An Approach for Effective Design Space Exploration

Eunsuk Kang[1], Ethan Jackson[2], and Wolfram Schulte[2]

[1] Massachusetts Institute of Technology, Cambridge, MA, USA
eskang@mit.edu
[2] Microsoft Research, Redmond, WA, USA
ejackson,schulte@microsoft.com

Abstract. *Design space exploration* (DSE) refers to the activity of exploring design alternatives prior to implementation. The power to operate on the space of potential design candidates renders DSE useful for many engineering tasks, including rapid prototyping, optimization, and system integration. The main challenge in DSE arises from the sheer size of the design space that must be explored. Typically, a large system has millions, if not billions, of possibilities, and so enumerating every point in the design space is prohibitive. In this paper, we present a method for systematically exploring the design space in a cost-effective manner. The key idea is that many of the design candidates may be considered equivalent as far as the user is concerned, and so only a small subset of the space needs to be explored. Our approach takes the user-defined notion of equivalence, and generates *symmetry breaking predicates* to ensure that the underlying exploration engine does not sample multiple equivalent design candidates. We describe how the method is integrated into our DSE framework, *FORMULA*, which uses an SMT solver to solve a set of global design constraints and search for valid design instances.

1 Introduction

Design space exploration (DSE) refers to the activity of discovering and evaluating design alternatives during system development. It has many uses including:

- **Rapid prototyping**: DSE is used to generate a set of prototypes prior to implementation. Simulating and profiling of these prototypes can increase understanding of the impact of design decisions while taking complex system dynamics into account.
- **Optimization**: When metrics are available for comparing one design to another, DSE can be used to perform optimization, eliminating inferior designs and collecting a set of final candidates that are further studied.
- **System integration**: System integration requires the assembly and configuration of multiple components into a working whole. DSE can be used to find legal assemblies and configurations that satisfy a set of global design constraints.

DSE must be performed carefully because of the sheer number of design alternatives to be explored. A large complex system may admit millions, if not billions of design alternatives; in some cases, the design space may be infinite. A manual, ad-hoc approach to DSE is tedious, error-prone, and does not scale. An effective DSE framework must consist of the following ingredients:

- **Representation**: A suitable representation of the design space is essential. The representation should be formal, so that it can be subject to automated analysis and exploration techniques. A complex system may have a large number of design constraints that must be satisfied by every valid design solution. These constrains may involve arithmetic operations, Booleans expressions, and data type constraints over infinite domains. The representation should be expressive enough to capture these types of complex constraints.
- **Analysis**: A DSE framework must be equipped with machine-assisted techniques for discovering potential candidates, and checking them against the design constraints to ensure that they are actually valid design solutions. The framework must also be able to tackle the challenge of solving a large number of complex constraints at reasonable computational costs.
- **Exploration method**: Even after an optimization procedure rules out all inferior designs, the user may end up with the task of exploring a large number of design candidates. Enumerating them one-by-one in an ad-hoc fashion is not desirable. As far as the user is concerned, some of the solutions may be considered equivalent, and the user may be interested in examining only the ones that are distinctive from each other. The framework must provide a method for navigating to interesting solutions.

In previous papers [15, 16], we proposed our DSE framework, called *FORMULA*, and discussed its representation of the design space and the underlying analysis engine, which is based on the *Z3* SMT solver [10]. In this paper, we describe the method in FORMULA for sampling a set of interesting design solutions. We say a solution is *interesting* if it is considered distinct from any other solution that has already been explored, under the user-defined notion of equivalence. Formally, two solutions are considered equivalent if their mathematical representations are *isomorphic* to each other. We show how we allow the user to define an equivalence relation that groups all isomorphic solutions into a single equivalence class. Our approach applies *symmetry breaking predicates* [8] to ensure that FORMULA returns exactly one solution from each equivalence class, thereby avoiding uninteresting designs from being presented to the user.

This paper is structured in the following way. We will begin by presenting a motivating example (Section 2). We will present background information on FORMULA—its representation of the design space and the SMT-based analysis engine for solving design constraints (Section 3). We will outline a method for exploring the design space in a way that guarantees only distinct design candidates to be found (Section 4). Then, we will discuss an experiment demonstrating the effectiveness of our approach (Section 5). Finally, we will conclude with discussions of related work (Section 6) and future directions (Section 7).

2 Motivating Example

We begin with an example that is simple but challenging for existing DSE methods. This example is borrowed from *platform mapping* problems found in automotive embedded systems [19, 26]. Given a set of software *tasks* and *devices*, the goal is to map each task onto a device in such a way that a certain set of design constraints are satisfied.

We formally describe the problem as follows. Let T be a set of named tasks. A *conflict graph* $C = (T, E_C)$ is a labeled undirected graph over tasks. An edge $(t_1, t_2) \in E_C$ indicates that tasks t_1 and t_2 are in conflict and should not be executed on the same device. Let D be a set of named devices. Then, a *distributed network* $N = (D, E_N, cap)$ is a triple where (D, E_N) is a labeled *directed* graph. For each edge $(d_1, d_2) \in E_N$, there is a directed communication channel from d_1 to d_2. The notation $in(d)$ indicates the set of incoming communication channels of device d, and $out(d)$ its outgoing channels. Every channel has a strictly positive *capacity* as assigned by the function $cap : E_N \rightarrow \mathbf{Z}_+$. Finally, tasks are bound to devices by the function $bind : T \rightarrow D$. The structures C, N, and *bind* provide a representation for instances of the design space—i.e. possible configurations of tasks, devices, and mappings between them.

