Efficient Constrained Random Sampling for use in a Model Based Testing Tool

Author: Martin Huijben
s4205332

Supervisors:
Msc. R. Janssen
ramonjanssen@cs.ru.nl
Dr. ir. G.J. Tretmans
tretmans@cs.ru.nl

Second reader:
Dr. N.H. Jansen
nils.jansen@ru.nl
Abstract

Constrained Random Sampling (CRS) is the act of choosing a solution for a problem, which is defined as a set of variables and a constraint, in such a way that each solution has about equal probability of getting chosen. CRS algorithms are commonly used in testing tools for generating random input when testing. Such a testing tool is TorXakis, which is model based, using a model specified as a labeled transition system. The goal of this thesis is to find a CRS algorithm that finds solutions faster and with a more equal probability of getting chosen per solution. I do this by listing expectations of the CRS algorithm within TorXakis, giving an overview of algorithms in the field of CRS, and finally introducing a new algorithm specialized for TorXakis.

1. Introduction

Constrained Random Sampling (CRS) is the act of choosing a solution for a problem in such a way that each solution has about equal probability of getting chosen. Here a problem is defined as a set of variables and a constraint. The main application of CRS algorithms is in the area of software and hardware testing, where they are used for input generation. The goal is then to find an input to which the software or hardware reacts unexpectedly.

Testing a SUT (System Under Test) is standardly done according to the following pattern: given an input fulfilling a certain condition, the SUT should react in a certain measurable way. Often the amount of inputs that fulfill that condition is too large for them all to be tested. Instead, we want to test with a random subset of all possible input.

TorXakis[1] is a tool for model based testing. The model used in TorXakis is specified as a labelled transition system. A state in the model has zero or more transitions to other states, each accompanied with a label. The model transits to another state either when giving input to the SUT, or receiving output from the SUT. In this thesis we are only interested in the transitions that give input.

When giving input, the label of the transition is a problem description, defined as a set of variables and a constraint. It is the goal of the CRS algorithm within TorXakis to find a random solution to this problem.

The model is allowed to be infinite, in such a way that every transition is different from every other. The model is also allowed to use information returned by the SUT in the constraints of following transitions, meaning it is possible to make models where constraints are different each testing session.

Suppose our model was finite and constraints known before testing, then we could generate input before running. The time it takes for a CRS algorithm to generate input would then matter far less. However, since this is not the case, the desired CRS algorithm should generate input in a timely manner. The current CRS algorithms within TorXakis are very fast. They buy this speed however by sacrificing uniformity, giving some solutions a very high probability of getting chosen.

This paper is organized as follows.

Chapters 2 and 3 set the foundation of this thesis, on which we can judge CRS algorithms introduced in later chapters. Chapter 2 contains preliminaries. Chapter 3 lists expectations over the desired CRS algorithm.

Chapters 4 through 6 judge existing CRS algorithms. Chapter 4 tests the four CRS algorithms currently used in TorXakis. Chapter 5 gives an overview of existing CRS algorithms. Chapter 6 compares the algorithms to the expectations listed in chapter 3, and concludes a better algorithm can be made.

Chapters 7 through 9 show a new algorithm specialized to the expectations, as they are...
described in chapter 3. In chapter 7 I describe a new algorithm that uniformly and efficiently finds a solution. In chapter 8 I extend that algorithm with space reduction tactics. In chapter 9 I describe the necessary semantic additions to embed this algorithm in TorXakis in a way that fulfills the expectations I set.

**Chapters 10 and 11 finalize this thesis.** In chapter 10 I conclude and in chapter 11 I describe possible future research.

1. Introduction......................................................................................................................................2
2. Preliminaries.......................................................................................................................................4
3. Expectations for TorXakis.................................................................................................................9
4. Testing the TorXakis algorithms........................................................................................................15
5. An overview of CRS algorithms......................................................................................................28
6. Comparison of CRS algorithms......................................................................................................30
7. A new algorithm...............................................................................................................................32
8. Space reduction tactics......................................................................................................................35
9. Semantic additions to the TorXakis specification language...........................................................50
10. Conclusion........................................................................................................................................54
11. Future research..................................................................................................................................55
12. References.......................................................................................................................................59
2. Preliminaries

Before I can talk about CRS algorithms, we first need some definitions. It must be noted that most of these definitions are not standardized or even defined in other papers.

**Problem description**

CRS algorithms should solve a problem, that can be defined as follows. There is a finite set $\mathcal{V}$ of variables. Each variable $v \in \mathcal{V}$ has a domain $\mathcal{D}_v$. An assignment assigns to every variable a value from its domain. There is a predicate on assignments, the *constraint*. The assignment space $\mathcal{A}$ contains all total assignments. An assignment is always total, unless explicitly specified otherwise. The solution space $\mathcal{S}$ contains all assignments that are valid according to the constraint, also called *solutions*.

The goal of a CRS algorithm is to return a solution in such a way that all solution have a close to equal probability of getting returned.

**Partial assignment**

An assignment $a$ is partial (as opposed to total) if not every variable is assigned a value. $a_\bot$ is the undefined assignment, where no variable is assigned a value.

A partial assignment $f$ is more defined than a partial assignment $g$, notated as $f \sqsupset g$, if $\mathcal{D}_g \subset \mathcal{D}_f$ and $\forall v \in \mathcal{V}, g(v) \neq \bot \rightarrow f(v) = g(v)$. In other words, $f$ is the same as $g$ except for the fact that some variables that were assigned no value, are now assigned a value.

A partial solution $ps$ is a partial assignment, such that there exists an $s \in \mathcal{S}$, $s \sqsupset ps$.

