Interaction Coverage meets Path Coverage by SMT Constraint Solving

Wolfgang Grieskamp[1], Xiao Qu[2], Xiangjun Wei[1],
Nicolas Kicillof[1], and Myra B. Cohen[2]

[1] Microsoft Corporation
[2] University of Nebraska, Lincoln, USA

Abstract. We present a novel approach for generating interaction combinations based on SMT constraint resolution. Our approach can generate maximal interaction coverage in the presence of general constraints as supported by the underlying solver. It supports *seeding* with general predicates, which allows us to combine it with path exploration such that both interaction and path coverage goals can be met. Our approach is motivated by the application to behavioral model-based testing in the Spec Explorer tool, where parameter combinations must be generated such that all path conditions of a model action have at least one combination which enables the path. It is applied in a large-scale project for model-based quality assurance of interoperability documentation at Microsoft.

1 Introduction

Combinatorial interaction testing (CIT) generates test cases that contain a specified set of combinations of parameter values for testing. The most common type of CIT for testing is pair-wise CIT where all 2-way combinations of parameter values are tested together in at least one test case [3]. CIT serves to reduce the full Cartesian product space of a set of parameters, which may be extremely large in real-world applications.

In a model-based testing tool in the style of Spec Explorer [10], CIT can play a role for generating parameter combinations for actions of the system-under-test. In these approaches, a labeled transition system or finite state machine is constructed from a set of guarded state update rules, where each model rule is associated with a parameterized action. The model rule is executed using symbolic path exploration, seeding parameter assignments for testing which cover all paths of the model. However, the implementation may have black-box behavior which is over-abstracted in the model, creating the need for generating more parameter combinations than needed for covering the model. This gap can be closed by combining path exploration with CIT. Thereby, the challenge is to come up with a CIT generator which aims at fulfilling the interaction coverage goal *and* the path coverage goal.

In this paper, we present a novel CIT algorithm which addresses this challenge. Our solution is integrated into the Spec Explorer model-based testing tool [10, 11, 13] version 2, and is driven by application requirements in protocol testing for Microsoft's interoperability program [12]. In protocol testing, actions represent messages sent via the network, which often have many different parameters. The implementation is naturally black-box, and not all its behavior is revealed in the model, therefore creating the

need to use different parameter combinations in tests beyond those that can be derived from the behavioral model.

The solution we have developed differs significantly from existing ones. Leveraging the performance and feature set of an SMT solver, we have devised an algorithm which generates interaction combinations solely based on constraint resolution and model enumeration as provided by the constraint engine. In this work we use the SMBT solver Z3 [9]. While other SMT solvers may also suffice we have not experimented with this. The algorithm works in reverse to existing algorithms (e.g. [8]): instead of constructing combinations bottom-up, it enumerates combinations using the solver in a top-down fashion, starting with the largest partitions of interactions. While this approach requires some heuristics to overcome scalability issues, the investigation of which is still preliminary, we have clearly identified the core of a novel algorithm, which will guarantee the maximum interaction coverage obtainable, while also satisfying general constraints and path coverage seeding. In our initial benchmarks, this algorithm performs comparatively well, and it is also used already on a daily basis for interoperability testing.

2 Motivation

In order to motivate the work we begin with an example. Suppose we have a highly abstracted file server. The file server provides two operations: write content to a file and read content from a file. **Write** takes several parameters. Its effect depends on the current state of the file system on the server. **Read** extracts the current file content. Part of the modeling and testing problem is to predict which content to expect after one or more write operations have happened.

A problem like this can be modeled in Spec Explorer using one of its supported notations, C#. The model is given in Fig. 1. It defines a global state variable `fileSys` which models the current state of the server as a mapping from file names to file contents. It next defines two actions and their state transition rules as C# methods. In `Write(doOverride, doAppend, name, content)`, the `doOverride` parameter indicates whether an existing file is allowed to be overridden. The `doAppend` parameter specifies whether content should be appended to an existing file. The remaining parameters describe the file name and content to write. `Read(fileName)` delivers the content of the file in the current state, as predicted by the model. We have used the `Contracts.Requires` statement as an enabling condition for this action, which excludes attempts to read files with unknown names.

For the general approach on how such models are executed and explored by Spec Explorer, we refer readers to the other publications about this technology ([10, 11, 13, 14]). For this exposition it should suffice to know that model exploration yields a transition system where each state represents the model state (in this case the file server content). Transitions are drawn from source to target states for all methods which carry the `[Action]` attribute *if* there exists a set of parameters such that the enabling condition is true in the source state. The target state then results from updates performed in the method.