Every valid design of the system must satisfy the following design constraints:

1. A pair of conflicting tasks cannot be mapped onto the same device: $\forall t_1, t_2 \in E_C \cdot \ bind(t_1) \neq bind(t_2)$.
2. A single device can provide at maximum two ingoing and/or outgoing channels: $\forall d \in D \cdot \ |in(d)| \leq 2 \wedge |out(d)| \leq 2$.
3. Each device with both input and output channels must have balanced capacities:
$\forall d \in D \cdot in(d) \neq \emptyset \wedge out(d) \neq \emptyset \Rightarrow \sum_{i \in in(d)} cap(i) = \sum_{o \in out(d)} cap(o)$.

Constraint (1) is equivalent to a graph coloring problem, and requires reasoning about the global topology of the system. Constraint (2) is a forbidden subgraph problem. Constraint (3) requires arithmetic reasoning and is guarded by a Boolean constraint.

Let us assume the engineer has chosen a set of tasks and identified conflicts appropriately. Then, the possible design alternatives arise from variations in network topologies, capacities, and task bindings. Figure 1 shows one possible configuration of the system. Note that this instance satisfies all of the design constraints. If, for example, the capacity of the channel from device B were to be altered from 1 to 2, then the modified instance would fail to satisfy constraint (3), and no longer qualify as a legal design candidate.

From the perspective of the engineer, the "best" design might be one that utilizes the communication channels most efficiently. However, this utilization depends on the code executed by the tasks, the scheduling strategy of the underlying operating system, the communication protocols implementing channels, and a number of other factors. An optimization problem cannot be formulated easily at this high-level, and so rapid prototyping combined with simulation is the approach that is often taken to evaluate design alternatives [19].

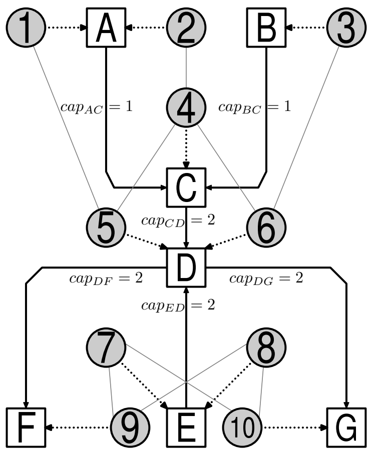

Fig. 1. A design instance of the platform mapping problem, representing one possible configuration of the system. Circles represent tasks, and squares represent devices. Every edge between a pair of devices is labeled with the channel capacity. This instance contains ten tasks, with conflicts indicated by gray lines. A dotted line from a task to a device indicates that the task has been mapped to the device.

The requirement on the DSE engine is to enumerate design alternatives that satisfy global constraints. However, this example is challenging for several reasons. First, instantiating a single solution requires solving non-trivial constraints, such as arithmetic and relational constraints. Second, the number of design alternatives is, in principle, infinite, because no bounds where placed on the channel capacities, the number of devices, or the domain of their labellings; certainly, the entire space cannot be explicitly enumerated. Third, labels on the devices are also a design parameter, and thus, each solution in the design space will have a large (possibly infinite) number of counterparts that differ only by labeling of devices. These countparts may be of no interest to the user, and so the exploration method must be able to eliminate these equivalent solutions. As we shall show, this last criterion is particularly challenging to achieve.

3 Background on FORMULA

FORMULA is our modeling framework for formally specifying domain-specific languages [17]. From the DSL perspective, the representation and constraints of a design space form a domain-specific abstraction; DSLs are ideal for capturing such abstractions. Additionally, the DSL metaphor allows complex design spaces to be built from smaller ones using DSL composition operators. In this section we introduce just enough of FORMULA to encode our motivating example; please see [16] for a more detailed discussion.

```
1.   domain Functionality
2.   {
3.     Task        ::=  (id: Basic).
4.     [Closed]
5.     Conflict    ::=  (t1: Task, t2: Task).
6.   }
7.   model ThreeTasks of Functionality
8.   {
9.     Task(1)
10.    Task(2)
11.    Task(3)
12.    Conflict(Task(1), Task(2))
13.    Conflict(Task(2), Task(3))
14.  }
```

Fig. 2. Examples of a domain and a model specified in FORMULA. The Functionality domain encodes the abstraction related to tasks and conflicts. The model contains three tasks with two conflicts among the tasks.

3.1 Representation

A domain block encapsulates the data types and constraints of a DSL, as shown in Figure 2. A data type is either the name of a sort (a set of constants, e.g. String), a record constructor, or an arbitrary union of other data types. Line 3 declares a constructor called Task, which takes an id argument of type Basic (which corresponds to the set of all constants). Line 5 declares a constructor for denoting conflicts between tasks, which requires two arguments of type Task. FORMULA data types are algebraic: Two data instances are the same if and only if they were built from the same sequence of constructors and constants. This formalism captures inductive data types with type constraints. A model is a set of record instances built using the constructors of a domain that satisfy domain constraints (lines 7-14). The declaration model ThreeTasks of Functionality is a *claim* that the model satisfies constraints; the claim is verified by FORMULA.

Some domain constraints are quite common; e.g. conflict edges form a relation over tasks: $E_C \subseteq T \times T$. FORMULA provides built-in support for common constraints via annotations on data type declarations. The [Closed] annotation applied to the Conflict constructor is an example. Let $[\![C]\!]$ be the set of all well-typed records that can be constructed by C. If M is a set of records, then $M(C) = M \cap [\![C]\!]$ is the set of C-records in M. For example $M(Task)$ and $M(Conflict)$ is the set of all tasks/conflicts respectively. The *closed* annotation requires every model M to satisfy $\{(t1, t2) \mid Conflict(t1, t2) \in M(Conflict)\} \subseteq M(Task) \times M(Task)$.