**Domains**

Common variable domains are finite and infinite integer ranges, booleans, and Algebraic Data Types (ADTs). ADTs are composite types: they are created by combining other types. An ADT has one or more *constructors*. Each constructor has zero or more *fields*. Each field has a domain. Values of an ADT have one constructor, and have a value from the domains of each of the fields belonging to that constructor. Let us look at some examples. The syntax shown here is the syntax of the TorXakis Specification language:

```
Weekday ::= Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

This ADT only contains constructors with no fields. Such an ADT is an enumeration.

Angle ::= Radial { r:: Int} | Degrees {d::Int}

This ADT contains two constructors, each with one field. Since this ADT does not call itself, it is called non-recursive. Since this ADT has fields, it is called complex.

IntList ::= Nil | Cons { hd :: Int; tl :: IntList }

Lists do not exist as primitives in TorXakis. Instead, one can define lists using ADTs. This is a recursive ADT.

Just like ADTs, constructors too can be divided into the subcategories enumeration versus complex, non-recursive versus recursive, based on whether they have a fields, and whether one of those fields is of the same type as the ADT using the constructor.
Notation of a problem

On the right are two examples of problems as notated throughout this thesis. The first one shows a name, two variables and their domains, and a constraint. In this problem \texttt{add}, \(x\) and \(y\) have as domain a finite int range defined as \(\{i \in \mathbb{Z} \mid i \geq 0 \land i < 100\}\). In this example, \(\{(x, 20), (y, 58)\}\) is an assignment, but not a solution. \(\{(x, 20), (y, \bot)\}\) is a partial solution.

In the second problem \texttt{list}, \(x\) has as domain an ADT. Both constructors and fields are named. The ADT list has the constructors \texttt{Nil} and \texttt{Cons}. \texttt{Nil} has no fields, \texttt{Cons} has two, named \texttt{hd} and \texttt{tl}. The domain of \texttt{hd} is a finite integer range, the domain of \texttt{tl} is once again a list.

The constraint specifies that the list should start with a \texttt{Cons} (it should have at least one element, and its first element should be larger than 50. To specify this, two functions are used, \texttt{isCons} and \texttt{hd}. Such functions, that respectively return true when the given ADT value uses a certain constructor, or that return the field domain belonging to a certain field name, are assumed to exist and be named appropriately for every ADT.

An example of a solution to this problem is \texttt{Cons(51, Cons(2, Nil))}.

Dense and sparse

A space is dense if almost all assignments are solutions.

A space is sparse if almost no assignments are solutions.

Partition

A partition of a set is a grouping of the set's elements into non-empty subsets, such that every element is included in one and only one subset. We will call such a subset a part.

A common partition of a variable is division in ranges. All parts are then ranges.

SAT solvers and SMT solvers

SAT solvers and SMT solvers try to answer satisfiability questions: when given a problem, they answer the question \(\exists a \in A, valid(a)\). Using a large number of optimizations, they can do so very efficiently.

SAT solvers solve satisfiability questions in propositional logic. SMT solvers solve satisfiability questions in first order logic, with addition of theories such as linear arithmetic and functions.

The most used SAT solvers in CRS algorithms are CryptoMiniSAT[8], which is good at handling constraints using XORs, and WalkSAT[9], which gives semi-random solutions. The most used SMT solver is Z3[10].

Specification language

The variables and constraint of a problem should be specified in a certain language. Not all CRS algorithms expect the same language. A number of languages can be discerned. The most well-known language is propositional logic, or logical formulas with boolean literals. If \(B\) is a set of boolean variables, it is defined as:

\[
\begin{align*}
\text{add} \\
\begin{array}{l}
x : [0,100] \\
y : [0,100] \\
x + y \leq 50
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{list} \\
\begin{array}{l}
x : \text{list} \\
\text{list} = \text{Nil} | \\
\text{Cons}(hd = [0,100], tl = \text{list}) \\
\text{isCons(list)} \land \text{hd(list)} > 50
\end{array}
\end{align*}
\]
A language that also uses boolean literals is Conjunctive Normal Form (CNF) with boolean literals, defined as:

\[
\begin{align*}
\text{constraint} & : \quad \text{constraint op constraint} \mid \text{literal} \\
\text{op} & : \quad \land \mid \lor \mid \rightarrow \mid \leftrightarrow \mid \neg \mid \bigotimes \\
\text{literal} & : \quad B_i \mid \neg B_i \mid \text{True} \mid \text{False}
\end{align*}
\]

Other used literals than boolean literals are expression relation literals. If we take \( I \) to mean the set of int variables, they are defined respectively as:

\[
\begin{align*}
\text{literal} & : \quad \text{True} \mid \text{False} \mid \text{expression relop expression} \\
\text{relop} & : \quad \leq \mid \geq \mid > \mid < \mid = \mid \neq \\
\text{expression} & : \quad I_i \mid i \in \mathbb{N} \mid \text{expression arithop expression} \\
\text{arithop} & : \quad + \mid - \mid \div \mid *
\end{align*}
\]

Then there are specialized languages. One of them is the TorXakis specification language. This language adds ADTs to be used as variable types, and function definitions to be used in constraints. Other specialized languages are specialized processor languages. Translating to or from them cannot be done easily. They can include ways to describe processor buses, processor instructions, etc.

And lastly, there is SMT2, the language standardly used by SMT solvers. This language includes a huge scala of theories, among which ADTs and function definitions. This means that the TorXakis language (aside from syntactical differences) can be seen as a subset of SMT2.

Translation between languages is not trivial. Translating from a higher or more compact language to a lower or less compact language can exponentially increase the representation size. The most compact languages are SMT2 and the specialized languages, followed by logical formulas with expression literals. Logical formulas (independent of literals) are more compact than CNF with the same literals. Expression relation literals are more compact than boolean literals.

When an algorithm uses a different language than the TorXakis specification language, a translation from the TorXakis specification language to that other languages is needed. In chapter 8 we will look at such a translation.

**Rejection sampling, iterating over the assignment space, and iterating over the solution space**

There are three language-agnostic CRS algorithms, on which all other CRS algorithms are based. They are rejection sampling, iterating over the assignment space, and iterating over the solution space. Rejection sampling tries random assignments until it finds a solution. Iterating over the assignment space iterates over all assignments until it has a set of all solutions, and then chooses uniformly from this set. Iterating over the solution space uses a solver (SAT or SMT) to find a set of all solutions, and then chooses uniformly from this set. The only thing one needs is respectively either a way to generate random assignments, a way to iterate over all assignments, or a solver.
When the assignment space is known to be dense and only one solution is needed, rejection sampling is fastest.

When the assignment space is known to be sparse and a lot solutions are needed, iterating is fastest.

If the constraint can be solved easily by the solver, iterating over the solution space is fastest, otherwise iterating over the assignment space.

Other algorithms extend such a base algorithm with discarding parts of the assignment space, based on assumptions or knowledge.

**Xor hash**

An xor hash can be used to divide a assignment space in two parts, if the problem is expressed using boolean literals. It is a constraint that excludes half of the assignment space, when conjuncted to the problem constraint. The constraint is made by xorring a constant and every variable or its negation.

The constant and the choice per variable whether to use the variable or its negation are chosen randomly.

For example, say we have three boolean variables, \(A\), \(B\) and \(C\). And our random xor hash is:

\[0 \oplus A \oplus \neg B \oplus \neg C\]

Then in the following table all assignments followed by 1 are valid:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>\neg C</td>
<td>0</td>
</tr>
<tr>
<td>\neg A</td>
<td>B</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>\neg A</td>
<td>\neg B</td>
<td>\neg C</td>
<td>1</td>
</tr>
<tr>
<td>\neg A</td>
<td>\neg B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>\neg A</td>
<td>\neg B</td>
<td>\neg C</td>
<td>0</td>
</tr>
</tbody>
</table>

Xor hashing does not create perfectly random partitions. For instance, assignments that differ only one value will never be in the same part, as is clearly visible for the assignments that only differ in the assignment of \(C\). Nonetheless, they are standardly considered random enough\[2\]. For this reason, and for how easy it is to make a xor hash, xor hashes are used in a number of CRS algorithms. The only downside is the necessary translation to a boolean formula.

**Covering CRS algorithms**

A covering CRS algorithm gives each solution a probability higher than zero of getting returned. The opposite of a covering CRS algorithm is a non-covering CRS algorithm. As an example, if our problem exists out of one variable \(x\) with domain \([0, 10)\) and no constraint, and a given CRS algorithm will in no circumstance output \(x = 3\), then it is non-covering.

**Neighbours**

Neighbours of an assignment differ at exactly one value. For example the assignments \((True, True, 5)\) and \((True, True, 11)\) are neighbours. This term only applies to assignments using non-complex, non-recursive variables.

**Pseudocode**

Throughout this thesis pieces of pseudocode will be shown. Following standard notation, a \(\leftarrow\) denotes an assignment. For sake of readability and keeping descriptions terse, I extend the notation
so that $\leftarrow\uplus$ assigns to the left hand side the union of both sides, and that $\leftarrow\downarrow$ inserts the right hand side to the set on the left hand side. Furthermore, the datatype `dict()` will be used. This is a dictionary: given a key, it returns a value. If a key is not known in the dictionary, it will return $\bot$. There is a function `keys(d)`, that returns all keys known to its argument dictionary.
3. Expectations for TorXakis

This chapter lists the expectations of TorXakis as a testing tool and, by extension, the expectations on the CRS algorithm to be used in TorXakis. This is done in two parts. The first part lists common problems given to TorXakis. The second part lists general expected requirements of TorXakis, such as requirements on uniformity and control over the generated solutions. The problems of the first part will be used as a test set to judge CRS algorithms on. For each requirement in the second part, I will refer to the chapters influenced by it.

Typical problems

TorXakis models typically model large runs of a SUT. These are split up in many small and simple transitions, together creating a complicated model. It is for this reason that many problems that the CRS algorithm in TorXakis needs to solve are quite simple.

Often occurring problems have either no constraints over a variable with finite type, constrain the solution space to only one solution, or ask for an integer to be within a certain range. I show for each of these cases in order an example.

<table>
<thead>
<tr>
<th>no constraint</th>
<th>equality</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x : \text{Boolean} )</td>
<td>( x : \text{Integer} )</td>
<td>( x : \text{Integer} )</td>
</tr>
<tr>
<td>( x = 5 )</td>
<td>( 20 \leq x &lt; 100 )</td>
<td></td>
</tr>
</tbody>
</table>

A typical problem has variables that are dependent on another variable; they describe properties of that variable. An example of this is properties. Here \( l \) is a list of Booleans, and there are three other variables, describing respectively its length, whether all elements are True, and whether the first element is True.

\[
\begin{align*}
\text{properties} & \quad \text{boollist} \\
l : & \quad \text{boolist} \\
length : & \quad \text{Integer} \\
all : & \quad \text{Boolean} \\
empty : & \quad \text{Boolean} \\
\text{boollist} & = \begin{cases} Nil \\ \text{Cons}(hd = \text{Boolean}, tl = \text{boollist}) \end{cases} \\
length(l) & : \begin{cases} 1 + length(tl(l)) & \text{if } \text{isCons}(l) \\ 0 & \text{else} \end{cases} \\
all(l) & : \begin{cases} hd(l) & \text{if } \text{isCons}(l) \\ all(tl(l)) & \text{else} \text{True} \end{cases} \\
length & = length(l) \\
all & = all(l) \\
empty & = isNil(l)
\end{align*}
\]

In the problem add, when we choose a value for \( x \), the amount of possible values we can take for \( y \) to still end up with a solution is dependent on the value we chose for \( x \). Such constraints, where the valid values of one value are dependent on which value is chosen for another, do occur, but not very often in comparison to other constraints.
Lists are used often, especially compared to other recursive ADTs. I have seen no recursive ADTs to model trees, but does not mean nobody uses them. Both ADTs are shown in list and tree.

One interesting problem using a list wants all elements of the list to take a certain value, except for the last, which should be in a certain range. An example of this is ranged last.

Constraints reasoning with the lengths of lists happen often. An example of this is length.
Users use nested complex ADTs to structure their data. An example is `struct`.

<table>
<thead>
<tr>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) : person</td>
</tr>
<tr>
<td>person = ( \text{Person}(\text{name} = \text{String}, \text{age} = [0,100], \text{favorites} = \text{favorites}) )</td>
</tr>
<tr>
<td>favorites = ( \text{Favorites}(\text{food} = \text{food}, \text{drink} = \text{drink}) )</td>
</tr>
<tr>
<td>food = ( \text{Pizza}</td>
</tr>
<tr>
<td>drink = ( \text{Coffee}</td>
</tr>
</tbody>
</table>

A specific interesting problem asks to generate a large number of integers. Integer values should be within a certain constant range, and should also differ maximally only a certain constant amount. Three small examples of this problem are 2snake, 3snake and 4snake.

<table>
<thead>
<tr>
<th>2snake</th>
<th>3snake</th>
<th>4snake</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a ) : ([0,100])</td>
<td>( a ) : ([0,100])</td>
<td>( a ) : ([0,100])</td>
</tr>
<tr>
<td>( b ) : ([0,100])</td>
<td>( b ) : ([0,100])</td>
<td>( b ) : ([0,100])</td>
</tr>
<tr>
<td>( a \leq b + 5 \land a \geq b - 5 )</td>
<td>( a \leq b + 5 \land a \geq b - 5 )</td>
<td>( a \leq b + 5 \land a \geq b - 5 )</td>
</tr>
<tr>
<td>( b \leq c + 5 \land b \geq c - 5 )</td>
<td>( b \leq c + 5 \land b \geq c - 5 )</td>
<td>( b \leq c + 5 \land b \geq c - 5 )</td>
</tr>
</tbody>
</table>

Another interesting problem asks to generate coordinates within a circle.

<table>
<thead>
<tr>
<th>circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) : ([-8,9])</td>
</tr>
<tr>
<td>( y ) : ([-8,9])</td>
</tr>
<tr>
<td>( x^2 + y^2 \leq 8^2 )</td>
</tr>
</tbody>
</table>
**Test tool expectations**

**Expectations on coverage confidence**

The goal of testing is to get a certain degree of confidence that the SUT is correct. This is done by performing multiple test runs. If some runs are done multiple times, then that is a waste, since no new information will be gained and the confidence will not rise. Especially in the case where the SUT takes a long time to process the input of the CRS algorithm, a slower CRS algorithm that performs same test runs less often, is much more preferable. However, even in the case where half of the test runs is a test run that has been done already, this is only a slowdown of a factor 2.