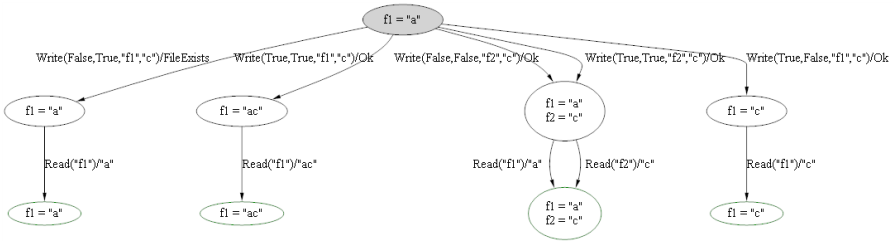
```
static MapContainer<string,string> fileSys;

enum ErrorCode { Ok, FileExists }

static ErrorCode Write(
  bool doOverride, bool doAppend, string fileName, string content)
{
  if (fileSys.ContainsKey(fileName))
    if (!doOverride) return ErrorCode.FileExists;
    else if (doAppend) fileSys[fileName] = fileSys[fileName] + content;
    else fileSys[fileName] = content;
  else
    fileSys[fileName] = content;
  return ErrorCode.Ok;
}

[Action]
static string Read(string fileName)
{
  Contracts.Requires(fileSys.ContainsKey(fileName));
  return fileSys[fileName];
}
```

Fig. 1. The File Server Model

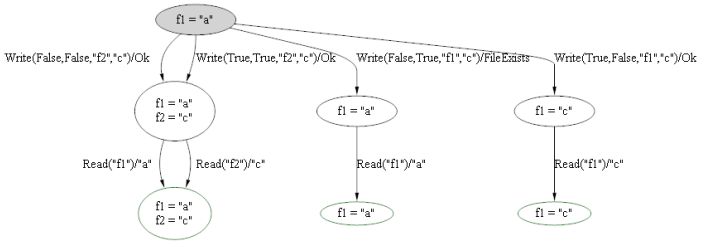
Fig. 2(a) shows the expected result of exploring this model (as constructed and rendered by Spec Explorer) with our approach to CIT, given the following settings (provided in a configuration file):

1. In the initial state, the server's file system contains one file named `"f1"`, the content of which is `"a"`;
2. The domain of values for the file name parameter of `Write` has been fixed to the set {`"f1"`,`"f2"`};
3. The domain of values for the file content parameter of `Write` has been fixed to the singleton set {`"c"`};
4. The three parameters `doOverride`, `doAppend`, and `fileName` are declared to be in pairwise interaction;
5. The allowed sequences of actions have been sliced to be just one `Write` followed by one `Read` using Spec Explorer's approach to model extraction and slicing (see e.g. [11])

Fig. 2(a) shows that five parameter combinations have been generated. In contrast, Fig. 2(b) shows only four combinations. The four combinations are in fact sufficient to cover all 2-way interactions of three parameters with the 2-ranked domains. However, as the model exploration graph shows, these four combinations do not cover all possible paths of the `Write` method: a combination is missing which covers the case where content is appended to an existing file. The reason is that the condition of this path actually represents a 3-way interaction: for to be path to be covered, a certain combination for the three parameters `doOverride`, `doAppend`, and `fileName` must be chosen, which will only by luck be generated when selecting 2-way interactions. On the other



(a) Expected parameter combinations



(b) Faulty parameter combinations

Fig. 2. Exploration Result of File Server Model

hand, having declared the problem as a full 3-way interaction in the first place, would have generated 8 combinations. As can be seen, this difference in the size of the solution set can easily explode when generalizing this observation to real-world samples.

Path Exploration and Seeding Spec Explorer (the newest version 2 – also called Spec Explorer for Visual Studio, [10]) performs model exploration by symbolic execution of the model code. In order to integrate CIT, we have established a suitable separation of concerns, which allows us to formulate the CIT problem independent from the model exploration problem.

Spec Explorer's model exploration is based on forward symbolic execution of a model rule, using the exploring runtime (XRT, [14]). Initially, all parameters are set to symbolic values. When execution proceeds and hits a branching point which depends on a symbolic value, Spec Explorer forks execution and continues with two paths, one assuming the branch condition to be true, and the other one assuming it to be false. It thereby effectively spawns a tree, where the leafs represent the final path condition as a conjunction of all branch conditions taken in that path, depending on the symbolic parameters. Loops are dealt with by pruning them according to code coverage and bounds.

For our example in Fig. 1, the path conditions of the **Write** operation are annotated in the code in Fig 3. Here, we assume execution in a state where the file system contains a single mapping of name `"f1"` to content `"a"`; in other states, the conditions may be different.

In order to generate combinations for parameters such that path conditions are respected, we not only need to provide the combination algorithm with interaction goals, but also to *seed* it with the set of path conditions, such that the generated combinations

```
if (fileSys.ContainsKey(fileName))
  if (!doOverride)
    return ErrorCode.FileExists; ⇝ fileName = "f1" ∧ ¬ doOverride
  else if (doAppend)
    fileSys[fileName] = fileSys[fileName] + content;
        ⇝ fileName = "f1" ∧ doOverride ∧ doAppend
  else
    fileSys[fileName] = content;
        ⇝ fileName = "f1" ∧ doOverride ∧ ¬ doAppend
else
  fileSys[fileName] = content; ⇝ fileName ≠ "f1"
return ErrorCode.Ok;
```

Fig. 3. Annotated Path Conditions of `Write`

satisfy each path condition at least once. Seeding has been introduced in other CIT algorithms [2, 3, 8], but concrete values as seeds are not sufficient for our purpose, we also need to be able to seed with general constraints.

Requirements for the CIT Algorithm From the observations above, the abstraction of the CIT algorithm in the context of the exploration problem is designed to take the following inputs:

1. A set of *interactions*, expressed as relations between parameters. The algorithm should generate the smallest set of value combinations it can find that includes the Cartesian product of each interaction relation (of variable strength).
2. A *constraint*: every solution the algorithm generates should satisfy the constraint. The constraint describes the possible overall solution set for the parameters, by giving for example ranges for individual parameters.
3. A set of *seeds* represented as predicates. There should be at least one solution satisfying each seed, provided it is consistent with the constraint.

