented as abstract syntax trees according to the grammar of the sought implementation. They feature conditional and iteration statements, assignments to one variable taken among a given set, and expressions involving such variables. Candidates for DCS only involve a series of assignments to a given subset of Boolean variables involved in the system (called "*controllables*").

6.3.2 Structure of PranCS

The structure of PranCS is shown in Figure 6.1. Via the user interface, the user can select a search technique, and enter the problem to solve along with values for relevant parameters of the selected algorithm. For program synthesis, the user enters the number, size, and type of variables that candidate implementations may use, and whether thay may involve complex conditional statements ("if" and "while" statements). DCS problems are manually entered as a series of assignments to state variables involving expressions expressed on state and input variables; the user also lists the subset of input variables that are "controllable". In both cases, the user also provides the specification as a list of objectives.

1. Generator

   The Generator uses the parameters provided to either generate new candidates or to update them when required during the search.

   The parameters are sent to the Generator, which generates new candidate according to the given parameters, using a 'grow' method [Koz92], in which the candidate tree is built starting from the root.

   It takes the depth of the current node to be generated as argument. If the depth is less than the maximum tree depth, a node is chosen randomly from the set of terminals and binary operators. If the depth equals the maximum tree depth, then a node is chosen from the set of terminals (leafs).

   The candidates take the form of a vector of predicates over state and non-controllable input variables of symbolic transition system, and the goal to find a good candidate that satisfied the specifications.

2. Translator & NuSMV

   We use NuSMV [CCG+02] as a model-checker. Every candidate is translated into the modelling language of NuSMV using a method suggested in [CJ01]. (We give some details about this translation in Chapter 1.

   In this translation, the candidate is converted into very simple statements (similar to assembly language). To simplify the translation, the lines of the algorithm are first labelled, and this label is then used as a pointer that represents the program counter *(PC)*. From this intermediate language, the NuSMV model is then built by creating *(case)* and *(next)* statements that use the *PC*.

   The resulting model is then model-checked against the desired properties. The result forms the basis of a fitness function for the selected search technique.

3. Fitness Measure

The model checking results form the basis of a fitness function for the selected search technique. To design a fitness measure for candidates, we make the hypothesis that the share of objectives that are satisfied so far by a candidate is a good indication of its suitability *w.r.t.* the desired specification.

We additionally observe that weaker properties that can be mechanically derived are useful to identify good candidates worth selecting for the generation of further potential solutions. For example, if a property shall hold on all paths, it is better if it holds on some path, and even better if it holds almost surely.

4. Search Technique

   Based on the model checking results, we derive a quantitative measure for the fitness (as a level of partial correctness) of a candidate. The fitness measure obtained for a candidate is used as a fitness function for the selected search technique. If a candidate is evaluated as correct, we return (and display) it to the user.

   Otherwise, depending on the search technique selected and the old and new fitness measure/s, the current candidate or population is updated, and one or more candidates are sent for change to the Generator. The process is re-started if no solution has been found in a predefined number of steps (genetic programming) or when the cooling schedule expires (simulated annealing).

6.3.3 Selecting and Tuning Search Techniques

In terms of search techniques, PranCS implements the following methods: *genetic programming*, and *simulated annealing*. [KP09a] extends genetic programming by considering the fitness as a pair of 'safety-fitness' and 'liveness-fitness', where the latter is only used for equal values of 'safety-fitness'. Building upon this idea, we define two flavours for both simulated annealing and genetic programming: *rigid* (where the classic fitness function is used) and *safety-first*, which uses the two-step fitness approach as above.

Further, genetic programming can be used with or without crossovers between candidates [HS16, HBS17].

Depending on the selected search technique, the tool allows the user to input parameters that control the dynamics of the synthesis process.

These parameters determine the likelihood of finding a correct program in each iteration and the expected running time for each iteration, and thus heavily influence the overall search speed. For the genetic programming approach, the parameters include the population size, the number of selected candidates, the number of iterations, and the crossover ratio. For simulated annealing, the user chooses the initial temperature and the cooling schedule.

Figure 6.2 shows the graphical user interface of PranCS.

The input specification should be a list of NuSMV specifications. The tool will produce weaker specifications for the purpose of producing a fitness measure.