A worst case is the following: the CRS algorithm is given a problem with only a small amount of solutions out of which it is supposed to choose uniformly. Instead, it gives one value a much lower probability than uniform, choosing it effectively never. This would give false confidence, since one path is then never looked at. Notice that in standard test environments, the amount of solutions is typically much larger than the amount of tests done over the environment, so such a solution that never gets chosen matters little, since most solutions will never get chosen. However, TorXakis tests using a transition system of multiple problems. Such problems *can* have only a few solutions, and if the CRS algorithm then ignores one of those solutions, it can cause TorXakis to never visit certain transition paths.

For this reason I set a lower bound on the probability of the least likely solution of a problem to get chosen. This probability should be not be lower than half the probability of that solution if the CRS algorithm would choose uniformly. Let us define a metric that measures this. This metric is 1 if the CRS algorithm chooses uniformly. It rises when the probability of the least likely solution to get chosen gets lower. Let us call this metric Lower Coverage Bound, or LCB, and it is calculated as:

\[
LCB = \frac{1}{\text{odds}(s) \times |S|}
\]

Where \(s\) is the least likely solution and \(S\) the set of solutions. We assume at least one solution when given a problem. When the algorithm is non-covering, the LCB is \(\infty\).

The LCB should not exceed 2. This demand should hold for *all possible problems*. That seems like an overly strict demand, as opposed to demanding it should hold for *problems that occur often*, such as the problems defined in the above section. However, since there is no efficient way to check if a problem will have a high LCB, we cannot warn users in the case that their tests do not cover the test environment as well as they would expect, thereby giving them false confidence.

Testing is standardly the last phase in a development cycle. It is the safety net catching the final mistakes before shipping. For this reason, we rather have *low confidence* than *false confidence*. The demand that the LCB should be \(\leq 2\) will be used to judge CRS algorithms on in the coming chapters.

**Expectations on uniformity over partial solutions**

Our desired CRS algorithm should choose uniformly from all solutions (even though, as we just read, it is allowed to make concessions within bounds). But in certain cases, uniformity is not desired. Uniformity over partial solutions is desired instead. For example, say that in the problem **Decision** the variable \(a\) causes an important decision within the SUT, while the value of \(x\) is considered less important. Then we want uniformity over the partial solutions in which \(a\) has been assigned a value, instead of over the complete set of solutions. In other words, we want

<table>
<thead>
<tr>
<th>Decision</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Boolean</td>
</tr>
<tr>
<td>(x)</td>
<td>([0,100])</td>
</tr>
<tr>
<td>(a \lor x &gt;= 20)</td>
<td></td>
</tr>
</tbody>
</table>
uniformity over \{(a, True), (x, ⊥)\}, \{(a, False), (x, ⊥)\}\} instead of over \(S\). After \(a\) is decided to take a certain value \(v\), we want uniformity over the set of solutions in which \(a = v\).

If the user wants uniformity over partial solutions, he/she should be able to signal that. Currently the user cannot. This will be solved in chapter 9, semantic additions.

**Expectations on uniformity over complex ADTs**

An interesting question is how we should handle ADTs that give a choice between multiple non-recursive complex constructors. An example of this is given in the problem **Adder**. Here the ADT \(op\) can take two directions. Either it adds two numbers, or subtracts two. Notice that there are more solutions when adding, then when subtracting, since numbers below zero are not allowed. The question here is, does the user expect us to choose uniformly over which direction \(op\) takes, or uniformly over the set of solutions?

My answer is that the representation of **Adder** does not signal that the user expects us to choose uniformly over \(op\). The user should however be able to signal that he/she does want uniformity over the direction of complex ADTs. This will be added in chapter 9, semantic additions.

**Expectations on natural skew**

For many measurements in real life, one will find that the results of the measurement will follow a distribution that rises when approaching zero. For example, when measuring the age of all people in a family, the range may be from 0 to 98, but most ages will be closer to 0 than to 98. The same holds for when measuring distances between objects, amounts, lengths of texts, etc.

Since lower numbers occur more often in real life, one could assume they are more important to test, and should a get higher probability of being sampled. It should also be noted that programs are often faster when given low numbers as input, and when an unexpected result is found, the SUT is often easier to debug when the number is low.

However, doing the same test multiple times is undesired. For that reason, I do not want to deliberately introduce skews.

**Expectations on infinity**

A wish from one of the designers of TorXakis is that algorithms can work over infinite spaces. Designing a CRS algorithm so that it can handle infinity raises a question, for it is impossible to sample uniformly over an infinite space.

When looking at the problems given by the clients, all problems contain a constraint limiting their variables to certain ranges. Therefore I will assume that our CRS algorithm will always get finite variables.

Finite ranges are enforced in chapter 9, semantic additions.

**Expectations on maximum recursion depth, maximum number of nodes and mutual recursion**

Currently, all four CRS algorithms in TorXakis make use of a parameter called maxDepth, to regulate recursion depth. This parameter is adjustable by the user, but standardly has the value 4.

When this parameter is 4, this effectively means that recursion depths longer than 4 will not be generated unless specifically forced by a constraint.
When a user has a recursive ADT that calls another recursive ADT, maxDepth influences the total recursion depth: if a recursion depth of 3 is chosen, and the first ADT has a recursion depth of two, then the second ADT will enter recursion only once.

I want the controls over recursion to be completely different. Firstly, I want a maximum number of nodes, instead of a recursion depth. This because the number of nodes has a more direct relation to the size of the assignment space than the recursion depth. Secondly, I want this maximum number of nodes to be strict: when stating a maximum number of nodes, assignments with more nodes are not solutions. This makes the assignment space finite. And thirdly, I want the user to be able to specify this number of nodes per ADT, instead of in general over all ADTs. Ways for the user to specify the number of nodes will be introduced in chapter 9, semantic additions.

Mutual recursion (an ADT that calls another ADT which (possibly through other ADTs) calls the first ADT again) becomes rather hard to define and work with after these changes, and will therefore be out of scope of this thesis. It will currently be forbidden, something to possibly be solved in future research. The most notable example of mutual recursion is trees where nodes have a variable amount of branches. These can currently not be defined.

**Expectations on uniformity over the length of a list**

See the problem list. Since the field `data` has a domain of more than one value, there are more assignments where `x` has more nodes, than there are assignments where `x` has less nodes. For example, since `data` has a domain of 100 values, there are 100 times more assignments where `x` has 3 nodes, than where `x` has 2 nodes. When sampling uniformly, this will cause the CRS algorithm to neglect small lists. This is undesired behavior.

This will be solved in chapter 7, a new algorithm.
4. Testing the TorXakis algorithms

All TorXakis algorithms use the SMT solver. This solver can be used in two ways: either to answer a yes/no question (using it as an oracle), or to give a value fulfilling a constraint (using it as a sampler). All TorXakis algorithms use the SMT solver in the second way, as a sampler.

SMT solvers try to find a solution to a satisfiability question as fast as possible. Specifically Z3 uses such a vast number of optimization tricks for this, that in this thesis we will not try to reason about Z3 using its source code. Instead we will look at Z3 as a black box.

Let us look at a couple of conjectures.

Conjecture 1: Z3 is non-covering.
This seems true for some simple tests.
We ask Z3 to give an integer value with some constraint. We run this test 100 times. Between tests we reset the solver. Otherwise, Z3 will internally remember what assignment it gave and give it again instead of calculating a new assignment.
When Z3 is asked to give a value with no constraint, it returns None instead, to signal every value is valid.
When Z3 is asked to give a value $\geq 5$, it returns 5 100 times.
When Z3 is asked to give a value >5, it returns 6 100 times.
When Z3 is asked to give a value <5, it returns 5 100 times.
When Z3 is asked to give a value $\leq 5$, it returns 4 100 times.
When Z3 is asked to give a value $\neq 5$, it returns 6 100 times.
When Z3 is asked to give two integer values that should be equal, the values are 0 100 times.
When Z3 is asked to give a value in the range $[20,100)$, it returns 20 100 times.

The creators of Z3 communicate often by answering questions on forums. Such an answer is [21], where it is explained that setting the parameter ‘smt.arith.random_initial_value’ to true increases randomness in models given by Z3.

Conjecture 2: Even if this parameter is true, Z3 is still non-covering.
When Z3 is asked to give a value with no constraint, it returns None instead, to signal every value is valid.
When Z3 is asked to give a value $\geq 5$, it returns 5 100 times.
When Z3 is asked to give a value >5, it returns 6 100 times.
When Z3 is asked to give a value <5, it returns 5 100 times.
When Z3 is asked to give a value $\leq 5$, it returns 4 100 times.
When Z3 is asked to give a value $\neq 5$, it returns 6 100 times.
When Z3 is asked to give two integer values that should be equal, the values are 0 100 times.
When Z3 is asked to give a value in the range $[20,100)$, it returns 20 100 times.

So the parameter seems to have no effect. In the following tests the parameter is turned on, just in case it has effect in a fringe case.

TorXakis has four CRS algorithms, TrueBins, IncrementChoice, Partition and IncrementBins. I will introduce them quickly, and then go over them in more detail in the following sections, where I will test them on how uniformly they return from the set of solutions.
TrueBins
This algorithm partitions the assignment space into parts, which it calls bins. It then gives the assignment space and the constraint to the SMT solver, accompanied with the tautology constraint that the output should be in one of the bins. The idea behind the algorithm is based on the hope that the output of the SMT solver will skew towards the first given bin, so that when the bins are generated randomly and given in random order, the output will be random. This is often not the case.

How those bins are chosen will be discussed later.

An example of a TrueBins run: suppose we have the variable $x$, with domain $[0,30)$, and the constraint ‘$x>3$’. Our random bins are the ranges: 0 to 9, 9 to 12, 12 to 26, 26 to 30. We give those bins in random order as an extra constraint to the SMT solver. So we might give: ‘($x\geq12$ and $x<26$) or ($x\leq0$ and $x<9$) or ($x\geq26$ and $x<30$) or ($x\geq9$ and $x<12$)’. The output given by the SMT solver is our random sample.

IncrementChoice
This algorithm starts with the undefined assignment (where no variable is assigned a value) and iteratively makes partial solutions with one more variable assigned.
When encountering an integer variable, it partitions its domain in two. This is done by taking a random value $x$ and assigning all values $< x$ to one part and all $\geq x$ to the other part. Per variable, the parts are given each to the SMT solver in random order. As soon as a part contains a value that would make the partial assignment valid, the value of the variable in the output of the SMT solver is chosen as the assigned value for that variable.

How IncrementChoice handles partial assignments of other variables than integers is not relevant for this thesis.

Partition
This algorithm partitions the assignment space by partitioning the domain of each variable and making a set of all combinations that contain one domain part per variable.
For example, let us have a problem with two variables, $x$ and $y$, which we both divide in three parts, A, B and C. Our assignment space will then be divided into the parts: $((A,A),(A,B),(A,C),(B,A),(B,B),(B,C),(C,A),(C,B),(C,C))$. These parts are shuffled. Then per part we ask the SMT solver if there is a solution in this part. If so, we return that solution.

IncrementBins
Just like IncrementChoice, this algorithm starts with the undefined assignment and iteratively makes partial solutions with one more variable assigned. It partitions variable domains into multiple parts, similarly to how Partition does it. It then shuffles the domain parts, and asks the SMT solver per part whether it contains a value that would make a partial solution. If so, the variable is assigned that value in the next partial solution.

How IncrementBins and Partition partition variable domains, I will show as far as is relevant for this thesis in the following sections.

TrueBins
The idea behind TrueBins is to divide the assignment space into parts of random size, called bins, and to then make a constraint forcing that the solution should at be in one of those bins. Since the bins span the assignment space, that constraint is always true. The inherent assumption is that by randomizing the order of the bins, the SMT solver will search in more unconventional ways, thereby giving more random results.
When the space is a single int variable $x$, bins are made using division in ranges, by choosing $n$ numbers $i_1$ through $i_n$ and specifying a constraint over the returned value. How those numbers are chosen is not relevant for this thesis, they can be considered chosen randomly. The constraint looks like this:

$$x < i_1 \lor i_1 \leq x < i_2 \lor \ldots \lor x_{n-1} \leq x < i_n \lor x \geq i_n$$

TrueBins assumes infinite int spaces. When looking at instances of **generalized range** where $low$ and $high$ are constants, TrueBins appears to be non-covering.

The folders `/tests2/truebins1_1` through `/tests2/truebins2_4` (generated by the file `/tests2/test_truebins.py`, 8 folders in total), which can be found at [14], show a number of tests confirming that TrueBins is most likely non-covering when tested over instances of **generalized range**.

We will look at `/tests2/truebins2_1` specifically. It tests over **range**. It tests with 20 bins made by choosing 19 random numbers in the range $[-200,200)$. It performs 1000 runs. Each run uses newly random generated bins.

The outcomes are as follows:
- 20 is chosen 992 times.
- 52 is chosen 2 times.
- 95, 94, 92, 84, 60 and 31 are each chosen once.

As can be seen in the following table.

<table>
<thead>
<tr>
<th>value</th>
<th>times chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>992</td>
</tr>
<tr>
<td>52</td>
<td>2</td>
</tr>
<tr>
<td>95</td>
<td>1</td>
</tr>
<tr>
<td>94</td>
<td>1</td>
</tr>
<tr>
<td>92</td>
<td>1</td>
</tr>
<tr>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

Another test is `/tests2/truebins1_1`. It does the same test over **smallrange**.

Here, 0 is chosen 1000 times.

<table>
<thead>
<tr>
<th>value</th>
<th>times chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>

**Conclusion**

When looking at the tests over **range**, TrueBins only returns 8 different values in a 1000 runs. I cannot say for sure whether it will eventually give the other 992 values after enough runs, so I
cannot say for sure it is covering, but it seems likely it is not. Even the given solutions are far from uniformly distributed.
When looking at the tests over smallrange, TrueBins only returns 1 value. It seems highly unlikely that it will give other values after more runs.