Having such an algorithm, we can integrate well with the exploration. Assuming the user has defined the interaction relation, a general constraint that must be satisfied, and seeds, the overall exploration method including CIT is as follows.

1. Compute the symbolic transitions and path conditions of each action method in the current state (by symbolic execution, concolic execution, or any other method).
2. Enrich user supplied seeds by a seed for each path condition.
3. Compute the combinations from the user supplied interactions, enriched seeds, and constraint.
4. For every symbolic transition, instantiate it (including the target state it reaches) for each generated solution which satisfies the path condition.

This approach has worked well for integration of CIT into the Spec Explorer tool. We will not formalize the actual integration but focus on the CIT algorithm we have implemented to fit the procedure outlined above.

3 Solution

In this section, we formalize the CIT problem as a constraint problem, and derive the algorithm based on the constraint resolution engine Z3. In general CIT generation is an NP hard problem; algorithms use heuristics to minimize the size of solutions [3].

Background Let X be a given set of variables (symbols), V a set of primitive values, OP a set of primitive operations, and T a set of terms, as defined below:

$$
\begin{array}{lll}
x \in X & & \textit{variables} \\
v \in V & ::= \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots & \textit{values} \\
op \in OP & ::= \wedge \mid \vee \mid \neg \mid = \mid < \mid + \mid - \mid \ldots & \textit{operations} \\
f \in F & & \textit{fields} \\
t \in T & ::= x & \textit{variable} \\
 & \mid\ v & \textit{value} \\
 & \mid\ op(t_1, \ldots, t_n) & \textit{operation} \\
 & \mid\ [f_1 \Rrightarrow t_1, \ldots, f_n \Rrightarrow t_n] & \textit{tuple} \\
 & \mid\ t.f & \textit{selection}
\end{array}
$$

Primitive values, V, and operations, OP, support at least Booleans and integers, but other types (like floats, strings) may be supported as well without additional conceptual complications. Note that terms can be used to denote values as well as constraints, which we consider to be Boolean terms. Henceforth we will call a Boolean term a constraint. Tuples are used to represent structured values and have named fields $f \in F$. It holds that $[\ldots, f \Rrightarrow v, \ldots].f \equiv v$. We assume terms are well typed (for example, only Boolean terms are combined with logical operators $\wedge$, $\vee$) but omit the details of the type system here for the sake of readability.

The subset of terms which only consists of primitive values or tuple values is given as follows:

$$T_V \subseteq T ::= v \mid [f_1 \Rrightarrow v_1, \ldots, f_n \Rrightarrow v_n]$$

Given a value assignment $\sigma \in X \to T_V$, the evaluation of terms is described by the function $\mathsf{eval}_\sigma \in T \to T_V$ which maps every term into a value term. We omit the obvious definition. This function is defined as the usual homomorphism over the structure of terms, substituting variables with their assignment in σ, computing the result of applying primitive operations $op(v_1, \ldots, v_n)$ to values in dependency of the semantics of the operation, and reducing tuple field selections. We omit the obvious definition.

Given a set of variables $Y \subseteq X$, and a constraint c which is closed under Y (all variables it contains are in Y), the *models* of Y for c are the value assignments under which the following constraint is true:

$$\mathsf{M}(Y, c) = \{\sigma \in Y \to T_V \mid \mathsf{eval}_\sigma c = \mathsf{true}\}$$

We next define projections over model sets w.r.t. a given set of terms. Given $Y \subseteq X$ a set of variables, $M \subseteq Y \to T_V$ a set of models for it, and $S \subseteq T$ a set of terms closed under Y, we write $M \uparrow S$ to denote the evaluations of those terms under each model in M.

$$M \uparrow S = \{\mathsf{eval}_\sigma \lhd S \mid \sigma \in M\}$$

In the above notation, $f \lhd S$ denotes the domain restriction of a function to the set S ($f \lhd S = \{(x, y) \in f \mid x \in S\}$).

With help of projection we can easily describe the exhaustive Cartesian combination of values for a set of terms as extracted from a model set. For example, if $Y = \{x, y\}$, $c = 0 \le x \le 1 \wedge 0 \le y \le 1$, and $S = \{x + 1, y + 1\}$, then $\mathsf{M}(Y, c) \uparrow S = \{\{x+1 \mapsto 1, y+1 \mapsto 1\}, \{x+1 \mapsto 1, y+1 \mapsto 2\}, \{x+1 \mapsto 2, y+1 \mapsto 1\}, \{x+1 \mapsto 2, y+1 \mapsto 2\}\}$.

CIT Problem Having defined the notions of terms, constraints, and models, we can now define CIT with generalized seeds.

A CIT problem with seeds (in the remaining of this paper, CIT problem for short) is a tuple $\Pi = (P, I, S, c)$, where $P \subseteq X$ is a set of variables, $I \subseteq \mathbb{P}\,T$ is a set of sets of terms, called the interaction relation, $S \in \mathbb{P}\,T$ is a set of constraints, the seeds, and $c \in T$ a general constraint. All terms (constraints) appearing in interactions, seeds, and the general constraint are closed under P, i.e. do not contain other variables than P.