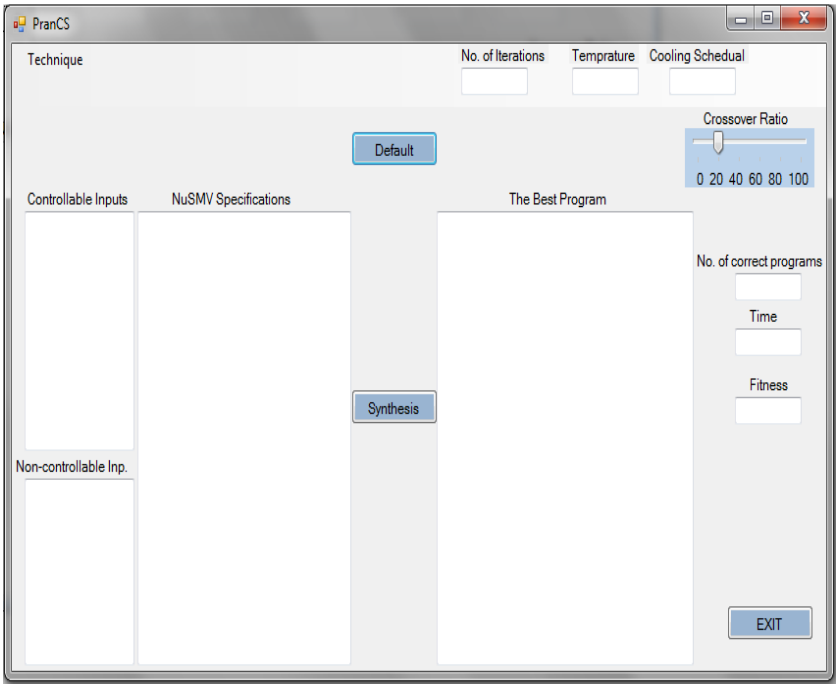


FIGURE 6.2: Graphical User Interface. PranCS allows the user to fine-tune each search technique by means of dedicated parameters.

Finally, the user defines the signature of the candidate by selecting input variables that are used to build the candidate that the user want to synthesise.

The model checker NuSMV is used for the individual model checking tasks. If the candidate satisfied all required properties, then the synthesiser returns it as a correct candidate. Otherwise, it will compare the fitness of the current candidate/s with the (stored) fitness value of the candidate/s it is derived from by mutation.

(This is the currently stored candidate or population.) The selection of the candidate / population to be kept is determined by the selected generic search technique. For simulated annealing, if the fitness is lower, then the tool will update the stored candidate with the probability defined by the loss in fitness and the current temperature taken from the cooling schedule.

If the fitness is not lower, the tool will always replaces the stored candidate by the mutated one. When the end of the cooling schedule is reached, the tool aborts.

6.3.4 Parameters for Simulated Annealing

In simulated annealing (SA), the intuition is that, at the beginning of the search phase, the temperature is high, and it cools down as time goes by. The higher the temperature, the higher is the likelihood that a new candidate solution with inferior fitness replaces the previous solution.

While this allows for escaping local minima, it can also happen that the candidates develop into an undesirable direction. For this reason, simulated annealing does not continue for ever, but is re-started at the end of the cooling schedule.

Consequently, there is a sweet-spot in just how long a cooling schedule should be and when it becomes preferable to re-start, but this sweet-spot is difficult to find. We report our experiments with PranCS for tuning the cooling schedule in Section 6.4.

6.3.5 Parameters for Genetic Programming

For genetic programming (GP), the parameters are the initial population size, the crossover vs mutation ratio, and the fitness measure used to select the individuals. The population size affects the algorithm in two ways: larger populations could provide better diversity. However, it also increases the number of iterations required to find a good solution. We investigate how the population size and crossover ratio affect the performance of these algorithms.

6.4 Exploration of the Parameter Space

In [HS16], we have used our techniques to generate correct solutions for mutual exclusion and leader election problems; we have then exercised the same techniques to solve scalable DCS problems [HBS17]. In both cases, and with parameter values borrowed from [KP09a, KP08], we could already accelerate synthesis significantly using simulated annealing compared to genetic programming (by 1.5 to 2 orders of magnitude).

In this work, our aim is to further explore the performance impact of the parameters for each search technique. We thus reuse the same scalable benchmarks as in [HS16, HBS17][1]: program synthesis problems consist of mutual exclusion ("2/3 shared bits") and leader election ("3/4 nodes"); DCS problems compute controllers enforcing mutual exclusions and progress between 1 to 6 tasks modelled as automata ("1/6-Tasks").