This shows that we should not be using TrueBins.
Other tests can be done by adapting /tests2/test_truebins.py
IncrementChoice
IncrementChoice, when given an integer variable, will partition the domain in two using division in ranges, and pick one of the parts that fulfill the constraints. It will then give the value returned by the SMT solver. I will show here that such a single split is most likely non-covering in common problems.

We test over the problem range.
We test all possibilities, by trying every possible division (in the range [-200,200)), and by trying both parts of the division.
We do this in two tests: /tests2/choice2_1 and /tests2/choice2_2. In the first test, we will always pick the lower part of the division and in the second part the higher one. Sometimes such a part contains no solutions. Then we will disregard it.

Constraints given to the SMT solver for the first test will look like: 'x>=20 \land x<100 \land x<\text{split\_number}'.
There are 400 possible divisions, for which all divisions lower than 20 give a unfulfillable constraint, since there are no solutions below 20. Here is a graph showing per split\_number what value Z3 returned. Underneath it is a graph showing how many times each returned solution is chosen.

<table>
<thead>
<tr>
<th>range</th>
<th>x : Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20 ≤ x &lt; 100</td>
</tr>
</tbody>
</table>
Z3 does the following. When the split_number is in the valid range \([20,100)\) it returns 20 most of the time, it returns sometimes the solution closest to the split_number, and in even less cases returns a solution with seemingly no connection to the split_number. When the split_number is not in the valid range \([20,100)\), it returns 20.

Now we do the same test in folder /\texttt{tests2/choice2_2}/ if we always take the higher half of the division. Constraints will look like: ‘\(x\geq 20 \land x<100 \land x\geq \text{split}\_\text{number}\)’. Here is a graph showing per split\_number what value Z3 returned. Underneath it is a graph showing how many times each returned solution is chosen.
Z3 behaves similar to its behavior in the previous test. When the split_number is in the valid range [20,100) it returns the solution closest to the split_number most of the time, it returns 99 or 100 sometimes, and in even less cases returns a solution with seemingly no connection to the split_number. When the split_number is not in the valid range [20,100), it returns 20.

A last test/tests2/choice2_3 performs IncrementChoice a 1000 times using only split_numbers in the range [20,100). Here is a graph showing how many times each returned solution is chosen.
Conclusion

When the split_number is outside of the valid range $[20,100)$, the returned value is always 20. Let us assume that this can be generalized to any range problem: we assume that when the split_number is outside the valid range, IncrementChoice returns the lowest solution. Then we would have an extremely non-uniform distribution.

Suppose that there was a way to efficiently enforce that the split_number has to be in the valid range. Even when the split_number is in the valid range of a given range problem, the distribution of values returned by IncrementChoice appears to be non-uniform, and possibly non-covering. In a 1000 runs, only 59 of 80 solutions are returned.

This shows that we should not be using IncrementChoice.

Other tests can be done by adapting `/tests2/test_IncrementChoice.py`
Partition

Partition, when given a problem with a single integer variable, will split the domain in multiple parts of random size using division in ranges. The algorithms will then pick randomly from the parts until one is found that contains a solution, at which they return that found solution.

That means that there will be uniformly chosen from all parts that contain at least one solution. The distribution is therefore dependent on what Z3 returns, given a certain problem and a certain part. Let us look at what Z3 returns when given the problem range. The parts we will let Z3 choose a value from are all parts of size 10 that contain at least one solution.

Those are all the parts that have a start index in the range [20-9,100), or [11,100).

Our constraint given to the SMT solver will look like this:

'\(x >= 20 \land x < 100 \land x >= \text{part\_index} \land x < \text{part\_index} + 10\)'.

This is tested in folder /tests2/partition2_1, giving the following graphs. The first graph shows per part index what value is returned. The second graph shows per value how often it was returned.
Aside from two outliers, the returned value is 20 when the part index is 20 or lower, and it is the part index when the part index is between 20 and 100.
The smaller the part size, the less parts with at least one valid solution there are where the part index is 20 or lower. So the smaller the part size, the less times 20 is returned, and the closer the distribution is to uniform.
Similarly, the smaller the valid range, the less parts with at least one valid solution there are where the part index is in the valid range. So the smaller the valid range, proportionally the more times that the first value of the valid range is returned. So the smaller the valid range, the less uniform the distribution.
The following two tests, /tests2/partition2_2 and /tests2/partition2_3, show respectively when the part size is small and when the valid range is small. The test partition2_2 tests with parts of size 3. The valid range is still [20,100). The following two graphs show respectively the returned value per part index, and per value the number of times it is returned.

The test partition2_3 tests with a valid range of size 3. Solutions are 20, 21 and 22. The part size is once again 10. The following two graphs show respectively the returned value per part index, and per value the number of times it is returned.
The LCB of partition2_2 is \( \frac{1}{P(s) \times |S|} = \frac{1}{\frac{1}{12} \times 100} = 1.22 \).

The LCB of partition2_3 is \( \frac{1}{P(s) \times |S|} = \frac{1}{\frac{1}{12} \times 3} = 4 \).

**Conclusion**

When using Partition on an instance of **generalized range**, Z3 is asked to find a value that lies in two overlapping ranges, the valid range and the part. When asked such a problem, Z3 seems to behave according to a simple rule: it returns the lowest valid value. We have seen two exceptions to this rule: when the part size was 17 or 93 in partition2_1. But on average the rule holds.

For any valid range size and part size, the LCB can then be calculated as:

\[
P(s) \times |S| = \frac{1}{\frac{1}{|\text{part}| + |\text{range}| - 1} \times |\text{range}|} = \frac{|\text{part}| + |\text{range}| - 1}{|\text{range}|}.
\]

where \( P(s) \), the least likely solution, is the number of times that solution gets chosen divided by the number of runs.

We want the LCB below 2 for any problem. That includes the problem where the valid range size is 1. Our LCB then becomes \( \frac{|\text{part}|}{1} \). The only possible part size is then 1.

When the part size is 1, the algorithm Partition becomes rejection sampling. When rejection sampling, random assignments are checked whether they are valid until a valid one is found.
IncrementBins
IncrementBins behaves the same as Partition when given a problem containing only one variable (small differences between IncrementBins and Partition when given such a problem are not relevant for this thesis). When given a problem with multiple variables, IncrementBins will start with the empty partial assignment, and iteratively make more defined partial solutions by iteratively assigning a valid value to a variable. When the part size is 1, IncrementBins will do this using rejection sampling over the respective variable domain.
Rejection sampling is a uniform CRS algorithm. Iteratively making more defined partial solutions by rejection sampling over variable domains is not a uniform CRS algorithm, as I will show in this simple example.

Take the problem isZero. If \( a \) is true, then \( x \) is zero. Three sets of solutions can be distinguished: one where \( x \) is zero and \( y \) is not, one where \( y \) is zero and \( x \) is not, and one where both are zero
There are two possible variable orderings: x-then-y and y-then-x.
If we choose the order x-then-y, we first choose a value for \( x \) before choosing one for \( y \). If \( x \) is not zero, then \( y \) can only take one value. The probability per set is:

<table>
<thead>
<tr>
<th>set of solutions</th>
<th>number of solutions in set</th>
<th>probability of set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) is 0 and ( y ) is not 0</td>
<td>99</td>
<td>99/10000</td>
</tr>
<tr>
<td>( x ) is not 0 and ( y ) is 0</td>
<td>99</td>
<td>99/100</td>
</tr>
<tr>
<td>( x ) is 0 and ( y ) is 0</td>
<td>1</td>
<td>1/10000</td>
</tr>
</tbody>
</table>

If we choose the order y-then-x, the probability per set is:

<table>
<thead>
<tr>
<th>set of solutions</th>
<th>number of solutions in set</th>
<th>probability of set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) is 0 and ( y ) is not 0</td>
<td>99</td>
<td>99/100</td>
</tr>
<tr>
<td>( x ) is not 0 and ( y ) is 0</td>
<td>99</td>
<td>99/10000</td>
</tr>
<tr>
<td>( x ) is 0 and ( y ) is 0</td>
<td>1</td>
<td>1/10000</td>
</tr>
</tbody>
</table>

If we use a random order (as IncrementBins does), on average, our final distribution will be:

<table>
<thead>
<tr>
<th>set of solutions</th>
<th>number of solutions in set</th>
<th>probability of set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) is 0 and ( y ) is not 0</td>
<td>99</td>
<td>9999/20000</td>
</tr>
<tr>
<td>( x ) is not 0 and ( y ) is 0</td>
<td>99</td>
<td>9999/20000</td>
</tr>
<tr>
<td>( x ) is 0 and ( y ) is 0</td>
<td>1</td>
<td>1/10000</td>
</tr>
</tbody>
</table>

The LCB is then
\[
\frac{1}{P(s) \times |S|} = \frac{1}{\frac{1}{2000} \times 199} \approx 10.
\]

Conclusion
When given a problem with only one variable or a problem with multiple variables, where valid values of variables are independent of the values of other variables, IncrementBins can give uniformly distributed solutions, if the part size is 1.
But when given a problem with multiple variables, where whether a value is valid is dependent on the values of other variables, IncrementBins does not return solutions uniformly distributed. Instead, IncrementBins can have a large LCB as we saw when it is given isZero.
Summary of this chapter
We have tested all four TorXakis algorithms on instances of generalized range. TrueBins and IncrementChoice both showed heavily skewed and likely non-covering distributions. Partition and IncrementBins only had LCBs lower than 2 if the part size is set to 1. IncrementBins showed high LCBs in cases where the valid values of variables are dependent on the values of other variables. Partition with the part size set to 1 behaves exactly like rejection sampling.

In chapter 3, expectations, I stated that I want an algorithm that has an LCB lower than 2 for every possible problem. According to that expectation, if we find an algorithm that is faster than rejection sampling, while keeping the LCB under 2, we have found an algorithm that is better than the TorXakis algorithms.
5. An overview of CRS algorithms

I have tried to list and categorize all best versions of CRS algorithms. This list is not complete; for every algorithm there are others imaginable which are close in definition and which may or may not perform better. I have categorized the algorithms based on what language they use. The languages are described in chapter 2, preliminaries. In the next chapter, chapter 6, I will show a table listing properties of all CRS algorithms and compare them. In chapter 7 and 8 I will introduce an algorithm that fits the expectations, and is faster than competing algorithms.

In this chapter, I describe algorithms globally, focusing on the underlying idea, rather than the details, in order to keep the description somewhat short. For a full definition I refer to the respective papers.

Specialized processor languages
These algorithms are heavily specialized towards specific problems. They include genesys-pro[16] and AVPGEN[17], which are aimed at the architectural level of processors. They expect a problem to be described using processor instructions. Another is X-gen[18], aimed at systems and systems on chips. Threadmill[19] is specialized for post-silicon validation for multi-threaded processors. Piparazzi[20] for micro-architectures. More algorithms exist.
Note that these are commercial tools/algorithms, so they cannot be used in TorXakis, since TorXakis is free.

Formulas with expression relation literals

RACE
The language used by RACE[4] is not exactly formulas with expression relation literals, but for the purpose of this thesis it translates to it easily. RACE starts with the empty partial assignment and iteratively creates more defined partial assignments. When it finds that the partial assignment it has created is not a partial solution, it backtracks. RACE applies a number of interesting tricks, such as propagating ranges (when a constraint restricts a variable space to a smaller range, this information is used), detecting various cycles, and good heuristics for choosing which variable to observe next in the variable ordering.
But alas, RACE is commercial and can therefore not be used.

CNF with expression relation literals

Ambigen
Ambigen[11] is based on SampleSAT (see below), and extends it to CNF with expression relation literals. Just like SampleSAT, it uses Monte Carlo Markov Chain methods to walk over the assignment space. It walks either randomly or towards a solution based on some complicated heuristics.

Formulas with boolean literals

QuickSampler
The idea behind QuickSampler[12] is to mass produce assignments that are likely solutions, and to filter out solutions. Generating solutions fast is considered more important than making sure all solutions can eventually be found. It starts with a solution (given by the SAT solver), and makes all inverse neighbours (by taking all neighbours and flipping all their bits). It then checks of all these assignments if they are solutions. For every that is, their inverse neighbours are made and checked.
if they are solutions, yielding out duplicates. As shown in their paper, on common assignment spaces these inverse neighbours turn out to be solutions quite often.

On some problems, QuickSampler will be non-covering. Those are the problems where there is at least one solution of which the inverse neighbours are not solutions, and the solution is not returned when the SAT solver is asked for a solution.

**woBDDs**
A weighted ordered Binary Decision Diagram is a compact data structure for representing all solutions of a boolean formula as a diagram with one root and two leaves, True and False. Every node represents a partial solution. Weights at the nodes represent the amount of solutions that can be made from this partial solution. The paper [22] introduces an algorithm that makes such a woBDD and uses it to uniformly generate samples.

**CNF with boolean literals**

**SPUR**
SPUR[7] adapts SharpSAT[13], a model counter. This model counter not only counts, but also lists all solutions of a problem. SPUR needs a large start-up time, but can then mass produce uniform solutions.

**SearchTreeSampler**
SearchTreeSampler[5] divides all booleans into parts of equal size. It then picks a part and generates all partial solutions with those booleans assigned a value (using a SAT solver). It then chooses a subset of those partial solutions and a new part, and generates all partial solutions that are more defined. And so on, until all variables have defined values, and we have a set of solutions. Since we only need one sample, we need after every iteration only one partial solution, instead of a subset.

Uniformity is dependent on how many groups are made. The less groups, the closer this algorithm is to iterating over the solution space.

**3-CN with boolean literals**
3-CNF is a specific form of CNF where every clause has maximally 3 literals.

**SampleSAT**
SampleSAT[3] is based on the SAT solver WalkSAT. This SAT solver applies an Monte Carlo Markov Chain method which the authors call biased random walking, which is very alike Hill Climbing: it starts with a random assignment, and changes the assignment towards one that fulfills more constraints.

