A Hot Method for Synthesising Cool Controllers

Idress Husien
Department of Computer Science
University of Liverpool, UK
idress.husien@liverpool.ac.uk

Nicolas Berthier
Department of Computer Science
University of Liverpool, UK
nicolas.berthier@liverpool.ac.uk

Sven Schewe
Department of Computer Science
University of Liverpool, UK
sven.schewe@liverpool.ac.uk

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 *w.r.t.* the desired objectives. Previous performance results advocated the use of simulated annealing over genetic programming for such problems. In this paper, 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, and give a complexity analysis of our simulated annealing algorithm for symbolic DCS.

KEYWORDS

General Search Techniques, Symbolic Model-checking, Discrete Controller Synthesis

ACM Reference format:
Idress Husien, Nicolas Berthier, and Sven Schewe. 2016. A Hot Method for Synthesising Cool Controllers. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 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 [2, 26]. 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 [22, 26, 29], DCS algorithms have also successfully been used to enforce deadlock avoidance in multi-threaded programs [28], enforce fault-tolerance [13], or for global resource management in embedded systems [1, 3]. A closely related algorithm was also applied by Ryzhyk et al. [27] for device driver synthesis.

Foundations of DCS and program synthesis are similar to principles of model-checking [6, 10], 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 [2, 4, 11, 23].

General Search Techniques. Clark and Jacob [8], Henderson et al. [14], Johnson [17], Katz and Peled [18, 19, 20], as well as Husien and Schewe [16] 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* [21] and *simulated annealing* [8, 14]. 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 *w.r.t.* the desired objectives (*e.g.*, using a model-checker to measure the share of objectives satisfied by the candidate program, as done by Katz and Peled [18, 19, 20] and Husien and Schewe [16]). The genetic programming algorithm maintains a population of candidate programs 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.

By investigating and comparing these search techniques for program synthesis using their proof of concept implementation, Husien and Schewe [16] found that simulated annealing performs significantly better than genetic programming for synthesising programs.

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 problem.

To assess these hypotheses, we adapt the six different combinations of candidate selection and update mechanisms of our previous work [16], and execute them on a scalable example DCS problem. From the performance results we obtain, we draw the

Conference'17, Washington, DC, USA
2016. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

conclusion that simulated annealing, when combined with efficient model-checking techniques, is worth further investigating to solve symbolic DCS problems.

Outline of the Paper. We formally define the symbolic model and DCS problems, and detail the particular kind of solutions we seek in Section 2. We then turn to a description of the general search techniques that we investigate in Section 3, and further detail how we adapted them for solving our symbolic DCS problems in Section 4. We detail our experiments and give a complexity analysis of our simulated annealing algorithm in Sections 5 and 6. We eventually summarise and discuss our results in Section 7.

2 SYMBOLIC MODEL-CHECKING & CONTROLLER SYNTHESIS

2.1 Predicates

We denote by $V = \langle v_1, \dots, v_n \rangle$ 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: $v = \{v_1 \mapsto \mathsf{ff}, \dots, v_n \mapsto \mathsf{tt}\}$. 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* 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. 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 $\neg\varphi \vee \varphi'$.)

2.2 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 variables according to the current values of the internal and input variables.

Definition 2.1 (Symbolic Transition System). A *symbolic transition system* is a tuple $S = \langle X, I, T, A, x^0 \rangle$ where:

- $X = \langle x_1, \dots, x_n \rangle$ is a vector of state variables encoding the memory necessary for describing the system's behaviour;
- $I = \langle i_1, \dots, i_m \rangle$ is a vector of input variables;
- $T = \langle T_1 \colon \mathcal{P}_{X \cup I}, \dots, T_n \colon \mathcal{P}_{X \cup I} \rangle$ 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.

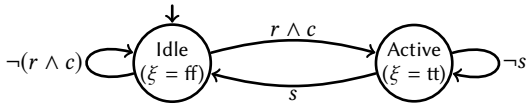

Figure 1: STS S_{Task} (Example 2.3) as a guarded automaton.

We give an illustrative STS in Example 2.3 below.