6.4.1 Exploring Population Size & Crossover Ratio

We have carried out several experiments in order to investigate how the population size and crossover ratio effect our synthesis approach when using genetic programming, with varying population sizes; details of our results are presented in Chapter 3 and Chapter 4. Although results indicate that some crossover ratio that is good in general may exist, the negative impact of small population sizes on success rates can in practice be mitigated by the faster execution times.

We found that the initial population size dramatically increases the overall expected time for finding a good solution.

Continuing on our investigations of the parameters, the "best" initial population size and crossover ratio are fixed and a new measure for the fitness applied.

[1] In all tables, execution times are in seconds; t is the mean execution time of single executions (succeeding or failing), and columns "expec. time" extrapolate t based on the success rate obtained in 100 single executions ("suc. %").

	Search Tech.	t	suc. %	expec. time
1-Task	Rigid SA	20	13	153
	Safety-first SA	**19**	16	**118**
2-Tasks	Rigid SA	25	10	250
	Safety-first SA	**24**	13	**184**
3-Tasks	Rigid SA	33	9	366
	Safety-first SA	**29**	10	**290**
4-Tasks	Rigid SA	47	9	522
	Safety-first SA	**43**	9	**477**
5-Tasks	Rigid SA	76	8	950
	Safety-first SA	**70**	9	**777**
6-Tasks	Rigid SA	119	7	1,700
	Safety-first SA	**106**	7	**1,514**

TABLE 6.1: Synthesis times with the best parameters observed for Simulated Annealing applied to our DCS benchmarks

$\|P\|$	slctd cand.	t	suc. %	expec. time
150	5	138	15	920
	7	139	15	926
	9	139	14	992
250	5	218	19	1147
	7	218	19	1147
	9	218	19	1147
350	5	340	24	1416
	7	340	24	1416
	9	340	24	1416

TABLE 6.2: On the left: Safety-first GP with crossover for DCS (2-Tasks only), with Various Population Sizes ($|P|$).

6.4.2 Exploring Cooling Schedules

We have studied good temperature ranges by keeping the temperature constant during a "cooling" schedule with 25,000 steps. We give the results of these experiments in Chapter 3 and Chapter 4. A robust temperature sweet-spot clearly exists for our scalable benchmarks, suggesting that the quest for robust and generic good cooling schedules is worth pursuing.

Adaptive cooling schedule enables the decrease of the temperature to be low when reach a good temperature otherwise it decremented normally. We have set the initial temperature to $20,000$. The cooling schedule decreases the temperature by 0.8 in each iteration in the normal movement and decrease it by 0.2 in the good temperature,

6.5 Conclusion

The results from [HS16, HBS17] already turn in favour of simulated annealing over the other general search technique we have studied.

These results shown on Table 6.2 (given in % and seconds, respectively, with the best values in bold) indicate that using simulated annealing with an adaptive cooling schedule slightly increase the single execution time and also overall time. We use a scalable example of DCS problem from [HBS17] each problem N-Tasks was built using N instances.

Together with our extensive exploration of the parameter space, the evaluation of PranCS indicates that simulated annealing is faster that genetic programming (we report some synthesis times with the best parameters observed for simulated annealing in Table 6.2), and that some temperature ranges are more useful than others.

In order to integrate this result into the cooling schedule we plan to use an adaptive cooling schedule, in which the decrements of the temperature depends on the improvement of the fitness.

Chapter 7

Conclusion

This chapter provides a summary of the research work described in this thesis, the main findings together with the contributions. The rest of this section is organised as follows: Section 7.1 provides a summary of the work presented while Section 7.2 presents the main findings in the context of the research question and research motivations identified in Chapter 1.

7.1 Summery

We have implemented an automated programming technique based on simulated annealing and genetic programming, both in the pure form of and the arguably hybrid form.