An interaction $i \in I$ describes a set of terms which are declared to be in mutual interaction. For example, if we have parameters x, y, z, and they should be in pairwise interaction, we will have $I = \{\{x, y\}, \{y, z\}, \{z, x\}\}$. Note that we do no restrict interactions to be just built from variables: they can be arbitrary terms. If the problem is to express pairwise interaction between the fields of tuple x with three fields a, b, c, we will have $P = \{x\}$ and $I = \{\{x.a, x.b\}, \{x.b, x.c\}, \{x.c, x.a\}\}$. And if the problem is to express pairwise interactions between the three first bits of a number x representing a bit mask, we will have $P = \{x\}$ and $I = \{\{x\&1 \neq 0, x\&2 \neq 0\}, \{x\&2 \neq 0, x\&4 \neq 0\}, \{x\&4 \neq 0, x\&1 \neq 0\}\}$.

A CIT algorithm $Alg(\Pi)$, with $\Pi = (P, I, S, c)$, produces a set of models for P. While a unique model set for the interaction combination problem does not exist in general, we can characterize the requirements for $Alg(\Pi)$ as follows:

- All models satisfy the constraint c and the disjunction of the seeds in S:

$$Alg(\Pi) \subseteq \mathsf{M}(P, c \wedge \vee S) \tag{1}$$

- For every seed $s \in S$, if there exists a model which satisfies $c \wedge s$, then at least one such model must be contained in the solution set:

$$\mathsf{M}(P, c \wedge s) \neq \varnothing \Rightarrow Alg(\Pi) \cap \mathsf{M}(P, c \wedge s) \neq \varnothing \tag{2}$$

- For every interaction $i = \{t_1, \ldots, t_n\} \in I$, the models for the t_i should be exhaustively combined:

$$\mathsf{M}(P, c \wedge \vee S) \uparrow i = Alg(\Pi) \uparrow i \tag{3}$$

Solver Our implementation uses the SMT solver Z3 [9], based on the satisfiability-modulo-theory method. The constraints produced by Spec Explorer's path exploration include encodings of the full .NET type system with numerous basic types and object references, as well as special types for modeling as symbolic recursive free data types and collections. While these domains can be in principle passed on to other solvers supporting a sufficiently expressive formalism, Z3 provides a match for our application as it is particularly targeted to solving such domains efficiently.

Assume the solver works on terms as have been described before, where constraints are represented as Boolean terms. The abstraction of the solver we use in this paper provides one single entry point: given a constraint c over variables $Y \subseteq X$, $\mathsf{solve}(c)$ delivers either the constant fail, indicating that it did not found a solution for the variables in c, or a model $\sigma \in Y \to T_Y$ which gives evidence that the constraint can be evaluated to true with the given variable assignment.

In order to have a solver exposing such an API enumerate all solutions for a term t under a constraint c, a loop as the one below is typically used:

given c, t
while $\sigma = \mathsf{solve}(c) \neq \mathsf{fail}$
    output $\mathsf{eval}_\sigma(t)$; $c := c \wedge \neg\,(t = \mathsf{eval}_\sigma t)$

Z3 is efficient at enumerating solutions this way. For a constraint of small to medium size, which is common for our relatively localized problem of generating parameter combinations, Z3 can enumerate thousands of solutions in a matter of seconds. While the solver is based on heuristics in order to deal with problems which are generally exponential, in current applications we rarely encountered situations where those failed.

Algorithm The algorithm we developed appears to be straightforward in comparison to other algorithms for generation of interaction combinations. It is based on the idea that the solver can be leveraged during enumeration to not only exclude complete solutions previously seen (as we have shown above), but also partial ones which represent combinations for interactions.

Suppose $\mathit{\Pi} = (P, I, S, c)$ a CIT problem. We construct *partitions* of interactions from the power set $\mathbb{P}\,I$ and process them with decreasing cardinality (please note that the notion of partition used here is not the same as in set theory, but just means a subset of $\mathbb{P}\,I$ – we borrow the definition from other CIT work). For each of the partitions, we enumerate the solutions and exclude the full solution as well as each individual interaction combination in that solution. We continue until partitions have cardinality one. This process ensures that we first attempt to put as many interactions as possible into one solution.

As an example, let $I = \{i_1, i_2, i_3\}$. We first generate solutions for partition $\{i_1, i_2, i_3\}$, trying to fit as many interactions as we can. We continue then with partitions $\{i_1, i_2\}$, $\{i_2, i_3\}$, and $\{i_1, i_3\}$. If we have not yet covered all interactions, we will try $\{i_1\}$, $\{i_2\}$ and $\{i_3\}$, which will finally ensure that we have reached the maximum possible coverage. To exclude solutions repeating interaction on the same partition level, we add an exclusion constraint for each of the interactions of the given partition. Let $\mathit{part} = \{\{x, y\}, \{y, z\}\}$, and $\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$. Then the exclusion constraint for the entire solution and for the partition will be

$$\neg\,(x = 1 \wedge y = 2 \wedge z = 3) \wedge$$
$$\neg\,(x = 1 \wedge y = 2) \wedge \neg\,(y = 2 \wedge z = 3)$$

In order to deal with seeds, we inject their processing into the processing of each partition. Maintaining a set of seeds which have not yet been covered, we try on each partition first to cover any of the seeds, before allowing generation of solutions for seeds which are already covered. If any seeds remain at the end that have not been covered with any partition, we will generate a solution for them. (This is possible since we may stop iterating partitions early, when all interactions have been covered.)