Remark 1 (Parallel Composition). Consider given the set of STSs $S_1, \dots, S_N$, 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* $S_1 \| \dots \| S_N$: *i.e.,*

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

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

Definition 2.2 (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 space of $[S]$;
- $\mathcal{T} \colon \mathcal{X} \times \mathcal{I} \to \mathcal{X} = \lambda(x, \iota). \left(T_j(x, \iota)\right)_{j \in \{1,\dots,n\}}$;
- $\mathcal{A} \subseteq \mathcal{X} \times \mathcal{I} = \{(x, \iota) \in \mathcal{X} \times \mathcal{I} \mid (x, \iota) \models A\}$;
- 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 $(x^j, \iota^j) \in \mathcal{A}$). $\mathit{Suff}[S]\left(x^0\right)$ denotes the set of all such sequences, and $\mathit{XSuff}[S]\left(x^0\right)$ is the sequences of states that are obtained from $\mathit{Suff}[S]\left(x^0\right)$ by removing the input component of each tuple of the sequences.

All *execution traces* of $[S]$ start in the initial state x^0, and $\mathit{XTrace}[S] \stackrel{\text{def}}{=} \mathit{XSuff}[S]\left(x^0\right)$ 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 $[S]$: $\mathit{XPath}[S] \stackrel{\text{def}}{=} \left\{\pi_s \,\middle|\, \exists \pi_p, \pi_p \cdot \pi_s \in \mathit{XTrace}[S]\right\}$.

Example 2.3 (Task STS). Let us now give a small illustrative example STS modelling a two-state task. We build upon this example throughout the paper. We model the behaviour of the system under consideration, called "Task", using the STS $S_{\mathsf{Task}} = \langle X, I, T, A, x^0 \rangle$, 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 $x^0 = \{\xi \mapsto \mathsf{ff}\}$. An automaton representation of Task is given in Figure 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 $\mathit{XTrace}[S_{\mathsf{Task}}]$.

2.3 Model-checking STSs

Model-checking [6, 10] 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 involve Binary Decision Diagrams (BDDs) [5], 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) [25] and *Computation Tree Logic* (CTL) [9]. 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 2.1);
- $x \models \neg\phi$ iff $x \not\models \phi$;
- $x \models \phi \vee \phi'$ iff $x \models \phi$ or $x \models \phi'$;
- $x \models A\psi$ iff $\forall \pi \in XSuff[S](x), \pi \models \psi$;
- $x \models E\psi$ iff $\exists \pi \in XSuff[S](x), \pi \models \psi$.

Let $\pi = x^0 \cdot x^1 \cdot x^2 \cdots \in XPath[S]$ 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 $\mathrm{tt}\, U \phi$.)

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

2.4 Symbolic Discrete Controller Synthesis

The theoretical framework for Discrete Controller Synthesis (DCS) algorithms was first introduced by Ramadge and Wonham [26] 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 $\varphi \in \mathcal{P}_X$.

2.4.1 Principles of Traditional DCS Algorithms. The traditional approach for solving DCS problems is 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, 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.,* $x^0 \in \mathcal{F}_\phi$.

2.4.2 Symbolic DCS. Symbolic DCS algorithms targeting various models were investigated, by Asarin et al. [2], Cury et al. [11] for instance. Regarding models close to STSs, symbolic DCS algorithms and tools were developed by Marchand et al. [23], Marchand and Samaan [24], and later extended by Berthier and Marchand [4] 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 are based on a fixpoint computation of a symbolic representation of $\mathcal{F}_\phi$, 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.

Example 2.4 (Controlling S_{Task}). Building up on the STS S_{Task} introduced in Example 2.3, 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 $u^0 = \{r \mapsto \mathrm{tt}, s \mapsto \mathrm{tt}\}$, according to the strategy K_ϕ we must choose $c^0 = \{c \mapsto \mathrm{ff}\}$ 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 $c^{0\prime} = \{c \mapsto \mathrm{tt}\}$ would lead to a violation of ϕ). The controlled system S'_{Task} can then evolve into state $x^1 = \{\xi \mapsto \mathrm{ff}\}$ (staying in Idle). With further inputs $u^1 = \{r \mapsto \mathrm{tt}, s \mapsto \mathrm{ff}\}$, we can either choose $c^1 = \{c \mapsto \mathrm{ff}\}$ or $c^{1\prime} = \{c \mapsto \mathrm{tt}\}$, that both fulfil the strategy K_ϕ and respectively lead to state $x^2 = \{\xi \mapsto \mathrm{ff}\}$ and $x^{2\prime} = \{\xi \mapsto \mathrm{tt}\}$.