The results are very clear and in line with the expectation we had drawn from the literature [Dav87, LMST96, MS96]. When crossovers are not used, the main difference between the established genetic programming techniques and simulated annealing is the search strategy of using many and using a single instance, respectively. The data gathered confirms that an increase of the number of iterations can easily overcompensate the broader group of candidates kept in genetic programming. In our experiments, we have used an increase that fell short of creating the same expected running time for a single full execution (with or without success), and yet outperformed even the hybrid approach w.r.t. the success rate on three of our four benchmarks. We have also added variations of genetic programming that include crossover to validate the assumption that crossovers do not lead to an annihilation of the advantage, but it proved that the hybrid approach, and thus the stronger competitor, does not benefit much from using crossover. The double advantage of shorter running time and higher success rate led to an improvement of 1.5 to 2 orders of magnitude compared to pure genetic programming (with and without crossover), and between half an order and one order of magnitude when compared to the hybrid approach (with or without crossover).

The results from [HS16] indicate that simulated annealing is the better general search technique to use. It will be interesting to see if these factors are essentially constant, or if they depend heavily on the circumstances. Together with the later extensions, the tool

evaluation indicates that there are good temperature ranges. Inspired by our previous investigations for program synthesis using general search techniques [HS16], and in an effort to inquire further applications of such techniques, we have defined a class of DCS problems where deterministic strategies are sought. We adapted our algorithms to seek solutions for these problems, and conducted an experimental evaluation using a scalable instance of a DCS problem. Results proved to be consistent with our assumption that the relative performance of the algorithms would match our previous results for program synthesis.

7.2 Main Findings and Contributions

This section presents the main findings from the research work presented in this thesis. In order to answer the research question from Section 1.3, the resolution of a number of subsidiary research questions was required. The work described in the thesis addresses each of these research questions as follows:

- **General Search Technique:** What is the best search technique that can be used for program synthesis? Is genetic programming, simulated annealing, or a hybrid of them better? can we implement them in another way as hybrid methods?

  We use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system. We have implemented a synthesis tool, which uses multiple calls to the model checker NuSMV to determine the fitness for a candidate program. The candidate programs exist in two forms. The main form is a simple imperative language. This form is subject to mutation, but it is translated to a secondary form, the modeling language of NuSMV, for evaluating its fitness. All choices of how exactly a program is represented and how exactly the fitness is evaluated are disputable. Generic search techniques are, however, usually rather robust against changes in such details. While there has been further research on how to measure partial satisfaction, we believe that the best choice for us is to keep to the choices made for promoting genetic programming, as this is the only choice that is completely free of suspicion of being selected for being more suitable for simulated annealing than for genetic programming. A second motivation for this selection is that it results in very simple specifications and, therefore, in a fast evaluation of the fitness. Noting that synthesis entails on average hundreds of thousands to millions of calls to a model checker, only simple evaluations can be considered. We have implemented six different combinations of selection and update mechanism to test our hypothesis: besides simulated annealing, we have used genetic programming both without crossover and with crossover. The tests we have run confirmed that simulated annealing performs significantly better than genetic programming. As a side result, we found that the assumption of the authors of [KP08, KP09a, KP09b] that crossover does not accelerate genetic

programming did not prove to be entirely correct, but the advantages we observed were minor.

- **Search Techniques Parameters:** What are the efficient parameters that can be used for genetic programming and simulated annealing? Is the application of crossover effective? How does initial population size affect the cost beside the success rate? How does the initial temperature and cooling schedule affect the results of simulated annealing? Are there 'best' parameters for our techniques?

  Besides serving as a synthesis tool, the main intention of PranCS is to allow for a comparison of different generic search techniques. In [HS16], we have used our technique to generate correct solutions for mutual exclusion and leader election and found that simulated annealing is much (1.5 to 2 orders of magnitude) faster than genetic programming.

- **Controller Synthesis:** Can we use the same techniques in controller synthesis? Are the results similar to those of program synthesis? What about the parameters do they have the same effect?

  We have defined a class of DCS problems where deterministic strategies are sought. We adapted our algorithms to seek solutions for these problems, and conducted an experimental evaluation using a scalable instance of a DCS problem. Results proved to be consistent with our assumption that the relative performance of the algorithms would match our previous results for program synthesis. Yet, our implementations are proofs of concept, and one can think of numerous practical improvements that constitute inescapable ways to pursue investigating efficient symbolic DCS algorithms using simulated annealingFor instance, canonically representing symbolic candidate strategies using BDDs instead of syntactic trees would allow for building a cache of fitness results, and thereby avoid re-evaluating the fitness of equivalent candidate strategies.