Of course, the full power set of I, which stands for all partitions, may be huge ($2^{\#I}$). However, we can use the solver to check whether enumeration of solutions for one partition has closed some interactions, i.e. fully covered them. Every subsequently processed partition containing a closed interaction can be then skipped. This drastically reduces the number of partitions that actually require solution enumeration for the average case.

```
# Input and Variables
given Π = (P, I, S, c)
var φ := c ∧ (∨ S)
var coverage ∈ I → ℙ(T → T_V) := {i ↦ ∅ | i ∈ I}
var open ∈ ℙ I := I
var seeds ∈ ℙ S := S

# The algorithm
proc Algo
    # process partitions
    for card := #I downto 1
        var parts := {part ∈ ℙ open | #part = card)
        while parts ≠ ∅
            let part = choose(parts); parts := parts \ {part}
            if SolvePartition(part)
                parts := {part ∈ parts | part ⊆ open}
            else
                parts := prune(parts, part)
    # process uncovered seeds
    foreach s ∈ seeds
        Solve(s)

# Generate solutions for one partition
proc SolvePartition(part)
    var progress := false
    # attempt to cover seeds in this partition
    foreach s ∈ seeds
        if Solve(ExcludeCovered(part) ∧ s)
            seeds := seeds \ {s}; progress := true
    # enumerate all remaining solutions of this partition
    while Solve(ExcludeCovered(part))
        progress := true
    return progress

# Generate solution for a given constraint
proc Solve(τ)
    ifσ = solve(φ ∧ τ) ≠ fail
        output σ # output solution
        φ := φ ∧ Exclude(σ)
        foreach i ∈ open
            coverage(i) := coverage(i) ∪ (σ ↑ i)
            if solve(φ ∧ ExcludeCovered({i}) = fail)
                # no more solutions, close interaction
                open := open \ {i}
        return true
    else return false

# Construct exclusion constraint for interactions
proc ExcludeCovered(interactions)
    return ∧ {Exclude(f) | i ∈ interactions, f ∈ coverage(i)}

# Construct exclusion constraint for term assignment
proc Exclude(assignment)
    return ¬ ∧ {t = t' | (t, t') ∈ assignment}
```

Fig. 4. Algorithm

Nevertheless, the situation can arise where the algorithm reaches a "plateau": that happens if for a given partition, no solution at all is found. The algorithm may search for a long time (and virtually not terminate) to find partitions that would contribute to the solution set and eventually close interactions. In this case, a heuristic to prune some of the partitions of higher cardinality is applied. As partitions of cardinality one are never pruned, the heuristic only affects the size of the solution set, by potentially producing more solutions than required, but never its correctness.

The algorithm is given in pseudo-code in Fig. 4. It is based on two primitives that represent heuristics, over which it is parameterized: choose selects the next partition to process from a set of partitions of equal cardinality, and prune prunes the set of partitions of equal cardinality in case a plateau is reached.

The primitive $\mathsf{choose}(\mathit{parts})$ chooses which partition to process next from the set of partitions *parts*. A good choice may influence efficiency, and decrease the chance of running into a plateau. In our current implementation of the algorithm, this has not been exploited, and the choice is arbitrary.

The primitive $\mathsf{prune}(\mathit{parts}, \mathit{part})$ represents the heuristic to prune partitions in case the algorithm has reached a plateau. Currently, we have implemented a very simple heuristic, which will just skip the entire cardinality level:

$$\mathsf{prune}(\mathit{parts}, \mathit{part}) = \varnothing$$

A deep investigation of heuristics has not been conducted yet for our approach, leaving space for improvement. However, even with the most simple heuristics as currently implemented, the algorithm performs well in comparative benchmarks and in everyday applications.

Illustration We illustrate the algorithm based on a small example. This simulation in particular shows how partition enumeration works. Suppose the CIT problem $\Pi = (P, I, S, c)$ where:

$$\begin{aligned} P &= \{a, b, c\} \\ I &= \{i_1, i_2\} \,\mathbf{where}\, i_1 = \{a, b\}, i_2 = \{b, c\} \\ S &= \{s_1, s_2\} \,\mathbf{where}\, s_1 = (a = b = c), s_2 = (\neg\, s_1) \\ c &= a \in [0..1] \wedge b \in [0..1] \wedge c \in [0..2] \end{aligned}$$

The constraint c defines the range of the parameters. Note that $x \in [l..u]$ is just a shortcut for $x \geq l \wedge x \leq u$. The constraint can be more general than simple value ranges. However, for the CIT algorithm to terminate, the number of possible solutions to the parameters under the constraint and seed disjunction must be bounded. These are the steps perform by the algorithm for that input:

1. Initialize $\phi := c \wedge \vee S$ (i.e. $\phi = c$ as $\vee S = \mathsf{true}$).
2. Solve partitions of cardinality 2. There is only one such partition. This leads to the call $\mathit{SolvePartition}(\{i_1, i_2\})$, which is processed as follows:
   (a) Call $\mathit{Solve}(\mathit{ExcludeCovered}(\{i_1, i_2\}) \wedge s_2)$. As there is no coverage at this point, this amounts to $\mathit{Solve}(s_2)$. The solution the solver produces is $(0, 0, 1)$, and henceforth ϕ will be updated to $\phi := \phi \wedge \neg\, (a = 0 \wedge b = 0 \wedge c = 1)$. Moreover, interaction solutions $\{a \mapsto 0, b \mapsto 0\}$ and $\{b \mapsto 0, c \mapsto 1\}$ are added to $\mathit{coverage}(i_1)$ and $\mathit{coverage}(i_2)$, resp.