2.4.3 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 $\left\{ c \in \mathcal{D}_C \,\middle|\, (x, u, c) \models K_\phi \right\}$ 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 2.4); 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 paper, to obtain deterministic, easily implementable controlled STSs, we seek algorithms that give results similar to those obtained after applying the translation of option (ii).

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

A triangulation technique similar to the one described by Hietter et al. [15] 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.

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.,*

$$\Gamma_\phi = \langle \gamma_1 : \mathcal{P}_{X \cup U}, \ldots, \gamma_k : \mathcal{P}_{X \cup U} \rangle. \tag{1}$$

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 2.5 (Deterministic Controller for S'_{Task}). Consider again the controller obtained in Example 2.4. 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.

Algorithm 1 Simulated Annealing

```
i := 0
randomly generate a first candidate x
repeat
  i := i + 1
  derive a neighbor 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/θ_i) then
      x := x'
    end if
  end if
until the goal is reached or i = i_max
```

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

3 GENERAL 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.

3.1 Simulated Annealing

Simulated annealing [8, 14, 16] is a general local search technique that is able to escape from local optima. The algorithm, described in Algorithm 1, 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 [14].

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.

3.2 Genetic Programming

Genetic programming is a different general search technique [21]. It has already been used for program synthesis by Johnson [17], Katz and Peled [18, 19, 20], and Husien and Schewe [16]. 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 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.

3.3 Hybrid Search Technique

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 by Katz and Peled [18, 19, 20] as well as Husien and Schewe [16].

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

Algorithm 2 $\mathsf{grow}_{maxdepth}\,(depth)$

```
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
```

4 PRINCIPLES OF OUR DCS ALGORITHMS

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

4.1 Representing Deterministic Strategies

Recall that the candidate deterministic strategies are vectors of predicates involving state and non-controllable variables (*cf.* Section 2.5), 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 BDDs (*cf.* Section 2.3). Yet, implementing efficient random generation, mutations, and crossovers on such diagrams appears to be challenging, and more importantly, unnecessary *w.r.t.* 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 2.1:

- 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.

4.2 Random Generation of Candidates

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 Koza [21] 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.

$\mathsf{grow}_{maxdepth}\,(depth)$ is shown as a recursive function in Algorithm 2, generating trees of maximum depth *maxdepth*. It takes the depth *depth* of the current node to be generated as argument. 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 $\text{grow}_{maxdepth}(1)$ shall then be used to generate one candidate deterministic strategy Γ.

4.3 Performing Mutations and Crossovers

Mutations are changes applied on each candidate deterministic strategy Γ. 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 probabilities weighted by the number descendants);
3. 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 $\text{grow}_{maxdepth}(1)$; the latter case is illustrated in Figure 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 3 an example crossover between two trees.

4.4 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 Katz and Peled [18, 19, 20] and Husien and Schewe [16] 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, 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*ψ), and replacing one with an existential path quantifier ($E\psi$);
- 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 5.1 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 $\omega \cup \omega' \cup \omega''$ (*cf.* Section 2.3). 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''.

Following the works of Katz and Peled [18] and Husien and Schewe [16], 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.

4.5 Variants for Improved Search Techniques

To derive improved variants of the general search algorithms of Section 3, we note that a subset ω_s of given target objectives ω are *safety* ones, while the others ω_l $(= \omega \setminus \omega_s)$ 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 1) 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 $F_{\omega_s}(\Gamma') < F_{\omega_s}(\Gamma)$.

Our implementation of the hybrid algorithm presented in Section 3.3 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 $F_{\omega_s}(\Gamma)$.

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).

5 EXPERIMENTAL FEASIBILITY ASSESSMENT

5.1 Problem Instances

In order to get a preliminary feasibility assessment of our approach, we have generated several problem instances based on the parallel

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

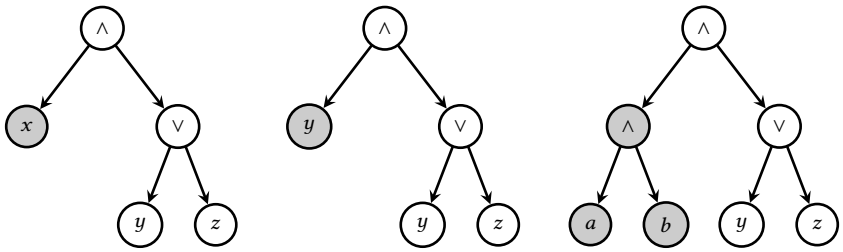

Figure 2: Candidate predicate (left) with two mutations (middle and right)