  Similarly, implementing the algorithms in a symbolic model-checker could permit to avoid the very costly—and often performance bottleneck in case of model-checkers using BDDs—reconstruction of symbolic representations at every iteration of the algorithms. Finally, note that our search based algorithms do not require the computation of the unsafe region to produce deterministic strategies. Hence, considering the current advances in model-checking technologies, our algorithms might constitute ways to solve DCS problems on some classes of infinite-state systems for which the unsafe region cannot be computed.

- **Synthesis Tool:** What is the scope of a synthesis tool based on generic search technique?

  We have developed PranCS, which is a tool for protocol and controller synthesis. It exploits general search techniques such as simulated annealing and genetic programming for homing in on an implementation, and use model checking for the

fitness function. We use NuSMV as a back-end for the individual model checking tasks and a simple candidate mutator to drive the search. Our Proctocol and Controller Synthesis (PranCS) tool is designed to explore the parameter space of different synthesis techniques. Besides using it to synthesise a discrete control strategy for reactive systems (controller synthesis) and for protocol adapters for the coordination of different threads (protocol synthesis), we can also use it to study the influence of turning various screws in the synthesis process. For simulated annealing, PranCS allows the user to define the behaviour of the cooling schedule. For genetic programming, the user can select the population size. Additionally, PranCS offers different automated techniques to quantify partial compliance with a specification and to infer the fitness of a candidate implementation based on this partial compliance. PranCS also allows for choosing small random seeds as well as user defined seeds to allow the user to define a starting point for the search, e.g. for candidate repair.

The main contributions of the research presented in this thesis were presented in Chapter 1. These are restated here, for completeness, as follows:

7.2.1 Program synthesis

In this part of our work we suggest to:

1. Use simulated annealing for program synthesis and compare it to similar approaches based on genetic programming.

2. Use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system.

3. We have implemented a synthesis tool, which uses multiple calls to the model checker NuSMV [CCG+02] to determine the fitness of a candidate program.

4. We have implemented six different combinations of selection and update mechanism to test our hypothesis: besides simulated annealing, we have used genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover and Hybrid genetic programming both without crossover (as discussed in [KP08, KP09a, KP09b]) and with crossover.

The tests we have run confirmed that simulated annealing performs significantly better than genetic programming. As a side result, we found that the assumption of the authors of [KP08, KP09a, KP09b] that crossover does not accelerate genetic programming did not prove to be entirely correct, but the advantages we observed were minor.

7.2.2 Controller synthesis

1. We first define a symbolic model and an associated class of DCS problems, for which deterministic strategies are sought.

2. Next, we adapt the aforementioned search techniques to obtain algorithmic solutions that avoid computing the unsafe portion of the state-space.

3. Then, we confirm the hypotheses that: (i) general search techniques are as applicable to solve our DCS problem as they are for synthesising programs; and (ii) one obtains similar relative performance results for our DCS problem. experimental results [HS16] for program synthesis , essentially that simulated annealing performs better than genetic programming.

4. To assess these hypotheses, we adapt the six different combinations of candidate selection and update mechanisms of our previous work [HS16], and execute them on a scalable example DCS problem.

5. perform an experimental feasibility assessment.

From the performance results we obtain, we draw the conclusion that , even though for technical reasons our current experimental results do not compare favourably with existing symbolic DCS tools, simulated annealing, when combined with efficient model-checking techniques, is worth further investigating to solve symbolic DCS problems.

As noted in Section 4.7, our experimental results do not compare favourably with existing symbolic DCS tools.

Bibliography

[ACMR03] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Eric Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*, pages 174–188, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[AHM+98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 521–525, London, UK, UK, 1998. Springer-Verlag.

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. pages 428–439, 1990.

[BM14] Nicolas Berthier and Hervé Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *12th Int. Workshop on Discrete Event Systems*, WODES '14, pages 46–53. IFAC, May 2014.

[BMM13] Nicolas Berthier, Florence Maraninchi, and Laurent Mounier. Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems. *ACM Trans. Embed. Comput. Syst.*, 12(1s):39:1–39:26, March 2013.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando

Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *In CAV*, LNCS 2404, pages 359–364. Springer, 2002.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CJ01] John A. Clark and Jeremy L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43:891–904, 2001.

[CKN98] José ER Cury, Bruce H Krogh, and Toshihiko Niinomi. Synthesis of supervisory controllers for hybrid systems based on approximating automata. *IEEE Transactions on Automatic Control*, 43(4):564–568, 1998.