SampleSAT interweaves steps in this biased random walking with steps from Fixed Temperature Simulated Annealing. The first method is good at finding a solution, the second at making that solution random.

**Unigen2**
Unigen2[6] uses partition refinement (it iteratively partitions the space in smaller parts) through xor hashes until the algorithm decides that parts are small enough. It then picks such a small part randomly, finds all solutions in it, and picks one randomly.

As shown in chapter 2, preliminaries, xor hashes do not partition completely random: certain values will never be in the same partition. This makes this algorithm not perfectly uniform.
6. Comparison of CRS algorithms

The following table lists all algorithms introduced in this paper with some properties.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>language</th>
<th>uniform</th>
<th>covering</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>rejection sampling</td>
<td>any finite language</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>iterating over the answer space</td>
<td>any finite language</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>iterating over the solution space</td>
<td>SMT2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>TrueBins</td>
<td>TorXakis specification language</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>IncrementChoice</td>
<td>TorXakis specification language</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Partition (with part size &gt;1)</td>
<td>TorXakis specification language</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>IncrementBins</td>
<td>TorXakis specification language</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>specialized processor algorithms</td>
<td>specialized processor languages</td>
<td>no</td>
<td>unknown</td>
<td>no</td>
</tr>
<tr>
<td>RACE</td>
<td>formulas with expression inequality literals</td>
<td>no</td>
<td>unknown</td>
<td>no</td>
</tr>
<tr>
<td>Ambigen</td>
<td>CNF with expression inequality literals</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>QuickSampler</td>
<td>formulas with boolean literals</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>woBDDs</td>
<td>formulas with boolean literals</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SPUR</td>
<td>CNF with boolean literals</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SearchTreeSampler</td>
<td>CNF with boolean literals</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SampleSAT</td>
<td>3-CNF with boolean literals</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Unigen2</td>
<td>3-CNF with boolean literals</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
The three base CRS algorithms, rejection sampling, iterating over the answer space, and iterating over the solution space can all be used without translation (translating to SMT2 is trivial, the TorXakis specification language can be seen as a subset of that language). They are however quite slow. Specifically both ways of iterating are extremely impractical. Users of TorXakis should be able to ask for an integer value in the range \([-2^{31}, 2^{31}]\) without trouble. Both ways of iterating would take (at least) \(2^{-32}\) steps to generate all assignments/solutions.

All four of the TorXakis algorithms have a too high LCB, except for Partition when the part size is 1, but in that case it behaves as rejection sampling.

I do not know the LCB of the specialized processor algorithms, and I do not know if they are covering. They are not described in detail enough to recreate them, and make use of terminology from their respective fields, of which I do not know enough. They are not suited for TorXakis, since they are not free, and since they use languages that are quite different from the TorXakis specification language.

RACE uses the language formulas with expression relation literals. In chapter 8, section 1, we will see that the TorXakis specification language cannot be completely translated to formulas with expression relation literals. So even if RACE was free, it would not fit our requirements. In chapters 7 and 8 we will make a base algorithm that works on the TorXakis specification language and extend it with space reduction tactics that use formulas with expression relation literals. Afterwards I will show the differences between those tactics and the tactics used in RACE, as far as they are described. The tactics used in RACE are not described in full detail in their paper, since they are too complicated (and because the tool is commercial).

Ambigen and SampleSAT use Monte Carlo Markov Chain methods: they walk the search space starting from a random assignment towards a solution. These algorithms tend to show very non-uniform distributions. They are therefore not in the direction we want to take.

Quicksampler is non-covering, and therefore not an option.

woBDDS, SPUR, SearchTreeSampler and Unigen all work on boolean formulas. The assignment space of problems users can give can be quite huge. Translating such an assignment space to booleans is not an option. Simple constraints too can become quite large when translating to a boolean formula. For example when translating a problem involving a comparison operator (like \(>\)) over finite integer ranges, the constraint becomes exponential to the size of the integer range.

None of the algorithms fit completely what we need. Rejection sampling fits most, as in that it is usable on the TorXakis Specification language, is uniform, and is the fastest of the three base algorithms. But it is quite slow.

In the next chapter two chapters I will introduce an algorithm that fits the expectations and is on common problems faster than rejection sampling.
7. A new algorithm

In this chapter I will introduce a new algorithm, called \textit{iteratively\_halve}. It can be used with the TorXakis specification language, without a non-trivial translation. This because it primarily uses Z3, which uses the language SMT2, which is trivial to translate to from the TorXakis specification language.

This algorithm will be our base algorithm. In the following chapter we will extend this base algorithm with tactics for space reduction that do need a translation.

The algorithm uses partition refinement: it iteratively partitions the assignment space into smaller parts. It keeps all parts that contain a solution, and disregards parts that do not.

The first category will always keep a uniform distribution, as long as it picks solutions uniformly from the parts it keeps. This makes it comparable to rejection sampling, which also has a uniform distribution, and also does not need translations.

Rejection sampling is fast when the space is dense. It is however extremely slow when the space is sparse.

Because our algorithm filters (possibly large) parts that do not contain solutions, it does not have that deficit. We make use of the assumption that Z3 can decide whether a certain part contains no solutions a lot faster than the time lost if the part was not excluded. For common problems, Z3 can decide this for a given part in just a few steps. For example, since Z3 can reason with comparison operators, when given an instance of \textbf{generalized range}, it can decide near instantly whether a certain part contains a solution.

The pseudocode is:

\begin{verbatim}
definition iteratively\_halve(space):
  parts ← \{space\}
  while True do
    s ← maybe\_find\_solution(parts)
    if s ≠ None then
      return s
    newparts ← \{\}
    for part in parts do
      newparts ← newparts ∪ partition(part)
    parts ← filter\_valid(newparts)
\end{verbatim}

where:

The implementation of \textit{maybe\_find\_solution} tries a couple of rejection samples. If one finds a solution, it returns it, otherwise it returns None.

The implementation of \textit{filter\_valid} will ask Z3 per part whether it contains a solution.

The implementation of \textit{partition} is a bit more complicated. It receives as input an assignment space. It tries to split the space in two equally sized parts. It does not need to, the way \textit{partition} splits has no influence on the uniformity as long as \textit{maybe\_find\_solution} samples uniformly from
the set of parts. But splitting in equally sized parts makes the algorithm faster, since it makes it most likely that the part to discard is large.

An assignment space is a set of domains. The domains in TorXakis are finite int ranges, booleans, chars, strings and user defined ADTs. An ADT is a type that has one or more constructors. Each constructor has zero or more fields. Each field has a domain. The set of possible ADTs can be split up in enumerations, complex non-recursive ADTs and recursive ADTs.

`partition` chooses one domain from its input at random.

- If that domain can be defined as $[\text{low}, \text{high})$, it divides it in $[\text{low}, \text{middle})$ and $[\text{middle}, \text{high})$, where $\text{middle} = \frac{\text{high} - \text{low}}{2}$. Domains that can be defined as such are integer ranges, enumerations, booleans and chars.
- If the chosen domain is a complex non-recursive ADT instead, it divides its constructors over two parts. If the ADT has only one constructor, it instead divides its fields over two parts. If the constructor has only one field, it instead divides the domain of that field.
- If the chosen domain is a recursive ADT, we make a distinction between whether it is a list (it has only one recursive constructor, which has only one recursive field) or a tree.
  - If it is a list, we define the amount of nodes it may take as a range $[\text{low}, \text{high})$. The function `partition` makes two parts where one has all assignments where the amount of nodes is in $[\text{low}, \text{middle})$ and the other has all assignments where the amount of nodes is in $[\text{middle}, \text{high})$. If the amount of nodes the recursive ADT may take is in a range of size 1, then we know the length of the list. `partition` then chooses an element of the list at random and partitions that.
  - If the ADT is a tree, we handle it as a complex non-recursive ADT: `partition` divide its constructors over two parts. If the ADT has only one constructor, it instead divides its fields over two parts. If the constructor has only one field, it instead divides the domain of that field.

A remaining question is how many rejection samples `maybe_find_solution` should do per iteration. That is extremely situation dependent, and therefore does not have a clear answer. When facing a problem with a difficult constraint, asking z3 whether a part contains a solution will be much slower than a rejection sample. More research could find an optimal value, but for now I suggest setting the number of rejection samples per iteration rather arbitrarily to 10.

One of the expectations in chapter 3 was to choose uniformly over the length of a list. If one chooses uniformly from all values instead, lists of long length are more common, since there are more possible lists of long length. This is undesired behavior, since lists of short length should be tested too.

Choosing uniformly over the length of a list is as trivial as sampling uniformly over it. This is done using a two step sample: first we choose a value for the length of the list, secondly we choose a value for every element. This two-step-created list value is then rejection sampled.
Rejection sampling over ranges, as described by **generalized range**, can take a long time when \( \text{high} - \text{low} \) is much larger than \( \text{high2} - \text{low2} \). **iteratively_halve** will be faster, since it can cut empty parts. Z3 has no trouble with this constraint. Let us look at a made up example run over **range in a finite domain**, an instance of **generalized range**. In the first iteration, **maybe_find_solution** will try 10 rejection samples, which have an extremely low probability of hitting a solution. In this example run all are rejected. The space is then partitioned by **partition** into the parts \([-2^{31}, 0)\) and \([0, 2^{31})\). The first part is rejected by **filter_valid**. Our space is now \([0, 2^{31})\). In the second iteration, again none of the 10 rejection samples hit. The space is partitioned into the parts \([0, 2^{30})\) and \([2^{30}, 2^{31})\), of which the second part gets filtered out. 21 more iterations happen, in which no solution is hit. Our space is now \([0, 2^9)\) or \([0, 512)\). The probability of one of the rejection samples hitting is 1 - the probability no rejection samples hit.

The probability that no rejection samples hit is \(\left(\frac{512 - 80}{512}\right)^{10} \approx 18\%\). The probability that a rejection sample hits is 82\%. But in this example run none of the rejection samples hit. In the next iteration our space is \([0, 256)\). The probability of a rejection sampling hitting is now 98\%. One rejection sample hits and the solution is returned.

In total this took 25*10 rejection samples and 24*2 Z3 calls by **filter_valid**, so a total of 298 events. This while when doing pure rejection sampling, after \(2^{16}\) rejection samples, the probability of one of those samples hitting would be \(1 - \left(\frac{2^{32} - 80}{2^{32}}\right)^{2^{16}} \approx 0.1\%\).

**tests**

The algorithm is implemented in `/test2/test_iteratively_halve.py`. When ran 20,000 times on **range in a finite domain**, a close to uniform distribution can be seen. Running more times will show a more uniform distribution.
The algorithm performed on average:
214 rejection samples
21 iterations
41 requests whether a part contains a solution

When ran a 1000 times on **equality in a finite domain**, the algorithm performed on average:
278 rejection samples
28 iterations
54 requests whether a part contains a solution

When ran a 1000 times on **circle**, the algorithm performed on average:
1 rejection sample
1 iteration
0 requests whether a part contains a solution
8. Space reduction tactics

The algorithm introduced in the previous chapter is our base algorithm. It does not look at constraints (any more than passing them to Z3). That while constraints often contain easy to parse information that could drastically reduce the assignment space, sometimes even directly to the solution space. In this chapter we extend the base algorithm with space reduction tactics. This is difficult when using the TorXakis specification language, since it is a very complex language. Therefore we will first discuss how to translate to the easier language formulas with expression relation literals.

The TorXakis specification language has as types finite integer ranges, booleans, chars, strings, and user defined ADTs. The set of possible ADTs can be split up in three smaller sets: enumerations, complex non-recursive ADTs and recursive ADTs.

A constraint may be defined using functions. Functions are allowed to be recursive. It is possible to define functions that will never halt.

Formulas with expression relation literals has as types only integers with finite ranges. Its constraint is defined as:

\[
\begin{align*}
\text{constraint} & : \quad \text{constraint op constraint} \mid \text{literal} \\
\text{op} & : \quad \land \mid \lor \mid \rightarrow \mid \leftrightarrow \mid \neg \mid \otimes \\
\text{literal} & : \quad \text{True} \mid \text{False} \mid \text{expression relop expression} \\
\text{relop} & : \quad \leq \mid \geq \mid \lt \mid \gt \mid \lt \mid \gt \mid \neq \\
\text{expression} & : \quad I_i \mid i \in \mathbb{N} \mid \text{expression arithop expression} \\
\text{arithop} & : \quad + \mid - \mid \div \mid \ast
\end{align*}
\]

Translation to formulas with expression relation literals

**Enumerations and chars**

Enumerations will be translated to a number in the range \([0, \text{number of enumeration values})\). Chars will be translated to a number in the range \([0, \text{number of char values})\).

**Complex non-recursive constructors**

Complex ADTs will be flattened to remove complex non-recursive constructors. For example, take `struct`. The flattened version has a variable for every field in the ADT and its children ADTs. It is shown in `struct2`. Notice the similarities between the name of variables in the flattened version (for example \(x\_\text{person}\_\text{name}\)) and how one would reference a field in a constraint in the unflattened version (for example `name(x)`).