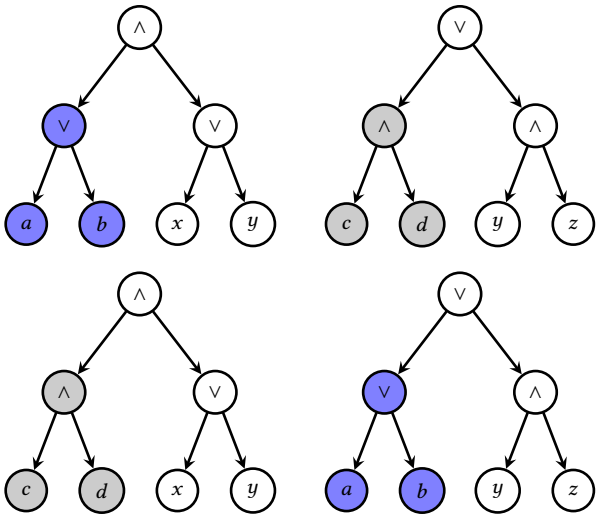

Figure 3: Crossover: two parents (above) and two offspring (below)

composition of STSs as per Remark 1. Synthesis objectives were then derived as CTL formulae in a scalable manner.

Regarding the STSs, each problem N–Tasks was built using N instances of the STS S'_{Task} described in Example 2.4 (modulo renaming of variables to ensure disjointness); *i.e.,* the STSs involved were built as

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

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

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 $\xi_i = \text{tt}$) 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:

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

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

Partial Objectives. The partial objectives automatically generated from the $\omega_{s,N}$'s above, according to the procedure explained in Section 4.4, 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

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

5.2 Experimental Setup

We have implemented the simulated annealing, and genetic programming, and hybrid algorithms as described, using NuSMV [7] 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 by Katz and Peled [18] and Husien and Schewe [16] 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 (*cf.* Section 4.4).

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

Table 1: Search Techniques Comparison

	Search Tech.	sngl exec.	suc. rat%	overall 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

5.3 Experimental Results

The results are shown in Figures 4 and 5 and summarised in Table 1. Figure 4 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 5. Table 1 shows the average running time for single executions in seconds, the success rate in %, and the resulting overall running time. 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 [16], it is interesting to note that both the pure and the hybrid approach to genetic programming benefit from crossovers.

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 [4] (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 7.

6 COMPLEXITY ANALYSIS

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.

Theorem 6.1. *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 6.1, 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