(b) Remove the seed s_2 from the *seeds* set.
(c) Call $\mathit{Solve}(\mathit{ExcludeCovered}(\{i_1, i_2\}) \ \wedge \ s_1)$. This amounts to the call $\mathit{Solve}(\neg\ (a == 0 \wedge b == 0) \wedge \neg\ (b == 0 \wedge c == 1) \wedge s_1)$. The only solution the solver can produce is $(1, 1, 1)$. Hence, $\phi := \phi \wedge \neg\ (a = 1 \wedge b = 1 \wedge c = 1)$, and interaction solutions $\{a \mapsto 1, b \mapsto 1\}$ and $\{b \mapsto 1, c \mapsto 1\}$ will be added to *coverage*.
(d) Remove the seed s_1 from the *seeds* set which becomes empty at this point.
(e) From now on, additional solutions will be generated without involvement of seeds. The solutions are $(1, 0, 0)$ and $(0, 1, 2)$. At the end of this step, we have:

$$\mathit{coverage}(\{a, b\}) = \{(0, 0), (1, 1), (1, 0), (0, 1)\}$$
$$\mathit{coverage}(\{b, c\}) = \{(0, 1), (1, 1), (0, 0), (1, 2)\}$$

This closes interaction $i_1 = \{a, b\}$, which will be removed from the *open* set; yet, i_2 is still open.

3. The algorithm now goes to partition cardinality level 1. While there exists two partitions at this level, namely $\{\{i_1\}\}$ and $\{\{i_2\}\}$, since i_1 is not longer open, only one call $\mathit{SolvePartition}(\{i_2\})$ will be processed. As there are no seeds open at this point, the algorithm will try to produce as many solutions as it can using $\mathit{Solve}(\mathit{ExcludeCovered}(\{i_2\}))$. In the first iteration, the exclusion expands to $\mathit{Solve}(\neg\ (b = 0 \wedge c = 1) \wedge \neg\ (b = 1 \wedge c = 1) \wedge \neg\ (b = 0 \wedge c = 0) \wedge \neg\ (b = 1 \wedge c = 2))$. The algorithm produces subsequent solutions $(0, 0, 2)$ and $(0, 1, 0)$. This adds the missing coverage for $i_2 = \{b, c\}$, while necessarily repeating some of the already covered combinations for i_1.

Implementation The implementation of our CIT solution follows the conceptual approach as shown in Fig. 4. However, there are a few notable differences which result from efficiency considerations. Instead of constructing the exclusion constraints from scratch over and over again, they are incrementally built. Moreover, Z3's capability to maintain nested contexts and assert constraints in these contexts is exploited to avoid passing the exclusion constraint as a whole wherever this is possible. Instead, in the loop where models are generated, exclusions are added in each iteration to the current active Z3 context. This allows the solver to maintain internal caches and analysis results for the accumulated constraint, and is key to efficiency. In addition, calls to the solver are avoided when possible, e,g. for the closure check, which only needs to be performed on those interactions which have new solutions, and only at the end of each partition processing.

4 Benchmarks

In order to measure the efficiency of the CIT algorithm, we have compared it with other available tools. Such a comparison is not completely adequate, as our approach is more expressive by enabling general constraints and seeds, but useful anyway to assess the cost of the generalization. The results are shown in Table 1.

Many tools [3–5, 17] are available to generate t-wise CIT samples, where t is fixed between all parameter interactions, and is usually equal to 2 or 3. Some of these are also able to handle certain types of constraints expressed as "exclusions": specific value

GROUP 1 Fixed Strength with or without "Excludes" [6]

NO	Problem definition	mAETG_SAT	SA_SAT	PICT	TestCover	Our
1	CA(3, 5^4)	143	127	151	NA	147
2	CA(2, 6^3)	37	36	39	36	38
3	CA(2,3^3) Excludes: {(5,6), (4,6), (0,7), (2,3), (2,8), (1,5,8)}	10	10	10	10	10
4	CA(3, 6^4) Excludes: {(4,20),(15,19),(8,20),(7,14)}	241	251	250	NA	244
5	*t*=2 SPIN simulator without constraints	25/0.4s	16/25.6s	23/1s	NA	40/3s
6	*t*=2 SPIN simulator with constraints	24/1.5s	19/694.3s	26/1s	NA	43/4s

GROUP 2 Variable Strength: Disjoint Interaction Sets (average of ten runs) [4]

7	CA(2,$4^3 5^3 6^2$) 3-way(V_0, V_1, V_2, V_3, V_4)	NA	101	NA	NA	176/1.4s
8	CA(2,$4^3 5^3 6^2$) 3-way(V_0, V_1, V_2); 3-way(V_3, V_4, V_5)	NA	125	NA	NA	142/0.8s
9	CA(2,$4^3 5^3 6^2$) 3-way(V_0, V_1, V_2)	NA	180	NA	NA	187/0.8s
10	CA(2,$3^9 2^2$) 3-way(V_0, V_1, V_2); 3-way(V_3, V_4, V_5); 3-way(V_6, V_7, V_8)	NA	27	NA	NA	50/0.7s
11	CA(2,3^{15},{CA(3,3^4), CA(3,3^5),CA(3,3^6)}) 3-way(V_0, V_1, V_2, V_3); 3-way(V_4, V_5, V_6, V_7, V_8); 3-way($V_9, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}$)	NA	34.8	NA	NA	120/4s