In general, a rich constraint language is needed to specify domain constraints. Many modeling tools use the *object constraint language* (OCL) for this purpose. However, the intricacies of OCL complicate automated analysis of arbitrary OCL constraints [21]. For this reason, we choose *constraint logic programming* (CLP)

```
1.    domain Distribution
2.    {
3.      Device    ::= (id: Basic).
4.      [PartialFunction(src, dst -> cap)]
5.      Channel  ::= (src: Device, dst: Device, cap: PosInteger).
6.
7.      bigFanIn    :=d is Device, count(Channel(_, d, _)) > 2.
8.      bigFanOut  :=d is Device, count(Channel(d, _, _)) > 2.
9.      clog          :=d is Device,
10.                     sum(Channel(_, d, _), 2) != sum(Channel(d, _, _), 2).
11.     conforms    :=!(bigFanIn | bigFanOut | clog).
12.   }
```

Fig. 3. A domain representing the distribution of devices through channels.

for the core constraint language of FORMULA. CLP is well studied, has an unambiguous execution semantics, and can be converted into first-order logic. In fact, FORMULA converts all built-in constraint annotations into logic programs.

Figure 3 shows an abstraction for the distributed network of devices through channels, which requires more complex constraints to specify. A Channel is a partial function from a pair of Devices to a positive integer. Line 7 defines a *query* for checking whether an input model M has a Device with too many incoming Channels. For each binding of the variable d to a Device, the count operator counts the number of distinct Channels terminating on d (the underscores are "don't care" variables). If there is any binding of d with more than two incoming Channels, then the Boolean variable bigFanIn evaluates to true. The bigFanOut query (line 8) performs the same check for outgoing Channels. The clog query checks if the communication network is unbalanced by summing the capacities on incoming/outgoing Channels. The second argument of the sum operator is the zero-indexed field within the record that is summed.

Every FORMULA domain has a query called conforms. By definition, an input model satisfies domain constraints only if conforms evaluates to true. The design space associated with a domain is the set of models satisfying its conforms query. In the example from Figure 2, the conforms query has not been explicitly defined by the user; in this case, FORMULA will implicitly define the query as a conjunction of compiler generated constraints (e.g. [Closed]). In Figure 3, the Distribution domain explicitly requires that none of bigFanIn, bigFanOut, and clog should evaluate to true. The entire conforms query for Distribution also contains compiler generated constraints due to the annotation [PartialFunction].

The DSL approach supports modular and compositional specification of abstractions. The Architecture domain (Figure 4) is an extension of the product of the Functionality and Distribution domains. These composition operations allow the Architecture domain to use the data structures of Functionality and Distribution while provably ensuring that all constraints are enforced the same way [16]. Architecture also adds a new data structure Binding and requires that Bindings must respect task conflicts (lines 18-19). Again, the complete conforms of Archi-

```
domain Architecture  extends Functionality, Distribution
{
  [Function]
  Binding  ::= (t: Task, d: Device).

  conflict    := Binding(t1, d), Binding(t2, d), Conflict(t1, t2).
  conforms  := !conflict.
}
```

Fig. 4. A domain as a composition of Functionality and Distribution.

tecture contains constraints imported from the other domains. In summary, the models conforming to Architecture are exactly those legal systems described in Section 2. The DSL approach allows the user to encode the interesting degrees of design freedom via formal and composable abstractions. Specifically, FORMULA utilizes algebraic data types and CLP to accomplish this.

3.2 Solving for Instances

In order to find non-trivial solutions to design spaces, FORMULA specifications are translated into the SMT solver Z3. Let $D.q$ be a query q defined in domain D, then the translation procedure must produce a first-order formula $\varphi[X]$ with the following property: Finite models (sets of records) satisfying $D.q$ are in correspondence with satisfying instances of $\varphi[X]$, where X denotes the vector of variables appearing in φ. A satisfying instance is a mapping of variables to values $\{x_1 \mapsto v1_, \ldots, x_n \mapsto v_n\}$; a reverse translation converts satisfying instances into FORMULA models.

SMT solvers represent a significant step in automated theorem proving by soundly combining decision procedures for different theories while using efficient SAT-based backtracking techniques to drive the search process. For example, the clog query (lines 9-10, Figure 3) imparts the following fragment into φ:

$$\begin{array}{c} test_{Device}(d) \wedge test_{Channel}(in1) \wedge test_{Channel}(in2) \wedge \\ sel_{Channel,1}(in1) = d \wedge sel_{Channel,1}(in2) = d \wedge in1 \neq in2 \wedge \\ x = 2Int(sel_{Channel,2}(in1)) + 2Int(sel_{Channel,2}(in2)) \ldots \end{array} \tag{1}$$

This fragment sums the incoming channel capacities for a device d with two distinct incoming channels. SAT techniques provide a strategy for satisfying sub-formulas, and specific decision procedures actually solve the sub-formulas. In this example, two decision procedures are required: (1) term algebras (TA) for inductive data types and (2) linear arithmetic for summing channels. The first line of the formula uses TA to test that the variables d, $in1$, and $in2$ have the appropriate record structure. The second line extracts the second components of the channels $in1$ and $in2$ using TA *selectors*; the equalities here invoke unification and the *occurs check*. The third line extracts the channel capacities, coerces them to integers using the function $2Int$, and calculates their sum via the linear arithmetic decision procedure.