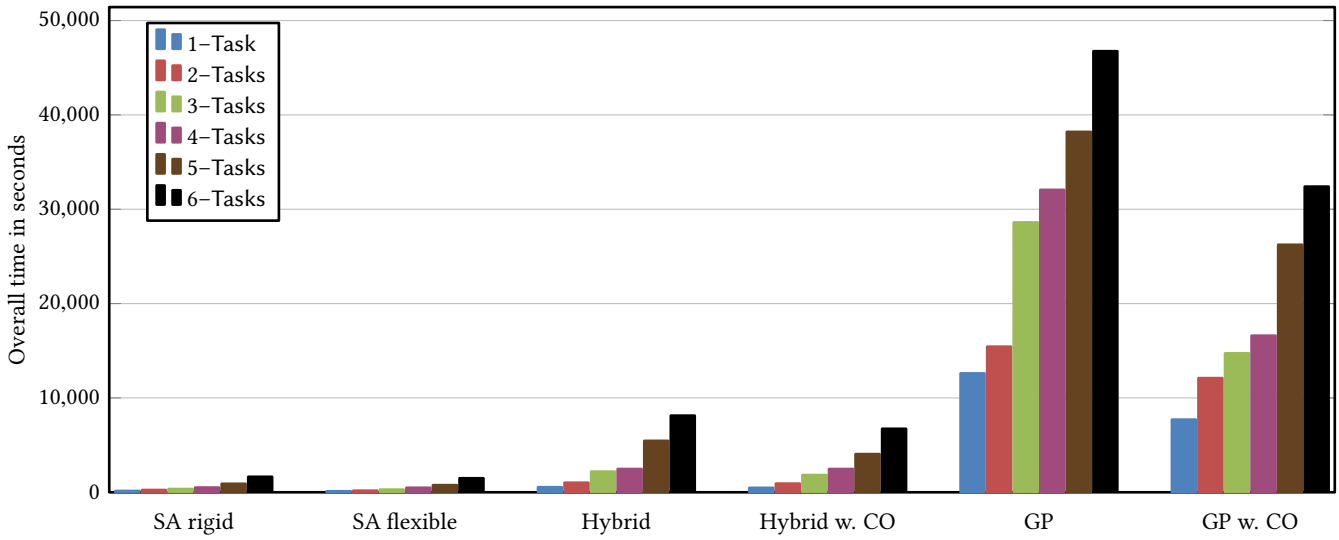

Figure 4: Overall time required for synthesising a correct candidate

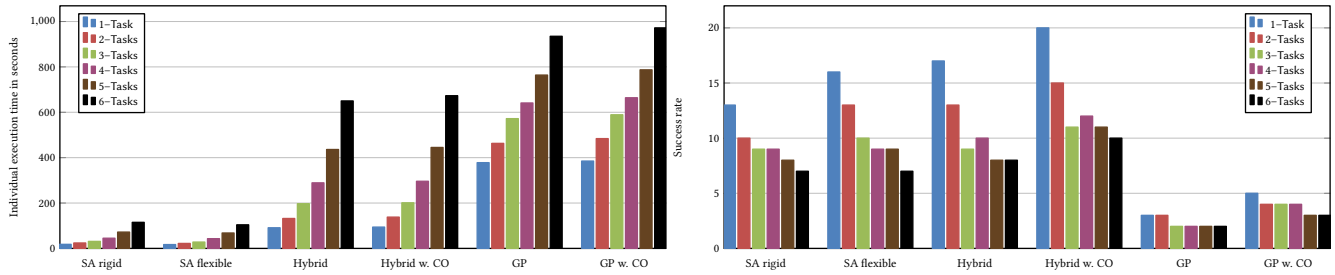

Figure 5: Average running time of an individual execution (left) and success rate of individual executions (right)

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

Proof of Theorem 6.1. 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 $m_{\text{attempts}} \in O(m)$ 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 [12].

7 CONCLUSION

Inspired by our previous investigations for program synthesis using general search techniques [16], 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.

7.1 Discussion

As noted in Section 5, 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. 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.

REFERENCES

[1] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Éric Rutten. 2003. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the 12th European Conference on Programming (ESOP'03)*. Springer-Verlag, Berlin, Heidelberg, 174–188.

[2] Eugene Asarin, Oded Maler, and Amir Pnueli. 1994. Symbolic controller synthesis for discrete and timed systems. In *International Hybrid Systems Workshop*. Springer, 1–20.

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

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

[5] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35 (1986), 677–691.

[6] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98, 2 (1992), 142–170.

[7] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*. Springer-Verlag, London, UK, UK, 359–364.

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

[9] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, London, UK, UK, 52–71.

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

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

[12] John Fearnley, Doron Peled, and Sven Schewe. 2015. Synthesis of Succinct Systems. *J. Comput. System Sci.* 81, 7 (2015), 1171–1193.

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

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

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

[16] Idress Husien and Sven Schewe. 2016. Program Generation Using Simulated Annealing and Model Checking. In *SEFM (LNCS)*, Vol. 9763. 155–171.

[17] Colin G. Johnson. 2007. Genetic Programming with Fitness Based on Model Checking. In *EuroGP (LNCS)*, Vol. 4445. Springer, 114–124.

[18] Gal Katz and Doron Peled. 2008. Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In *TACAS (LNCS)*, Vol. 4963. Springer, 141–156.

[19] Gal Katz and Doron Peled. 2009. Model Checking Driven Heuristic Search for Correct Programs. In *MoChArt (LNCS)*, Vol. 5348. Springer, 122–131.

[20] Gal Katz and Doron Peled. 2011. Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In *Proceedings of the 5th International Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC'09)*. Springer-Verlag, Berlin, Heidelberg, 117–132.

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

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

[23] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. 2000. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic System: Theory and Applications* 10, 4 (Oct. 2000), 325–346.

[24] Hervé Marchand and Mazen Samaan. 2000. Incremental Design of a Power Transformer Station Controller Using a Controller Synthesis Methodology. *IEEE Trans. Softw. Eng.* 26 (Aug. 2000), 729–741. Issue 8.

[25] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. IEEE Computer Society Press, 46–57.

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

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

[28] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. 2009. 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)*. ACM, New York, NY, USA, 252–263.

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