[Con90] D.T. Connolly. An improved annealing scheme for the qap. In *European J. of Operational Research*, volume 46, pages 93 –100, 1990.

[Dav87] Lawrence Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers Inc., 1987.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, page 569, 1965.

[DKL15] Cristina David, Daniel Kroening, and Matt Lewis. Using program synthesis for program analysis. In *LPAR*, volume 9450 of *LNCS*, pages 483–498. Springer, 2015.

[Ehl11] Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, pages 272–275, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for ltl realizability. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 263–277, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[FPS15] John Fearnley, Doron Peled, and Sven Schewe. Synthesis of succinct systems. *Journal of Computer and System Sciences*, 81(7):1171–1193, 2015.

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS*, pages 321–330. IEEE Computer Society Press, 2005.

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[GR09] Alain Girault and Éric Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190, 2009.

[HBS17] Idress Husien, Nicolas Berthier, and Sven Schewe. A hot method for synthesising cool controllers. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 122–131, New York, NY, USA, 2017. ACM.

[HJJ03] Darrall Henderson, Sheldon H Jacobson, and Alan W Johnson. The theory and practice of simulated annealing. In *Handbook of metaheuristics*, pages 287–319. Springer, 2003.

[HO13] Thomas A. Henzinger and Jan Otop. From model checking to model measuring. In *CONCUR*, volume 8052 of *LNCS*, pages 273–287. Springer, 2013.

[HRL08] Yann Hietter, Jean-Marc Roussel, and Jean-Jacques Lesage. Algebraic Synthesis of Transition Conditions of a State Model. In *9th International Workshop on Discrete Event Systems*, WODES '08, pages 187–192. IEEE, May 2008.

[HS16] Idress Husien and Sven Schewe. Program generation using simulated annealing and model checking. In *SEFM*, volume 9763 of *LNCS*, pages 155–171, 2016.

[HSB17] Idress Husien, Sven Schewe, and Nicolas Berthier. Prancs: A protocol and discrete controller synthesis tool. In *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*, pages 337–349, 2017.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, volume 3576 of *LNCS*, pages 226–238. Springer, 2005.

[Joh07] Colin G. Johnson. Genetic programming with fitness based on model checking. In *EuroGP*, volume 4445 of *LNCS*, pages 114–124. Springer, 2007.

[KH91] Bruce H Krogh and Lawrence E Holloway. Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4):641–651, 1991.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[KP08] Gal Katz and Doron Peled. Model checking-based genetic programming with an application to mutual exclusion. In *TACAS*, volume 4963 of *LNCS*, pages 141–156. Springer, 2008.

[KP09a] Gal Katz and Doron Peled. Model checking driven heuristic search for correct programs. In *MoChArt*, volume 5348 of *LNCS*, pages 122–131. Springer, 2009.

[KP09b] Gal Katz and Doron Peled. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *HVC*, volume 6405 of *LNCS*, pages 117–132. Springer, 2009.

[KV99] Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, June 1999.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *LICS*, pages 389–398. IEEE Computer Society Press, 2001.

[LMST96] Jussi Lahtinen, Petri Myllymäki, Tomi Silander, and Henry Tirri. Empirical comparison of stochastic algorithms. In *2NWGA*, pages 45–60, 1996.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107. ACM, 1985.

[MBLL00] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[MS96] J.W. Mann and G.D. Smith. A comparison of heuristics for telecommunications traffic routing. In *Modern Heuristic Search Methods*, pages 235–254, 1996.

[MS00] Hervé Marchand and Mazen Samaan. Incremental Design of a Power Transformer Station Controller Using a Controller Synthesis Methodology. *IEEE Trans. Softw. Eng.*, 26:729–741, August 2000.

[MT01] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *ICALP*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society Press, 1977.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society Press, 1990.

[RCK+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, New York, NY, USA, 2009. ACM.

[RW89] Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[SF06] Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *LOPSTR 2006*, volume 4407 of *LNCS*, pages 127–142. Springer, 2006.

[Sol13] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[vEJ13] Christian von Essen and Barbara Jobstmann. Program repair without regret. In *CAV*, volume 8044 of *LNCS*, pages 896–911. Springer, 2013.

[WLK+09] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 252–263, New York, NY, USA, 2009. ACM.

[ZD12] MengChu Zhou and Frank DiCesare. *Petri net synthesis for discrete event control of manufacturing systems*, volume 204. Springer Science & Business Media, 2012.