This example illustrates the power of SMT, but also shows that the translation process from a high level specification to SMT is non-trivial, since most SMT solvers support only the existential fragment of first-order logic. In our approach, universal quantifiers are eliminated by *symbolically executing* a specification over a set of symbolic inputs and emitting all interesting branches of the logic program as a quantifier free formula. The symbolic execution loop is implemented outside of the theorem prover, and it takes as input a finite set of records with variables where constants would otherwise be. For example, symbolic execution on the following set:

$$S = \left\{ \begin{array}{c} Task(x_1), Task(x_2), Task(x_3), \\ Device(x_4), Device(x_5), \\ Conflict(x_6, x_7), \\ Channel(x_8, x_9, x_{10}), \\ Binding(x_{11}, x_{12}), Binding(x_{13}, x_{14}), Binding(x_{15}, x_{16}) \end{array} \right\} \tag{2}$$

produces a formula φ capturing all the possible ways that zero to three Tasks, zero to two Devices, etc... can satisfy design constraints. When the DSE procedure does not know cardinality bounds for all record types, then it repeatedly attempts larger and larger symbolic sets as input to the symbolic execution engine. Even though each symbolic input set has a finite number of records, the resulting SMT formula may still have an infinite number of solutions, because variables occurring in φ range over infinite domains. We refer to the original finite set used to produce φ via symbolic execution as the *generator set of* φ.

4 Design Space Exploration Method

After symbolic execution, elements of the design space can be enumerated by repeatedly querying the SMT solver. This procedure is not sufficient for rapidly exploring diverse solutions, because the solver does not know which solutions are considered similar. Also, solving strategies are optimized to find any *next* solution, and not necessarily solutions that are highly distinct. In this section, we describe a technique for grouping related solutions based on isomorphisms over algebraic data types.

4.1 Projection-Based Equivalence Partitioning

Let Σ be the set of constants that might appear in the field of some record. Let $\mathcal{C}$ be the set of all constructors of a domain D. A *term homomorphism* ϕ is a function over constants lifted onto records. If $c(r_1, \ldots, r_n)$ is a record built by applying constructor c to records $r_1, \ldots, r_n$, then $\phi(c(r_1, \ldots, r_n))$ returns a new record that is equal to $c(\phi(r_1), \ldots, \phi(r_n))$. Term homomorphisms preserve the structure of records, but change the constants appearing in their fields. If M is a model (i.e. a set of records), then $\phi(M)$ is a model formed by applying ϕ to each record in M. Homomorphisms induce a preorder on models: $M' \preceq M$ if

$\exists \phi \cdot \phi(M) = M'$. Two models M, M' are *isomorphic* if $M' \preceq M$ and $M \preceq M'$; this can be written as $M \sim M'$.

Isomorphic models are equivalent up to relabeling of the constant values appearing in their records. Our approach groups all isomorphic solutions into a single equivalence class, and finds only one representative per equivalence class. We take this one step further, allowing isomorphisms to be considered on some subsets of data types, thereby further decreasing the number of distinct equivalence classes. We call $\Pi \subseteq \mathcal{C}$ a *projection* on the records of domain D. If M is a model, then $\Pi(M)$ discards records not in Π:

$$\Pi(M) = \{r \mid r = c(r_1, \ldots, r_n) \wedge r \in M \wedge c \in \Pi\} \tag{3}$$

Given a projection Π, then $M' \preceq_\Pi M$ if $\exists \phi \cdot \ \phi(\Pi(M)) = \Pi(M')$. Again, two models are in the same equivalence class if and only if $M \preceq_\Pi M'$ and $M' \preceq_\Pi M$.

FORMULA provides the user with an interface for specifying a projection. Returning to the motivating example, suppose the user wants to see only those solutions with distinct channel topologies. In the case, the user may specify $\Pi = \{Channel\}$, and every solution returned by FORMULA will have a distinct communication topology.

This approach must be integrated with the solver, so it knows to return non-isomorphic solutions. Communicating isomorphism-based equivalence classes to solvers can be accomplished using *symmetry breaking predicates* [8].

4.2 Exploration Algorithms

Encoding equivalence classes into the SMT solver using symmetry breaking predicates can ensure that every new solution is non-isomorphic to previous ones. However, this alone does not diversify the exploration throughout the design space; in other words, FORMULA may consecutively return non-isomorphic but structurally similar solutions within a small portion of the space. Ideally, we want the solver to "jump around" various parts of the design space, sampling a wide variety of non-isomorphic solutions. In this section, we describe an algorithm that explores the design space for a particular generator set G, and show how we employ randomization to incrementally construct a diverse set of non-isomorphic solutions.

Naïve Exploration Algorithm We begin by describing a simple candidate algorithm *Explore* (Figure 5) to build intuition. The algorithm accepts as inputs the generator set G, the formula φ generated from G, which encodes the design constraints, and a user-specified projection Π. The algorithm randomly samples an equivalences class in the design space, and then checks if that equivalence class contains a model satisfying φ. A sample s is a symbolic set of records under the projection Π. For example, given the generator set from Equation 2 in Section 3.2, and $\Pi = \{Binding\}$, one possible sample is:

$$\{Binding(x_1, x_2), Binding(x_3, x_2), Binding(x_4, x_5)\}. \tag{4}$$

```
Explore($G$, $\varphi$, $\Pi$)
 1: $solutions := \{\}$
 2: $sampled := \{\}$
 3: while $True$ do
 4:     $s := SampleClass(G, \Pi)$
 5:     for all $p$ in $sampled$ do
 6:         if $TestIsomorphism(s, p)$ then
 7:             goto Line 3
 8:         end if
 9:     end for
10:     $sampled := sampled \cup \{s\}$
11:     $soln := FindModel(s \wedge distinct(s) \wedge \varphi)$
12:     if $soln \neq NULL$ then
13:         $solutions := solutions \cup \{soln\}$
14:     end if
15:     if $CheckExhaustive(sampled)$ then
16:         return $solutions$
17:     end if
18: end while
```

Fig. 5. Naïve exploration algorithm.

Note that the first two Binding terms contain the same variable for the second argument to the constructor (x_2). This sample represents the set of all design instances in which two of the tasks are mapped to the same device. Figure 6 provides a graphical illustration of the sample.

The basic algorithm consists of a single while loop. In each iteration, an equivalence class s in the design space is sampled based on the projection (line 4), and is discarded if it is isomorphic to any of the samples that *Explore* has visited (lines 5-9). This check avoids isomorphic solutions from being collected.