Table 1. Benchmark Results (number of solutions/execution time)

combinations that must not occur in the sample. We use the benchmarks proposed by Cohen et al. [6] as the first group, since they are in this category and are based on real CIT problems for configurable software. We compare our results with the published results from 4 tools [6]: mAETG_SAT, a greedy algorithm; SA_SAT, a simulated annealing meta-heuristic algorithm; and two commercial tools, PICT and TestCover [8, 20].

The first column of Table 1 defines the problem for each benchmark. We use a notation similar to that of [4, 6] to make the comparison easier. CA(t,$v_i^{x_i}$) means there are x_i parameters, each with v_i values, and t is the testing (or interaction) strength. All parameters will be combined with the same interaction strength. Values for each parameter are written as unique continuous integers starting from 0, across all parameters. For instance, if the first parameter has 3 values, then these are assigned $0, 1, 2$, while the next parameter values start at 4. CA(2,$4^3 5^3 6^2$) stands for a pairwise CIT sample between 8 parameters: the first 3 parameters contain 4 values each, the second 3 parameters contain 5 values each, and the last 2 parameters contain 6 values each. In this example,

the total number of unique values is 39 labeled from 0 to 38. "Excludes" sets contain combinations of values that must be excluded.

The first group can be divided into 2 subgroups. The first subgroup (case 1 to 4) is composed of benchmarks containing a small number of parameters – no more than 5–, while the other is composed of real subjects (like the "spin simulator" – case 5 and 6) that contain between 20 and 60 parameters. The definitions can be found in [6].

Since the problems in the former subgroup are small, execution time is not provided in [6]. We compare these by measuring the number of solutions only: the smaller the solution size, the less effort testing will require. From this viewpoint, our approach works well by generating solutions within the middle of the range of other tools. For the larger sized problems, as in the latter subgroup, both size and time are used to measure the results. In this subgroup (5 and 6), we find that our approach takes less than 5 seconds to run, but generates more solutions than the other tools. Note that when disabling our preliminary pruning heuristic the size of the solution set becomes comparable to the best in the group; however, this can cause our algorithm to hang for some cases (not the given benchmarks).

In the second group, we provide additional subsets of parameters that are to be combined at higher strengths of testing. We use the notation (V_i) to represent the parameter indexed from left to right that is contained within the higher strength subset. For example, in case 9, in addition to pairwise interactions between each pair of variables, a 3-way interaction among variables V_0,V_1 and V_2 is also required. The data in this group is from Cohen et al. [4] (average of ten runs) where variable strength is restricted to disjoint sets.[3]

For the second group, only the SA algorithm is available for comparison. We use the benchmarks that were created to measure it [4]. In case 7 through 9, where there are a small number of parameters – less than 10– our approach sometimes generates a solution comparable in size to that of SA, in less than 1 second. However, some of our solutions (case 1) are considerably larger. As the number of input parameters increases to 15 (case 11), although our approach still can find solutions within 5 seconds, the size of the generated solutions increases to nearly 4 times that of the solutions produced by SA.

In summary, our approach is able to generate small sized solutions competitive with other tools for small problems, containing less than 10 parameters, as in cases 1, 2, 3 and 4. Our absolute run-times, although sometimes greater than those of other tools, never exceed a minute across all runs and most finish within 10 seconds. In the context of the application to Spec Explorer, this additional time is marginal compared to other processes, such as symbolic exploration in general.

As the number of parameters increases, our approach generates solutions larger than those of some other tools. In these cases, the failure point is the pruning heuristic failed, since in previous benchmarks where pruning was not yet applied, timing and solution size were very competitive. However, pruning is required, as we discovered when testing the algorithm against other samples, and finding a better heuristic than the current one needs further investigation.

[3] PICT may be able to handle this, but its public version is not.

5 Related Work

There has been a large body of work on generating test suites and configuration samples using CIT [1, 3, 15, 22, 18]. Applications of CIT include functional input testing, [3], configuration sampling [22], regression testing [18] and more recently event sequence testing [23]. In most of this work the focus has been to construct a CIT sample using a model that has a single interaction strength across all parameters of the problem. Two primary algorithm types have been used for CIT sample generation – greedy methods [3, 8, 17] and meta-heuristic search [4, 5]. Other techniques use direct or recursive mathematical constructions (see [15] for a survey). The focus of much of this work has been on minimizing the CIT sample size.

There has been limited work on constructing and applying variable strength CIT [3, 4, 8, 22]. The work of both D. Cohen et al. [3] and Czerwonka [8] models hierarchies of parameters represented as varying strengths into their greedy construction algorithms. In both [4, 22] a restricted type of variable strength CIT is generated and empirically evaluated. The solution is limited to a subset of cases where interactions are partitioned into disjoint sets for stronger testing and a base CIT sample is maintained over the whole set of parameters. Our work goes further, by making interaction sets the primary focus of our construction – they drive the generation and there is no restriction on the relationships between interaction sets.