```
struct

<table>
<thead>
<tr>
<th>x : person</th>
</tr>
</thead>
<tbody>
<tr>
<td>person = Person(name = String, age = [0, 100], favorites = favorites)</td>
</tr>
<tr>
<td>favorites = Favorites(food = food, drink = drink)</td>
</tr>
<tr>
<td>food = Pizza</td>
</tr>
<tr>
<td>drink = Coffee</td>
</tr>
</tbody>
</table>
```
Choice between multiple complex constructors
A choice between multiple complex constructors, such as in \textbf{Op}, is flattened by creating an enumeration of all choice paths and stating that if a certain path is not taken, the values belonging to it should be \(\perp\). This is shown in \textbf{Op}\textsubscript{2}.

\begin{verbatim}
struct2
  x_name : string[0,100)
  x_favorites_food : {Pizza,Fries}
  x_favorites_drink : {Coffee,Tea}
\end{verbatim}

\textbf{Op}

\begin{verbatim}
x : op
op = Plus(left = [0, 100), right = [0, 100]) | Minus(left = [0, 100), right = [0, 100)]
\end{verbatim}

\textbf{Op}\textsubscript{2}

\begin{verbatim}
x_op : {plus, minus}
x_plus_left : [0,100) \cup \{\perp\}
x_plus_right : [0,100) \cup \{\perp\}
x_minus_left : [0,100) \cup \{\perp\}
x_minus_right : [0,100) \cup \{\perp\}

x_op = plus \leftrightarrow x_plus_left \neq \perp
x_op = plus \leftrightarrow x_plus_right \neq \perp
x_op = minus \leftrightarrow x_minus_left \neq \perp
x_op = minus \leftrightarrow x_minus_right \neq \perp
\end{verbatim}

Recursive ADTs
Recursive ADTs should be accompanied with a maximum amount of nodes \(n\). A recursive ADT exists out of two parts: the nodes and the leaves. The maximum amount of leaves \(l\) is dependent on how many children \(c\) nodes are allowed to have. \(l = c \times n - (n - 1)\), since every node can have \(c\) children, so maximally \(c \times n\), but then \((n - 1)\) of those children are not leaves but nodes.

Here we show the translation of a binary tree where \(n\) is 3. Then \(l\) is 4. We can see three kinds of variables: constructor choice variables similar as in \textbf{Op}\textsubscript{2}, node variables and leaf variables. Node and leaf variables can either take a value or are \(\perp\). The constructor choice variables specify the structure of the tree, by specifying who are the children of every node.

\begin{verbatim}
Tree
x : tree

tree = Leaf(data = [0, 100)) | Node(data = [0, 100), left = tree, right = tree)
\end{verbatim}
As one can see, quite a lot of constraints are needed to ensure the resulting structure forms a tree. For brevity and readability, the constraints are not written out completely, and are instead written in English.

The most common recursive ADT is a list. It is a tree where every node has only one child and there is only one leaf. A translation is shown here.

### Strings
Strings are lists of chars.

### Functions
Since in formulas with expression relation literals functions do not exist, we will replace occurrences of function calls with the respective function body. This is called inlining.

We cannot inline recursive functions however, since we do not know the amount of recursions they will take, or whether they will halt. This means that we cannot translate recursive functions. This means that we will have some illegible parts in our translation on which we can do no space reductions.
Range shrinking and tethering

We will apply two space reduction tactics which I call range shrinking and tethering. When range shrinking, one tries to find a smaller range for a variable. When tethering, one tries to define the range of a variable as dependent on an expression. Let us look at an example.

<table>
<thead>
<tr>
<th>problem1</th>
<th>reduced1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : [0,11)$</td>
<td>$x : [2y, 2y+1)$</td>
</tr>
<tr>
<td>$y : [0,100)$</td>
<td>$y : [5,10)$</td>
</tr>
<tr>
<td>$x = 2y \land 5 \leq y &lt; 10$</td>
<td>$0 \leq x &lt; 11$</td>
</tr>
</tbody>
</table>

In problem1, without preprocessing, there are 1100 assignments. That while this problem has exactly one solution: $(10, 5)$. But when applying range shrinking, we see that $y$ can be defined on a smaller range, and shrink its range to $[5, 10)$. And when tethering, we see that $x$ can be defined based on $y$, so that its range has size 1: $[2y, 2y+1)$. With these reductions, the assignment space size is 5, and finding a solution can be done in maximally 5 rejection samples.

In the clients problems, situations where ranges are defined over variables and where variables are defined as properties, occur often. Those situations can be solved efficiently with these two tactics.

The language the tactics work on is formulas with expression relation literals:

- $constraint : constraint \ or \ constraint \ | \ literal$
- $op : \land \ | \lor \ | \rightarrow \ | \leftrightarrow \ | \neg \ | \otimes$
- $literal : \text{True} \ | \text{False} \ | \ expression \ relop \ expression$
- $relop : \leq \ | \geq \ | \lt \ | \gt \ | \leq \ | \geq \ | \lt \ | \gt \ | = \ | \neq$
- $expression : I_i \ | \ i \in \mathbb{N} \ | \ expression \ arithop \ expression$
- $arithop : + \ | \ - \ | \div \ | \ast$

Here is the pseudocode for both tactics, assuming the constraint can be written as a conjunction of expression relation literals. We will look later at how to handle other boolean operators. Here we call defining a variable solely on constants and not on other variables: tethering it to the ground. As such, $[0, 11)$ (or $0 \leq x < 11$) is the ground range of $x$.

The term rewrite appears at some places in the pseudocode. Rewrites can perform addition, subtraction, or side switching ($a > b$ become $b < a$).
definition shrink_ranges(problem):
1: \( R \leftarrow \text{dict}() \)
2: for var in \( \mathcal{V} \) do
3: \( R[var] \leftarrow \{\} \)
4: for literal in constraint do
5: if literal can be rewritten to var relop constant then
6: \( R[var] \leftarrow \text{relop constant} \)
7: for var in \( \mathcal{V} \) do
8: ground range of var \leftarrow \text{find_smallest_range}(R[var], var)

definition find_smallest_range(set, var):
1: rewrite all = constant in set to \( \geq \) constant and \( \leq \) constant
2: set \leftarrow \text{ground range of var}
3: set \leftarrow set \setminus \{\neq \text{constant}\}
4: upper \leftarrow \text{highest constant in set where relop is} < \text{or} \leq
5: lower \leftarrow \text{lowest constant in set where relop is} > \text{or} \geq
6: if upper \leq lower then
7: state is invalid
8: else
9: return \((\text{lower, upper})\)
**definition** tether(state):

1: \( \mathcal{T} \leftarrow \text{dict()} \)
2: for var in \( \mathcal{Y} \) do
3: \( \mathcal{T}[\text{var}] \leftarrow \{\} \)
4: for literal in constraint do
5: \( \quad \) for var in \( \mathcal{Y} \) do
6: \( \quad \) if literal can be rewritten to var relop exp + constant then
7: \( \quad \) \( \mathcal{T}[\text{var}] \leftarrow \) relop exp + constant
8: \( \mathcal{R} \leftarrow \text{dict()} \)
9: for var in \( \mathcal{Y} \) do
10: \( \quad \mathcal{R}[\text{var}] \leftarrow \text{all.possible.ranges}(\mathcal{T}[\text{var}], \text{var}) \)
11: \( \mathcal{Y}^s \leftarrow \mathcal{Y} \) sorted on variable with largest ground range first
12: for var in \( \mathcal{Y}^s \) do
13: \( \quad \) ground range of var = smallest range from \( \mathcal{R}[\text{var}] \) that does not contain a previous variable in \( \mathcal{V}^s \)

**definition** all.possible.ranges(set, var):

1: rewrite all = exp + constant in set to \( \geq \) exp + constant and \( \leq \) exp + constant
2: set \( \leftarrow \) ground range of var
3: set = set \setminus \{\# constant\}
4: \( \mathcal{G} \leftarrow \text{dict()} \)
5: for comp exp + constant in set do
6: \( \quad \) if \( \mathcal{G}[\text{exp}] = \bot \) then
7: \( \quad \) \( \mathcal{G}[\text{exp}] \leftarrow \{\text{comp exp} + \text{constant}\} \)
8: \( \quad \) else
9: \( \quad \) \( \mathcal{G}[\text{exp}] \leftarrow \text{comp exp} + \text{constant} \)
10: for exp in keys(\( \mathcal{G} \)) do
11: \( \quad \) if \( \mathcal{G}[\text{exp}] \) does not contain a < or \( \leq \) then
12: \( \quad \) \( \mathcal{G}[\text{exp}] \leftarrow \bot \)
13: \( \quad \) if \( \mathcal{G}[\text{exp}] \) does not contain a > or \( \geq \) then
14: \( \quad \) \( \mathcal{G}[\text{exp}] \leftarrow \bot \)
15: \( \mathcal{R} \leftarrow \{\} \)
16: for exp in keys(\( \mathcal{G} \)) do
17: \( \quad \) upper \( \leftarrow \) highest constant in \( \mathcal{G}[\text{exp}] \) where comp is \( < \) or \( \leq \)
18: \( \quad \) lower \( \leftarrow \) highest constant in \( \mathcal{G}[\text{exp}] \) where comp is \( > \) or \( \geq \)
19: \( \quad \) if upper \( \leq \) lower then
20: \( \quad \) problem is invalid
21: \( \quad \) else
22: \( \quad \) \( \mathcal{R} \leftarrow \langle \text{lower}, \text{upper} \rangle \)
23: return \( \mathcal{R} \)
The complexity of \textit{find.smallest.range} is $O(\text{set})$, caused by line 4 and 5. When called by \textit{shrink.range}, it is $O(\text{literals})$. The complexity of \textit{shrink.range} is $O(\mathcal{V} \times \text{literals})$, caused by line 8.

The complexity of \textit{all.possible.ranges} is $O(\text{set})$, caused by line 5 to 9. When called by \textit{tether}, it is $O(\mathcal{V}^2 \times \text{literals})$. The complexity of \textit{tether} is $O(\mathcal{V}^2 \times \text{literals})$, caused by line 10.

This keeps their implementation in polynomial time.

Here are some examples of what range shrinking and tethering can and cannot do.

In \textbf{problem2}, tethering will notice that $x$ can be defined as $[y, y + 10)$, and similarly that $y$ can be defined as $(x - 10, x]$, because if one variable can be defined as a range over another, so the other can be defined as range over the first variable. Since the range of $x$ is smaller than the range of $y$, $y$ will be defined based on $x$.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{problem2} & \textbf{reduced2} \\
\hline
$x$ : & $[0,20)$ \\
$y$ : & $[0,100)$ \\
$y \leq x < y + 10$ & $0 \leq y < 100$ \\
\hline
\end{tabular}
\end{center}

In \textbf{problem3}, range shrinking will notice that $y$ can be defined as $[10, 100)$. A smart observer may notice, that since $x \geq y$, $x$ too can be defined as $[10, 100)$, but neither range shrinking nor tethering will notice this. Range shrinking only searches for comparisons with constants, and tethering needs both an upper and a lower bound over the same expression.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{problem3} & \textbf{reduced3} \\
\hline
$x$ : & $[0,100)$ \\
$y$ : & $[0,100)$ \\
$x \geq y \land y \geq 10$ & $x \geq y \land y \geq 10$ \\
\hline
\end{tabular}
\end{center}

In \textbf{problem4}, $x$ can be defined as $2y$, but the $y$ cannot be defined as $\frac{1}{2}x$. Both range shrinking and tethering do not apply any arithmetic aside addition and subtraction. This to stay away from rounding error difficulties introduced by division.

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{problem4} & \textbf{reduced4} \\
\hline
$x$ : & $[0,20)$ \\
$y$ : & $[0,100)$ \\
$x = 2y$ & $0 \leq x < 20$ \\
\hline
\end{tabular}
\end{center}

In \textbf{problem5}, $x$ can be defined as $y + z$, but $y$ can be defined as $x - z$, and $z$ can be defined as $x - y$. Tethering is smart enough to realize though that we should have no circular definitions.
In problem 6, range shrinking is smart enough to realize that \( x \) has no valid assignments, and that the range of \( y \) can be shrunk to \([5, 10)\).

\[
\begin{array}{|c|c|}
\hline
\text{problem 6} & \text{reduced 6} \\
\hline
x : & [0, 100) \\
y : & [0, 100) \\
\hline
2 \leq x < 5 & 10 \leq x < 12 \\
2 \leq y < 10 & 5 \leq y < 12 \\
\hline
\end{array}
\]

In problem 7.1, tethering will see that \( x = y + 1 \) can be rewritten to \( y = x - 1 \), and in an effort to find the smallest range, realize that \( y = x - 1 \) and \( y = x + 1 \) constrain all values of \( y \). However, in problem 7.2 tethering will not see that there is no solution. It will notice that for all three variables that they can be expressed in both other variables. It will also recognize that they can be circularly defined, and that therefore at least one of them should stay defined using a constant range. But it does not recognize that if \( x \) can be expressed in \( y \), and \( y \) in \( z \), that \( x \) can be expressed in \( z \). And without that realization it does not recognize that there are no valid solutions.

\[
\begin{array}{|c|c|}
\hline
\text{problem 7.1} & \text{reduced 7.1} \\
\hline
x : & [0, 100) \\
y : & [0, 100) \\
x = y + 1 & \text{invalid} \\
y = x + 1 & \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{problem 7.2} & \text{reduced 7.2} \\
\hline
x : & [0, 100) \\
y : & [0, 100) \\
z : & [0, 100) \\
x = y + 1 & \\
y = z + 1 & \\
z = x + 1 & \\
0 \leq x < 100 & \\
0 \leq y < 100 & \\
z = x + 1 & \\
\hline
\end{array}
\]

In problem 8, \( y \) and \(-y\) are considered two different expressions. Since tethering only defines ranges over the same expressions, no smaller range can be found here.

\[
\begin{array}{|c|}
\hline
\text{problem 8} \\
\hline
x : & [0, 100) \\
y : & [0, 100) \\
x < y & \\
x > -y & \\
\hline
\end{array}
\]
Multiple worlds

In all the examples above, no logical or (\(\lor\)) occurred. It does in problem9. This creates three possibilities (worlds) and excludes one. If \(a\) is false, \(x\) cannot be \(\geq 20\). We want to express this in ranges, instead of constraints. We do this by splitting problem9 in three subproblems.

<table>
<thead>
<tr>
<th>problem9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True, False}</td>
</tr>
<tr>
<td>(x) : ([0, 100))</td>
</tr>
<tr>
<td>(a \lor x &lt; 20)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>problem9a</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True, False}</td>
</tr>
<tr>
<td>(x) : ([0, 100))</td>
</tr>
<tr>
<td>(a \land x &lt; 20)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>problem9b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True, False}</td>
</tr>
<tr>
<td>(x) : ([0, 100))</td>
</tr>
<tr>
<td>(a \land \neg x &lt; 20)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>problem9c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True, False}</td>
</tr>
<tr>
<td>(x) : ([0, 100))</td>
</tr>
<tr>
<td>(\neg a \land x &lt; 20)</td>
</tr>
</tbody>
</table>

Which then only contain logical ands. These subproblems can be given to shrink_ranges() to produce the following subproblems.

<table>
<thead>
<tr>
<th>reduced9a</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True}</td>
</tr>
<tr>
<td>(x) : ([0, 20))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reduced9b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {True}</td>
</tr>
<tr>
<td>(x) : ([20, 100))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>reduced9c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : {False}</td>
</tr>
<tr>
<td>(x) : ([0, 20))</td>
</tr>
</tbody>
</table>

Let us generalize this for all constraints written in CNF (with expression relation literals). Such a constraint is an and of clauses, which are ors of literals. Such a clause can be rewritten into multiple subproblems as we have done in problem9 to problem9a, problem9b, and problem9c. Then each clause represents multiple subproblems. The product of these clauses represents all possible worlds we can make. These worlds too can be rewritten to subproblems. These subproblems can then be reduced by range shrinking and tethering.

Let us express that in pseudocode:

```plaintext
definition CRS(problem):
1: convert constraint to CNF with expression inequality literals
2: \(P \leftarrow \{}\)
3: for clause in constraint do
4: \(P \leftarrow P \prod_{\text{clause}} \{\text{literal}, \neg \text{literal}\}\)
5: \(P \leftarrow P \setminus \{\neg \text{literal for literal in clause}\}\)
6: \(P \leftarrow P\)
7: all_worlds \leftarrow \prod P
8: all_subproblems \leftarrow \{(Y, world) for world in all_worlds\}
9: for subproblem in all_subproblems do
10: subproblem \leftarrow shrink.ranges(subproblem)
11: subproblem \leftarrow tether(subproblem)
12: filter_valid(all_subproblems)
13: return iteratively_halve(all_subproblems)
```

45
The lines 3 through 8 do the following. Per clause, \( \mathcal{P} \) will contain all possible worlds created by that clause. A world exist out of either a literal or its negation per clause. In a clause, the only world that can be made out of its literals and their negations that is not valid, is the one where every literal is negated. The lines 4 and 5 create \( \mathcal{P} \). \( \mathcal{P} \) is the set of all \( \mathcal{P} \)'s.

In a constraint, all clauses should hold. So one creates one world of a constraint by picking one world from a clause for each clause. At line 7 we create all such worlds.

At line 8 we make all subproblems. problems are a tuple of a set of variables and a constraint. A world is a collection of literals, and as such a constraint.

Problems that contain many logical ors will create many worlds. Luckily, in a lot of the clients problems, there were few or none logical ors. This can partly be explained by the expressiveness of expressions, and partly by the fact that TorXakis models using a transition system, giving users the possibility to model boolean choices as transition steps.

One question is left unanswered, because how do we give multiple subproblems to our CRS algorithm? Our CRS algorithm is \textit{iteratively halve}, and needs to be able to do two things: to rejection sample and to ask Z3 whether there is a solution in a part of the assignment space. Let us look at \textbf{problem9}. \textbf{reduced9a} and \textbf{reduced9c} each contain 20 assignments, \textbf{reduced9b} contains 80 assignments. If we want to give each assignment equal probability of being chosen when rejection sampling, we should therefore sample from the subproblems with ratio 20:80:20. Computing the size of the assignment space of such a subproblem, and sampling according to a ratio is trivial. In this case, all assignments are solutions, but let us imagine that is not the case. After a certain amount of failed rejection samples, \textit{iteratively halve} will call \textit{partition} to divide the problem into parts. The easiest way is to make each subproblem a separate part. In the next iteration it can partition each remaining subproblem.

**Optimizations**

The bottleneck of this algorithm is creating all the worlds of a constraint. That is why in this section we focus on ways to reduce the amount of worlds.

Firstly, \textbf{problem9} could also be divided in two subproblems, instead of three. Either \textbf{problem9a} and \textbf{problem9b} could be combined to one subproblem, or \textbf{problem9a} and \textbf{problem9c}.

There are therefore two ways in which we can express \textbf{problem9} as two subproblems:

<table>
<thead>
<tr>
<th>State9ab</th>
<th>problem9c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : { \text{True, False} } )</td>
<td>( a : { \text{True, False} } )</td>
</tr>
<tr>
<td>( x : [0,100) )</td>
<td>( x : [0,100) )</td>
</tr>
<tr>
<td>( )</td>
<td>( \neg a \land x &lt; 20 )</td>
</tr>
</tbody>
</table>

and:

<table>
<thead>
<tr>
<th>State9ac</th>
<th>problem9b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : { \text{True, False} } )</td>
<td>( a : { \text{True, False} } )</td>
</tr>
<tr>
<td>( x : [0,100) )</td>
<td>( x : [0,100) )</td>
</tr>
<tr>
<td>( x &lt; 20 )</td>
<td>( a \land \neg x &lt; 20 )</td>
</tr>
</tbody>
</table>
This can be generalized for clauses of any length, such that the amount of subproblems per clause is the amount of literals, instead of $2^{\text{amount of literals}} - 1$. To do so, replace lines 3, 4 and 5 in CRS with the following function:

\[
\textbf{definition subproblems}(\text{clause}): \\
\begin{align*}
1: & \quad \mathcal{P} \leftarrow \{\} \\
2: & \quad \text{for literal}_i \text{ in clause do} \\
3: & \quad \text{subproblem} \leftarrow \{-\text{literal}_j \text{ for literal} \text{ in clause where } j < i\} \\
4: & \quad \text{subproblem} \leftarrow \text{ literal}_i \\
5: & \quad \mathcal{P} \leftarrow \text{subproblem} \\
6: & \quad \text{return } \mathcal{P}
\end{align*}
\]

Secondly, often in problems groups of variables can be recognized that do not influence other groups. Such groups can be given to a CRS algorithm separately. Take for example the problem 5Vars. Independent variable groups that can be recognized, are $\{a, b, c\}$ and $\{d, e\}$. There are two clauses, the first has 3 subproblems, the second 2. This gives 6 worlds in total. But since the amount of solutions in $\{a, b, c\}$ is unrelated to the amount of solutions in $\{d, e\}$, we can separate them. 5Vars can be split up in two subproblems, one for each variable group, each containing only the clause(s) that influence their group. These can each be split up in their subproblems, giving a total of 5 worlds. The difference gets larger when there are more literals.

<table>
<thead>
<tr>
<th>5Vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$ : {True,False}</td>
</tr>
<tr>
<td>$b$ : [0,100)</td>
</tr>
<tr>
<td>$c$ : [0,100)</td>
</tr>
<tr>
<td>$d$ : [0,100)</td>
</tr>
<tr>
<td>$e$ : [0,100)</td>
</tr>
<tr>
<td>$a = \text{True} \lor b + c &gt;= 100 \lor b + c &lt; 50$</td>
</tr>
<tr>
<td>$d = 0 \lor e = 0$</td>
</tr>
</tbody>
</table>

The algorithm that separates these independent subproblems is quite simple:

\[
\textbf{definition separate_independent_subproblems}(\text{state}): \\
\begin{align*}
1: & \quad g \leftarrow \{\} \\
2: & \quad \text{for constraint in constraint do} \\
3: & \quad a \leftarrow \{\mathcal{V}_i \text{ in clause}\} \\
4: & \quad \text{if } \exists b \in g, a \cap b \neq \{\} \text{ then} \\
5: & \quad b \leftarrow a \\
6: & \quad \text{else} \\
7: & \quad g \leftarrow a \\
8: & \quad \text{for } \mathcal{V}_i \in \mathcal{V} \text{ do} \\
9: & \quad \text{if } \exists a \in g, \mathcal{V}_i \in a \text{ then} \\
10: & \quad g \leftarrow \{\mathcal{V}_i\} \\
11: & \quad \text{return } g
\end{align*}
\]

This algorithm is also used in Unigen2[6], although there it works over CNF with boolean literals.

And lastly a tactic that we already implicitly applied. We could rewrite literals of the form $expression_1 \neq expression_2$ to $expression_1 < expression_2 \lor expression_1 > expression_2$, thereby possibly discovering smaller ranges at the cost of an extra or, but decide that this rewrite is
not worth the effort, since in most situations it would at most restrict just one value. Instead, we ignore \( \neg \)-literals when range shrinking and tethering (but not when sampling).

**Comparison with RACE**

A form of range shrinking is used in RACE[4]. In our version, range shrinking and tethering are purposely kept simple and in polynomial time. The form of range shrinking used in RACE would propagate ranges, so that for example in **problem3** it would recognize that \( x \) too could have its range shrunk to \([10, 100]\). **problem10** is an adapted problem from their paper, where they show the disadvantage of range propagation: long propagation sequences. When using range propagation, one would notice that because of the second literal, the highest value of \( a \) is 14. When propagating that to the first literal, the lowest value of \( b \) becomes 1, which makes the highest value of \( a \) 13, etc, until both ranges are empty. This takes a lot of steps. RACE applies certain unspecified metrics to recognize and prevent such propagation sequences.

Tethering would notice that the first literal can be rewritten to \( a > 14 - b \) and the second to \( a < 15 - b \), and that therefore there is no valid value for \( a \). To my knowledge, RACE does not apply (a form of) tethering.

**Performance**

I have not implemented the tactics. So we will not test them. Instead in this section we will reason about the performance.

When a problem has no logical ors, range shrinking runs in \( O(\mathcal{V} \ast \text{literals}) \), and tether in \( O(\mathcal{V}^2 \ast \text{literals}) \). When given certain problems, they can reduce the assignment space to the solution space. When that is the case, iteratively_halve runs in \( O(\mathcal{V}) \), the cost of one rejection sample. When the assignment space is sparse, iteratively_halve will run in \( O(\log_2(\text{space}) \ast \mathcal{V}) \), since per iteration the assignment space gets halved. In the worst case, iteratively_halve will run in \( O(\text{space} \ast \log_2(\text{space}) \ast \mathcal{V}) \), where parts always contain solutions, but rejection sampling cannot find one. But in the problems of chapter 3, solutions are never spread out over the assignment space, and are instead close to each other. So we will assume iteratively_halve to run in \( O(\log_2(\text{space}) \ast \mathcal{V}) \).

The number of variables, as well as the number of literals is in most problems vastly smaller than the space size, which is in \( O(d^\mathcal{V}) \), where \( d \) is the average domain size.

So, if a given problem has no logical ors, and if an assignment space is sparse, and if an assignment space can through reduction tactics be reduced to its solution space, reduction tactics change the running time from \( O(\log_2(d) \ast \mathcal{V}^2) \) to \( O(\mathcal{V}^2 \ast \text{literals}) \). When domain sizes are large, this can be a large speed improvement. In case we only apply range shrinking and not tethering, the running time is \( O(\mathcal{V} \ast \text{literals}) \).

But when a problem does contain logical ors, the amount of subproblems created is in \( O(o^a) \), where \( o \) is the average amount of literals in a clause, and \( a \) is the amount of clauses. The complexity of applying both reduction tactics is then \( O(\mathcal{V}^2 \ast o^a) \), which can get large rather quick.

When translating a list to formulas with expression relation literals, a number of clauses is needed in translation related linearly to the maximum length \( n \) of the list, where every clause has size 2.
Same for translating a tree, except that every clauses has a size also related linearly to $n$. See the translation at the beginning of this chapter.

So the complexity of applying both reduction tactics on problems with lists is $O(\mathcal{Y}^2 \cdot 2^n)$ and for trees $O(\mathcal{Y}^2 \cdot n^n)$. Therefore the reduction tactics are not suited for recursive ADTs, since they are more likely to increase the running time than decrease it.

In chapter 3 we saw a number of problems. They can be split up in three categories: common problems, rare problems (of which I found no evidence that they were used, but which TorXakis still should be able to handle), and specific problems used by specific users.

The common problems are no constraint, equality, range, properties, length and struct.

The rare problems are add and tree.

The specific problems are ranged last, 2snake e.a. and circle.

Let us look for those problems at whether the reduction tactics actually reduce the assignment space, and whether they reduce the running time.

In the problem no constraint the assignment space is already equal to the solution space. Reduction tactics will only take unnecessary time. So too for struct and tree.

In the problems equality and range range shrinking reduces the assignment space directly to the solution space. If we turn tethering of, we can find a solution in $O(\mathcal{Y} \cdot \text{literals})$, which should be much faster than $O(\log_2(d) \cdot \mathcal{Y}^2)$.

In the problem properties, tethering can tether empty to the first child variable of $l$, thereby halving the assignment space. The rest of the constraint cannot be read for reduction tactics, since both other literals make use of recursive functions. Halving the assignment space is in no way worth the runtime cost of translating and tethering, especially since the problem contains a list.

The reduction tactics can reduce nothing in length, add and circle. Length uses a recursive function, and add and circle use too difficult to parse arithmetic.

<table>
<thead>
<tr>
<th>properties</th>
<th>ranged last</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$ : boollist</td>
<td>$x$ : list</td>
</tr>
<tr>
<td>length : Integer</td>
<td>list = Nil</td>
</tr>
<tr>
<td>all : Boolean</td>
<td>Cons(hd = [0, 100], tl = list)</td>
</tr>
<tr>
<td>empty : Boolean</td>
<td>$lsNil(lsCons(lsCons(lsCons(x))))$</td>
</tr>
<tr>
<td>boollist = Nil</td>
<td>hd$(x) = 1$</td>
</tr>
<tr>
<td>$\quad$ Cons(hd = Boolean, tl = boollist)</td>
<td>hd$(tl(x)) = 2$</td>
</tr>
<tr>
<td>length$(l)$ : $\text{if isCons}(l) \text{ then } 1 + \text{length}(tl(l)) \text{ else } 0$</td>
<td>$20 \leq \text{hd}(tl(tl(x))) &lt; 100$</td>
</tr>
<tr>
<td>all$(l)$ : $\text{if isCons}(l) \text{ then } \text{hd}(l) \land \text{all}(tl(l)) \text{ else True}$</td>
<td></td>
</tr>
<tr>
<td>length = length$(l)$</td>
<td></td>
</tr>
<tr>
<td>all = all$(l)$</td>
<td></td>
</tr>
<tr>
<td>empty = lsNil$(l)$</td>
<td></td>
</tr>
</tbody>
</table>

In the problem ranged last, range shrinking can heavily reduce the assignment space from about $100^n$ (every element of the list can take a value in $[0,100)$, the list can have any length $\leq n$), to 80 (only the third element is undecided, and its range is [20,100]), thereby reducing the assignment space to the solution space. It does so at a cost of $O(\mathcal{Y} \cdot 2^n)$. This while iteratively_halve finds a solution in a space of size $100^n$ in $O(\log_2(100) \cdot n \cdot \mathcal{Y})$. So it is not worth the effort. 2snake e.a.
contains a number of variables that can all be tethered to other variables, heavily reducing the assignment space in problems with higher numbers (think 8snake and higher). However, tethering is in $O(y^3)$ (the amount of literals is linearly related to the amount of variables), which is higher than $O(\log_2(d) * y^2)$.

So tethering on the problems from chapter 3 does not give a speed improvement. Range shrinking can give a speed improvement when well implemented and when ran mostly on problems similar to equality and range. For some users this is exactly what their problems look like.
9. Semantic additions to the TorXakis specification language

Without changes to the TorXakis specification language, the language allows infinite recursions and infinite int variables, and has no way for the user to show desire for uniformness over partial solutions.

In this chapter I will change the syntax and semantics of the TorXakis specification language to correct this, so that it matches the expectations set in chapter 3.

There are three main areas that are interesting for changes. Those are the problem specification, the ADT specification and the parameters. The problem specification and ADT specification are written in a file. The parameters can be adjusted when running the tool. We want to keep all files compatible, so changes to the constraint specification and ADT specification should be backwards compatible.

Finite bounds for integer variables

Currently such finite bounds can be defined as constraints in the problem specification. That is not ideal, since they are then not given separately from other constraints, and they are easy to forget for the user. A more natural place is at the ADT specification. For this we add the following syntax, the rangedef:

```
RANGEDEF SmallInt ::= [0,100) ENDDEF
```

With this syntax one can define versions of integers that have a shorter range, which can then be used in other ADTs or as domain in the problem specification. However, for backwards compatibility reasons, integers that are not bound by such a rangedef are still allowed. That is why we add a parameter defining bounds for integers that are not defined with a rangedef. They have no effect on integers that are defined with a rangedef.

Finite bounds for recursion

Currently finite bounds for recursion can be defined as a constraint by using a recursive function. That is not ideal, for the same reasons as for the bounds over integer domains. A more natural place is the ADT specification. To accomplish that, we allow ranges to be added to recursive ADT specifications. Here is an ADT of an IntList as it would be defined currently:

```
TYPEDEF IntList ::= 
  Nil 
  |  Cons { hd :: Int ; tl :: IntList }
ENDDEF
```

And here is the same ADT with added bounds:

```
TYPEDEF IntList [0,10) ::= 
  Nil
```

The amount of recursions the IntList may take is now limited to 9.
The only place such a range is allowed, is after the name of a recursive ADT when defining it.
A parameter is added for bounds over recursive ADTs that do not have such a range defined.

Strings are a special kind of lists, since they are predefined. We add the following syntax to the
ADT, the stringrangedef, and a parameter for strings that are not defined using that keyword.

**STRINGRANGEDEF**

SmallString ::= [0,4) ENDDEF

**Ways for the user to show desire for uniformness over partial solutions**
Currently, every generated value is sent to the SUT. I propose a new step symbol (let us call it >S<)
which splits value generation in parts before it sends the last generated value to the SUT. Let us
write the appropriate pieces of TorXakis syntax to define **Decision** (only relevant parts of the
specification shown here):

```torxakis
TYPEDEF Decision ::= 
    Decision { a :: Boolean 
                   ; x :: Int 
               } 
ENDDEF

Decision ?d [[a(d) OR x(d)>=20]]
```

Since there are more solutions when \( a \) is False, this will give preference for \( a = \) False. To give both
values of \( a \) equal probability, we can define the last line as follows:

```torxakis
Boolean ?b [[ ]] >S> Decision ?d [[ a(d)==b AND (a(d) OR x(d)>=20) ]] 
```

This piece of code first generates a Boolean \( b \) without constraints, and then generates a Decision \( d \),
where \( a \) takes the value of \( b \). Since \( d \) is the last value generated, it is given to the SUT.
A limitation of this approach is that the user should be sure that for every value of the variables in
the first step of generation, there should be at least one solution. Otherwise, when a partial
assignment is chosen for which no solutions exist, no solution can be found.
An advantage of this approach is its relatively clear syntax.

Here is another example where \( a \) is True ten times more often than False.

**RANGEDEF**

OneInTen ::= [0,11) ENDDEF

OneInTen ?i [[ ]] >S>

52
Decision \[ a(d) = (i = 0) \land (a(d) \lor x(d) \geq 20) \]

And this piece of code then means the following. In the assignment space, \( a \) will be true 9 times more often than false.

Instead of adding \( >S< \), another way to give users ways to show desire for uniformness over partial solutions is to use the place of variables within the ADT structure as the evaluation order metric. This is however not advisable, and therefore not pursued. See struct for an example.

<table>
<thead>
<tr>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x : )</td>
</tr>
<tr>
<td>person =</td>
</tr>
<tr>
<td>favorites =</td>
</tr>
<tr>
<td>food =</td>
</tr>
<tr>
<td>drink =</td>
</tr>
</tbody>
</table>

Since \( name \) is at a higher place in the structure than \( food \), it would be generated before \( food \). This approach gave a lot of problems. One normally uses the ADT structure for clarity purposes and structuring the data. By giving it an extra meaning, one now needs to change the structure to accommodate it, creating unclear code. Tactics to enforce certain orders include doubly nesting parts of the ADT, something that looks very unclear.

It can also give strange results, assuming clients were not constantly keeping the generation order in mind. For example, suppose one discovers he/she wants to give a coordinate to the SUT instead of an int (where a coordinate exists out of two ints). He/she replaces within an ADT the int with a coordinate, which he/she defines as its own ADT of two ints. Now the ints generated in the coordinate ADT are a level lower than they used to be, thereby possibly changing the generation order.
Parameters
Here is an overview of parameters I would add. Users can change these parameters.

Ranges when no range defined at ADT or function level:
- Int_lower_bound: \(-2^{31}\)
- Int_higher_bound: \(2^{31}\)
- Rec_ADT_lower_bound (must be positive): 0
- Rec_ADT_higher_bound (must be positive): 100
- String_lower_bound (must be positive): 0
- String_higher_bound (must be positive): 100
Iteratively_halve: True
Samples_per_iteration (must be positive): 10
Range_shrinking: False
Tethering: False
Separate_independent_subproblems: True

The ranges are for when variables in a file have no bounds specified. If iteratively_halve is turned off, the base algorithm becomes rejection sampling instead. When both range_shrinking and tethering are turned off, no translation to formulas with relative expression literals is done.
10. Conclusion
The goal of this thesis is to find a better CRS algorithm for TorXakis than the current TorXakis algorithms. I have defined 'better' as fulfilling a list of expectations:
1. the LCB of the algorithm should not exceed 2.
2. users should be able to define uniformity over partial solutions, and be able to define their own distributions.
3. all assignment spaces should be finite.
4. users should be able to define the maximum amount of nodes a recursive ADT can take.
5. when given a problem containing a list, the algorithm should choose uniformly over the number of nodes in the list, instead of having a preference for longer lists.
6. the algorithm is efficient on the problems shown in chapter 3.

Expectations 2, 3 and 4 are met by semantic additions to the TorXakis specification language. I have shown that the algorithms in TorXakis, although fast, do not meet expectation 1, except for Partition with part size set to 1, which is then slow.
I have listed the current best CRS algorithms, and decided a better algorithm could be made.

I introduced the algorithm iteratively_halve and shown through test runs that it is uniform and efficient on the problems shown in chapter 3. It also does not have a preference for longer lists. I introduced two reduction tactics, range shrinking and tethering, but shown that tethering does not improve running time on common problems, and range shrinking only on specific problems (problems similar to equality and range).
11. Future research

There are a lot of possible paths towards improvements of the CRS algorithm. Many of those improvements need heavy research.

The directions of these paths are:
- Towards separating reduction tactics from sampling
- Towards easier to parse lists
- Towards better reduction tactics over complicated expressions
- Towards a larger understanding of typical clients problems
- Towards incorporation of new CRS tactics

Towards separating reduction tactics from sampling

In TorXakis one specifies a model as a labeled transition system of problems. When sampling, we look at one such problem. In many cases when walking over the transition system, we will visit the exact same problem multiple times. In those cases, we would have to do the preprocessing only once. Let us first use the assumption that problems do not use information from the SUT (information from the SUT however can be used to decide which path in the transition system to take), and look later when this assumption is not true.

If problems do not use information from the SUT, we can do reduction tactics before running tests. This is ideal in cases where the same problem occurs multiple times, and where the same TorXakis specification has to be used a number of times over multiple versions of a SUT.

However, this assumption, that problems do not use information from the SUT, is often not true. The SUT outputs values that can be used as terms in expressions in following problems. However, tethering can perfectly handle yet undefined terms. For example, if there is a constraint of the form \( x = \text{term} \), where \( \text{term} \) is as of yet unknown, tethering can still change the range of \( x \) to \([\text{term}, \text{term} + 1]\). In how far this trick can be extended to cover all possible uses of terms from the SUT in following problems, is to be seen, but it is without a doubt true that at least a part of preprocessing can be done well before visiting a problem, which can give drastic improvements when that problem is visited multiple times.

Towards easier to parse lists

The TorXakis specification language allows lists to be specified as recursive ADTs. This has the benefit that trees too can be specified. In the given clients problems, there was no use of trees, and ample use of lists. That does not necessarily mean that trees are not used, but it does show that lists are used and are considered important.

Let us look at some typical functions that could occur when working with lists.

\[
\begin{align*}
\text{FUNCDEF } \text{length}(l :: \text{IntList}) :: \text{Int} ::= \\
& \quad \text{IF } \text{isNil}(l) \\
& \quad \quad \text{THEN } 0 \\
& \quad \quad \text{ELSE } 1 + \text{length}(\text{tl}(l)) \\
& \quad \text{FI} \\
\end{align*}
\]

\[
\text{FUNCDEF } \text{all}(l :: \text{BoolList}) :: \text{Bool} ::= 
\]

56
isNil(l) \lor (\text{hd}(l) \land \forall(l))

ENDDEF

\text{FUNCDEF set\_to\_index}(l :: \text{IntList}, \text{length} :: \text{Int}) :: \text{Bool} ::= 
\begin{align*}
    \text{IF} & \quad \text{isNil}(l) \\
    \text{THEN} & \quad \text{TRUE} \\
    \text{ELSE} & \quad \text{hd}(l) == \text{length} \land \text{set\_to\_index}(\text{tl}(l), \text{length} + 1) \\
    \text{FI}
\end{align*}

ENDDEF

Here \text{length} calculates the length of a list, \text{all} checks whether a list of booleans only contains trues, and \text{set\_to\_index} sets every element in a list to its index. These functions could have been written in a number of ways, but whichever way one writes them, the functions would always need to be recursive and would always need to check per recursive call whether the given list is empty. Because of this, those functions are hard to parse for reduction tactics, since it is hard to estimate when a recursive function halts. A change to the TorXakis specification language, such that constraints over all elements of a list, access to the length of a list, access to variables at certain indexes, and access to the index of an element can be described in a standardized way, so that they can be parsed with ease, would be a great improvement.

If one does not want to change the TorXakis specification language, one could look at trying to find structure in functions over lists, and use that structure. However, those functions can be written in a number of ways, so one would need to standardize which forms such functions may take. How this would look exactly I do not know. Notice that for example, that if we were to express \text{2snake} e.a. as a list, we cannot express its constraint using a constraint over all elements of a list, since it has a constraint over pairs of elements.

Towards better preprocessing of complicated expressions

Range shrinking creates a bounding box around the solution space (or a bounding x-dimensional cube when there are x dimensions). And for most of our clients problems, this works surprisingly well. A clients problem where this does not work well, is 2snake (or 3snake or 4snake). The assignment space of 2snake can be drawn as follows (where grey means valid):
The assignment space of 3snake can be drawn as follows:

One can imagine what a possible 4snake, 5snake, etc. would look like: a multidimensional thin diagonal line. Even though the bounding box is minimal, the ratio of solutions/assignments is rather low. In this thesis we solved this with tethering. But other expressions are imaginable, where range shrinking and tethering have no effect. For example the problems Add and Add3.

The space of Add looks like:
The space of Add3 looks like:

Those are 2- and 3-dimensional orthogonal simplexes. How to sample from an orthogonal simplex is addressed in [15].
Using orthogonal simplexes, one can encapsulate interesting expressions tighter. Notice that we do not need a minimal encapsulation, just one in which the solution space fits.
Pursuing preprocessing tactics that use orthogonal simplexes can be useful when problems often contain expressions with large amount of variables.

Towards a larger understanding of typical users problems
In chapter 3 we saw a collection of typical problems. This was received through word-of-mouth. For privacy reasons I had no access to TorXakis files that are actively used. It could be worthwhile to recheck whether this collection can be extended.

Towards incorporation of new CRS tactics
It is still very possible that some major breakthroughs will be achieved in the next years, which can be utilized within TorXakis.
12. References

[1]: (October 2018). TorXakis: A tool for Model Based Testing. Retrieved from https://github.com/TorXakis/TorXakis