If the sample passes this check, then the SMT solver attempts to construct a solution to the new sample that satisfies all of the design constraints (line 11). In this procedure, all the variables appearing in the sample are constrained to be distinct (*distinct*(s)), ensuring that the solver does not return a homomorphic image of s. If the solver succeeds in finding a satisfying instance, it stores the instance into the set *solutions*. If not, it goes back to the beginning of the loop and attempts another sample.

The exploration loop terminates when *Explore* has visited all equivalence classes in the design space (line 15)[3]. The termination condition is based on a property of the random sampling procedure (*SampleClass*): The probability that all classes are visited can be made arbitrarily close to 1 with a finite number of iterations.

How well does this algorithm work? Our goal is to collect a set of non-isomorphic solutions at a reasonable amount of computational cost. For this

[3] The exhaustive list of equivalence classes can be computed using a variant of Polya's enumeration theorem [28].

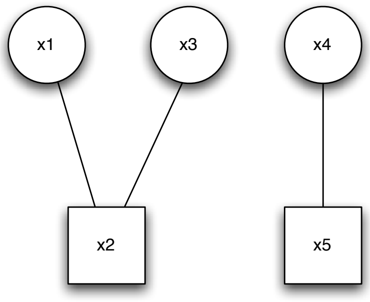

Fig. 6. A graphical representation of a sample for the platform mapping problem. Circles present tasks, and boxes represent devices. This particular sample represents the equivalence class of design solutions where two of the tasks are mapped to the same device.

purpose, we define the cost as the number of times that the algorithm invokes the SMT solver. Then, the success rate of the algorithm is the ratio of the number of non-isomorphic solutions to the number of calls to the SMT solver. A solving run that fails to find a satisfying instance is wasteful. But whether or not the SMT solver succeeds depends on the equivalence class that is picked by the sampling procedure. If the design space is loosely constrained, and a large number of equivalence classes contain models that satisfy constraints, then this simple algorithm should perform well. However, this assumption is often not true; the design space may be highly constrained, and random sampling may frequently pick samples that do not contain a satisfying instance. In practice, the algorithm performs poorly for a complex system that contains a large number of design constraints. This leads to a need for an algorithm that is able to avoid those parts of the design space that contain only invalid design instances.

Improved Algorithm In Figure 7, we present an alternative algorithm *ExploreII*, which avoids the observed problem with the naïve algorithm. Two key ideas distinguish the new algorithm. First, we allow the solver to check a possibly exponential number of equivalence classes per invocation. This is accomplished by removing distinctness constraints on variables in a random sample; the solver is now free to equate variables in the sample when searching for a satisfying instance. Secondly, *ExploreII* incrementally learns the regions of the design space that contain only invalid designs and then avoids examining these designs.

The outline of the new algorithm is as follows. It keeps track of two sets of samples, *valid* and *blocked*, whose purposes will be explained in the following paragraphs. Like the previous algorithm, *ExploreII* consists of a single while loop, which begins by sampling a symbolic set s in the design space (line 5). However, unlike the previous algorithm, the variables in this sample are no longer constrained to be distinct; some of them may be equated if the solver decides to assign the same constant to them. As a result of the relaxation, s is no longer

```
ExploreII($G$, $\varphi$, $\Pi$)
1: $solutions := \{\}$
2: $valid := \{\}$
3: $blocked := \{\}$
4: while $True$ do
5:     $s := SampleClass(G, \Pi)$
6:     for all $p$ in $blocked$ do
7:         if $TestHomomorphism(p, s)$ then
8:             goto Line 4
9:         end if
10:    end for
11:    $C := \{\}$
12:    for all $q$ in $valid$ do
13:        $C := C \cup ComputeHomorphism(s, q)$
14:    end for
15:    $soln := FindModel(s \wedge \neg C \wedge \varphi)$
16:    if $soln \neq NULL$ then
17:        $valid := valid \cup \{Simplify(s, soln)\}$
18:        $solutions := solutions \cup \{soln\}$
19:    else
20:        if $CheckMostGeneral(s)$ then
21:            return $solutions$
22:        end if
23:        $blocked := blocked \cup \{s\}$
24:    end if
25: end while
```

Fig. 7. Improved exploration algorithm

constrained to represent a single equivalence class, but can also represent *homomorphic* images of s spanning many equivalence classes. Consider Figure 8. The sample on the left hand side represents the equivalence class of design instances where two of the three tasks are mapped to the same device. By equating x1 and x3, we obtain a homomorphic image of the original sample; this image represents the equivalence class where each instance maps each task to exactly one device.

If a sample does not contain any instance that satisfies the given design constraints, then every homomorphic image of the sample will also be unsatisfiable. Thus, a new sample that is a homomorphic image of any element in *blocked* is deemed invalid or redundant, and immediately discarded to save the solver from doing wasteful work (lines 6-10).

Since a sample may admit solutions in multiple equivalence classes, *ExploreII* must prevent the solver from returning a solution that it has already found. The procedure *ComputeHomomorphism(s, q)* computes a homomorphism (if it exists) from s to q in the form of (dis)equalities over the variables in s and q. At the end of the loop on lines 12-14, C contains the set of all homomorphisms from s to the elements in *valid*. The negation of the disjunction of the constraints in C (represented by $\neg C$), prevents the solver from equating variables in s in a way

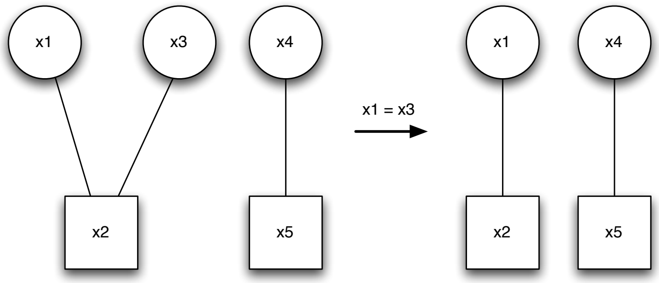

Fig. 8. Equality between x1 and x3 maps the sample on the left to its homomorphic image on the right. The resulting sample represents the set of instances in which each of the two tasks is mapped to exactly one device.

that would map s into one of the equivalence classes in *valid*. In other words, $\neg C$ is the symmetry breaking predicate that guarantees that the solver does not search the part of the design space that it has already visited. If a solution exists, *Simplify(s, soln)* derives a set of equalities between variables in s from *soln*, and uses them to reduce s into a canonical representation of the equivalence class that contains *soln* (line 17).