Inter-parameter dependencies (or constraints) have been the subject of several recent papers. They were first studied by D. Cohen et al. in [3]. The primary method for resolving constraints in that work is to remodel interaction definitions. Work by Cohen et al. [6, 7] introduces the use of Boolean SAT solvers to ensure that samples generated by heuristic search and greedy algorithms cover all required interactions while satisfying constraints. In this work all constraints are translated into "exclude" constraints and generalized seeds are not provided. Recent work by Calvagna et al. [2] provide a way to consider temporal constraints for execution sequences. Hnich et al [16] use a SAT solver to generate CIT samples, but do not incorporate any type of inter-parameter constraints. Their work is limited to generating pair-wise CIT samples with a small number of values.

The work that most closely resembles ours is that of Calvagna et al. [1, 2] who have developed an algorithm to generate CIT samples using the SAL model checker, which relies on an SMT solver. In this work the authors use a formal specification to derive the interaction model and enumerate the required test predicates, which are then encoded as negations to iteratively generate counter examples for each t-set. Heuristics determine predicate search order and a reduction step is included to remove duplicate test cases. As in our work, the approach uses only constraint resolution and they allow generalized seeds. A primary difference, however, is that they generate only a single fixed strength CIT sample that does not include isolates. In addition, our algorithm is direct – we do not have a reduction step, but rather avoid duplicates through the dynamic addition/deletion of constraints. Finally, they have integrated CIT into their model based testing tool, but our work explicitly uses the state machine path conditions to define varying sets of interactions and is therefore more tightly coupled with the behavioral model.

6 Conclusion

We have presented an integration of combinatorial interaction testing into path exploration as applied for the behavioral model-based testing tool Spec Explorer. The approach we have shown can be also well integrated with tools which do path exploration on program code, like PEX or CUTE [19, 21].

The requirements derived from this application have led to a novel CIT algorithm heavily dependent on constraints as inputs. In particular, generalized seeds, represented as constraints, need to be considered by the algorithm. This naturally led to an approach based entirely on constraint resolution, powered by the constraint engine Z3 [9]. The algorithm can process arbitrary constraints on the solution set, seeds as constraints, and interactions of variable strength. It also supports generalized isolation of solution tuples by a predicate, which we have not shown in this paper due to space restrictions. (Isolation helps exclude certain solutions from counting for interaction coverage, as they may hit error situations in the system under test or other special behavior that overrides the effect from other combinations in the path.)

Our design is simple and comprehensible, and the performance of our implementation is comparatively fast. The size of the generated solution set is not always as good (small) as it could be, which we relate to a poor heuristic in plateau pruning requiring further investigation.

Our approach has been motivated by a real-world application of model-based testing in a large scale industry project [12]. Since put in place, it is being increasingly used on an every-day basis by test suite developers for documentation interoperability testing of network protocols.

Acknowledgments Keith Stobie provided requirements and has been one of the driving forces to make CIT part of Spec Explorer. Yiming Cao and the Spec Explorer development and testing team gave many feedback and implemented some of this work. Special thanks go to Nikolaj Bjørner and Leonardo de Moura for their work on Z3 and their prompt and continuous support when developing this application of their tool. This work was supported in part by the National Science Foundation through award CCF-0747009.

References

1. A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and Proofs, Lecture Notes in Computer Science, 4966*, pages 66–83, 2008.
2. A. Calvagna and A. Gargantini. Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints. In *Workshop on Automated Formal Methods (AFM)*, pages 1–10, 2008.
3. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.
4. M. B. Cohen, C. J. Colbourn, J. Collofello, P. B. Gibbons, and W. B. Mugridge. Variable strength interaction testing of components. In *Proceedings of the International Computer Software and Applications Conference*, pages 413–418, 2003.

5. M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 38–48, May 2003.
6. M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, July 2007.
7. M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.
8. J. Czerwonka. Pairwise testing in real world. In *Pacific Northwest Software Quality Conference*, pages 419–430, October 2006.
9. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 08*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.
10. W. Grieskamp. Multi-paradigmatic Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006.
11. W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In J. Whittle, L. Geiger, and M. Meisinger, editors, *SCESM*, pages 59–66. ACM, 2006.
12. W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. L. Wurden. Model-Based Quality Assurance of Windows Protocol Documentation. In *ICST*, pages 502–506. IEEE Computer Society, 2008.
13. W. Grieskamp, N. Kicillof, and N. Tillmann. Action Machines: a Framework for Encoding and Composing Partial Behaviors. *International Journal of Software Engineering and Knowledge Engineering*, 16(5):705–726, 2006.
14. W. Grieskamp, N. Tillmann, and W. Schulte. XRT – Exploring Runtime for .NET: Architecture and Applications. *Electr. Notes Theor. Comput. Sci.*, 144(3):3–26, 2006.
15. A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math*, 284:149 – 156, 2004.
16. B. Hnich, S. Prestwich, E. Selensky, and B. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.
17. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. *Engineering of Computer-Based Systems, IEEE International Conference on the*, pages 549–556, 2007.
18. X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–85, July 2008.
19. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.
20. G. Sherwood. Testcover.com. http://testcover.com/pub/constex.php, 2006.
21. N. Tillmann and J. de Halleux. Pex – White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.
22. C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, 2006.
23. X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*, pages 405–408, 2007.