When no solution exists, *ExploreII* attempts to learn the characteristics of the failed sample. The unsatisfiability of $s \wedge \neg C$ implies that any homomorphic images of s besides those in C cannot satisfy the design constraints. Hence, *ExploreII* can safely reject any subsequent sample that is a homomorphic image of s, because every such image will either be unsatisfiable, or isomorphic to an element in *valid*. This knowledge can cause an exponentially large region of the design space to be avoided, but still allows random sampling over the good regions of the space.

The termination condition of *ExploreII* follows readily from the incremental aspect of the learning approach. We consider a sample to be *most general* when it is equal to the generating set G. The most general sample can be homomorphically mapped into any of the equivalence classes in the design space. If this sample becomes unsatisfiable, then this implies no more solutions are left to be discovered in the design space. Hence, the algorithm can be terminated when the most general sample is added to the blocked list (lines 20-21).

Since each sample represents a larger number of equivalence classes than it did in the naïve algorithm, the solver has more opportunities to find a satisfying design instance. Combined with the learning of failed samples, *ExploreII* attempts to overcome the difficulty of finding a satisfying instance in a highly constrained design space.

5 Evaluation

5.1 Experimental Setup

In this section, we evaluate the ability of the two algorithms to rapidly return solutions that are spread across the design space. We do not focus on the runtime performance of the SMT solver, since this has been well-studied [10]. In order to visualize the entire design space, we use a small generator set to produce a particular φ. In addition, the channel capacities have been moved outside of the Channel constructor in order to further reduce the number of equivalence classes. Let us assume that the generator set has the following set of distinct terms:

$$\begin{gathered} |\mathsf{Task}| = 3 \quad |\mathsf{Device}| = 3 \quad |\mathsf{Conflict}| = 1 \\ |\mathsf{Channel}| = 2 \quad |\mathsf{Binding}| = 3 \end{gathered} \tag{5}$$

The projection used is $\Pi = \{Binding, Channel\}$, and thus there are a total of 10 variables (one variable per each constructor argument in each distinct term) that determine membership in a particular equivalence class. The theoretical maximum number of equivalence classes is the number of partitions of ten variables, i.e. 115,975. However, due to symmetries in the sets of records, the actual number of equivalence classes for this example is 11,233.

We ran the two exploration algorithms, *Explore* and *ExploreII*, and observed the outcome of each invocation to the SMT solver. For every experimental run, we bounded the maximum number of the invocations to 100. Figure 9 shows plots for the experimental runs. Each one of the six plots is a graphical representation of the design space. Parts that are colored in red represent portions of the design space that were explored by FORMULA but did not admit a satisfying instance; ones in green are samples from which the solver was able to find a satisfying instance. The plots (a)–(c) represent the design space for the platform mapping example from Section 2. The plots (d)–(f) represent a relaxed design space where a number of design constraints from the platform mapping problem have been removed ; since this design space is less constrained, these plots exhibit a larger number of green samples than the plots (a)–(c). The plots (a) and (d) show results after the naïve algorithm was run.

The plots (b), (c), (e), and (f) were produced with the improved algorithm, but with a varying amount of randomness in the sampling procedure. When *ExploreII* is performed without randomization in sampling, the task of searching the design space for a solution is handed off entirely to the SMT solver. As a result, in each invocation, the solver is guaranteed to return a solution, if any exists. On the other hand, with randomization, there is a probability that the picked sample emits no solutions at all. Therefore, in some invocations, the solver may fail to return a solution. This is a trade-off for achieving diversity in the exploration; randomization may lead to unsuccesful invocations of the solver, but can help avoid clustering of the solutions that tends to appear when no randomization is used. We describe this trade-off in more detail in Section 5.3.

Let us first introduce background notations that are necessary to explain these plots. Let $[M] = \{M' | M \sim M'\}$ be the equivalence class represented by model M, and let $\Lambda = \{[M_1], [M_2], \ldots, [M_n]\}$ be the set of all equivalence classes, where the i^{th} class is represented by M_i. Then equivalence classes are partially ordered according to $[M_1] \leq [M_2] \Leftrightarrow M_1 \preceq_\Pi M_2$ (i.e. every member of the smaller equivalence class, $[M_1]$, is a homomorphic image of some member of the larger class, $[M_2]$). We plot the design space by dividing it into regions $R_1, \ldots, R_n$ such that: (1) For every $[M] \in R_i$ and $[M'] \in R_j$ it holds that $[M] \not\leq [M']$ and $[M'] \not\leq [M]$, for $i \neq j$. (2) Within a region R_i, there exists a greatest equivalence class $[M]^\top$: $\forall [M'] \in R_i,\ [M'] \leq [M]^\top$. These regions occur naturally due to models with zero occurrences of some record types, and can be identified uniquely by their greatest class. In our experiment the regions are:

$$\begin{gathered}
R_0 = \emptyset, \\
R_1 = \{Channel(x_1, x_2), Channel(x_3, x_4)\}, \\
R_3 = \{Binding(y_1, y_2), Binding(y_3, y_4), Binding(y_5, y_6)\}, \\
R_4 = R_1 \cup R_3
\end{gathered} \tag{6}$$

These regions are labeled in the figures using the short hand $\emptyset$, $\{f_0\}$, $\{f_1\}$, and $\{f_0, f_1\}$. The numbers at the top of each plot give the number of equivalence classes within regions.

Every equivalence class within a region is assigned a cell at some position along the y-axis. This position respects the $\leq$ order on equivalence classes. Since the number of classes per region grows rapidly, we shrink the cell size and split the y-axis into a number of columns per region. This setup means the plots exhibit two important properties: (1) The number and internal complexity of record instances increases from left to right and bottom to top. (2) Record instances that are homomorphically similar are physically nearby, except when a column is broken and wrapped into the next column.

5.2 Randomization

Our exploration algorithm should behave well over various types of design spaces, and there are two important factors to take into account. First, under ideal circumstances, equivalence classes should be sampled uniformly across the design space. Second, sampling must be able to adapt to design spaces that are highly constrained and therefore, contain only a few valid solutions. These goals may be contradictory, in which case a reasonable balanced should be achievable.

The first goal is costly to achieve because it requires a canonical representation for all non-isomorphic homomorphic images of the symbolic set used to generate φ. This representation cannot be explicitly constructed, as it grows too quickly in size. Instead, an effective random sampling procedure must generate equivalence classes cheaply, but without introducing too much bias in the sampling process. Though the full statistical analysis of this problem is outside the scope of the paper, we describe the intuition behind our solution.

Given a set of variables X, it is easy to generate a random partition of the variables. The structure of the partitions of X closely follows the *integer partitions* of $|X|$. An integer partition of n is a collection of integers that sum to n.

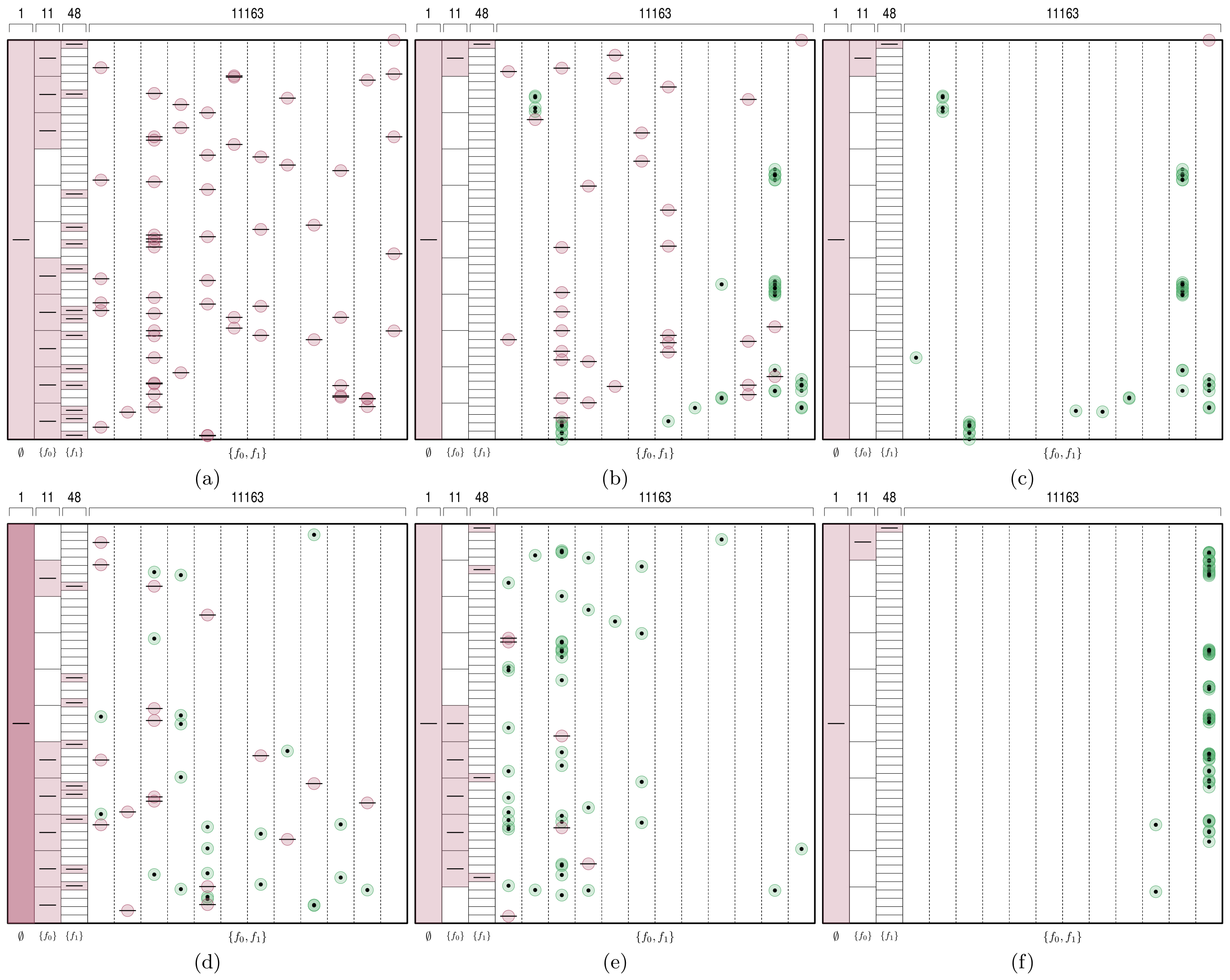

Fig. 9. Experiment results from running the exploration algorithms on the platform mapping problem. Each plot represents the design space; green dots mark explored samples that admitted a satisfying instance, and red dashes mark unsatisfiable samples. The plots (a)–(c) are results from the platform mapping example with all of the three design constraints described in Section 2; the plots (d)–(f) explores a relaxed design space. The plots (a) and (d) were generated using the naïve algorithm; the plots (b) and (e) using the improved algorithm (*ExploreII*) with $p_{gen} = 0.5$; the plots (c) and (f) using the improved algorithm with $p_{gen} = 1.0$.

Generating the integer partitions of $|X|$ in lexicographic order is also straightforward. Each integer partition serves as a template for building a random partition of X. For example, if there are three variables $X = \{x, y, z\}$, then the partition $[2, 1]$ means pick two distinct variables to equate, and then pick one variable (which is equated to itself). The number of possible partitions of X that fit a template grows exponentially with respect to the template's lexicographic order in the integer partitions of $|X|$ and then decreases exponentially.

We capture this behavior by fitting a normal distribution over the integer partitions of $|X|$. A random sample is constructed by first picking an integer partition as a template, and then randomly equating variables according to this template. The partitions are applied to the generator set to get a representative for an equivalence class. In our experiments, this approach removed an exponentially strong bias towards larger and more complex equivalence classes. The plots (a) and (d) in Figure 9 show the naïve random sampling algorithm with this correction applied. Our data suggests that this is a cost effective approach to sample design spaces with varying forms of symmetry in their generator sets.

5.3 Highly Constrained Design Spaces

Another difficulty arises when the design space is highly constrained. The plot (c) in Figure 9 shows all the solutions for the platform mapping under the generator set described in equation (5). In this case, there are only 41 non-isomorphic solutions out of 11,233, and these solutions are also highly clustered. By virtue of the solution set, no degree of random sampling can avoid the inherent clustering found here.

In order to address these cases, we add tuning probability p_{gen} to the *SampleClass* method. The method selects the generating set G with probability p_{gen} as the sample to search or a random sample with probability $1 - p_{gen}$ (using the technique described in Section 5.2). Recall, from the discussion of *ExploreII*, that the generating set is the most general sample, and so when the solver is invoked on this sample, a either a new solution is returned or the sampling process terminates. The plot (c) was generated by setting $p_{gen} = 1$, thereby enumerating a new solution with every invocation of the solver. The plot (a) was generated by running the naïve random algorithm on the same problem. Though the distribution of points is fairly random, none of these points hit a satisfiable equivalence class. Finally, the plot (b) was generated by setting $p_{gen} = 0.5$. Here we see the algorithm alternating between randomized and solver-driven exploration patterns. Our initial results suggest that 0.5 provides a reasonable default trade-off between finding diverse solutions and quickly finding some solutions.

However, $p_{gen} = 1$ is not a reasonable solution to DSE. We relaxed the constraints on the motivating example by removing conflict constraints and the relational/functional constraints on Channel and Binding. Under this relaxation there are 4298 equivalence classes with solutions, and these classes are more evenly distributed about the space. The plot (f) shows sampling the relaxed space with $p_{gen} = 1$. In this case, the solutions are highly clustered due to the solver's backtracking strategy. The plot (d) uses naïve random sampling and is even able to find some solutions. Finally, the plot (e) uses $p_{gen} = 0.5$ and finds a solution for almost every sample but does not exhibit the clustering behavior that is observed in the plot (f).

6 Related Work

6.1 DSE Frameworks

DESERT [26] is a framework that is closely related to FORMULA, with a goal of exploring design alternatives at the architectural level. Unlike FORMULA, design alternatives must be expressed as hierarchy of AND-OR choices with Boolean constraints describing interaction of design choices. DESERT encodes the design space and constraints symbolically, using BDDs [5]. However, the exploration in this framework is largely manual; the user specifies a constraint that should be true and DESERT prunes the design space accordingly. The user can also export points in the space to a modeling tool called the Generic Modeling Environment (GME).

CoBaSa [22] is a tool for automating the assembly of commercial off-the-shelf (COTS) components. It compiles system requirements and constraints among components into a pseudo-Boolean satisfiability (PBSAT) problem, which is tackled by a constraint solver. However, they focus on generating a solution that satisfies a large number of constraints, and do not focus on the exploration of the design space.

The technique developed by Hu et al. [14] collapses a multi-dimensional design space into a 3-dimensional space, after which the user selects portions of the space to explore with Cartesian co-ordinates. Their approach is similar to ours in that we both incorporate the user's feedback into the exploration. Their goal is not to enumerate distinct designs, but to find ones that are optimal with a particular fitness metric.

Kakita et al. [18] developed an algebraic approach for DSE of dataflow systems. In this approach, each point in the design space is defined as a dataflow graph. Graph rewrite rules are applied to an initial graph iteratively to generate a set of alternative designs that preserve the scheduling constraints of the original design. The authors tackle the problem of the exponential growth in the design space by representing regularity in structures with compact recurrence relations. This approach has been implemented in the METROPOLIS framework [1].

There have been a great number of works that focus on the goal of finding a set of globally optimal solutions, many of which are surveyed in [13]. We believe that our exploratory approach is complementary to theirs. In an early phase of the design, where the overall architecture of the system has not been clearly defined, coming up with optimization functions is difficult. Hence, it is desirable for the designer to sample and experiment with a diverse set of alternatives. In addition, even if objective functions can be defined, there may be a large number of optimal solutions, which would then be examined and evaluated per-basis.

6.2 Exploration Techniques

Tabu search [12] is a technique in combinatorial optimization that uses a memory structure to keep track of solutions that it has already visited in the search space. Our approach is similar to Tabu search in that it also store points that it has explored. In embedded system design, several groups [4, 11, 32] have evaluated Tabu search with respect to its effectiveness in finding globally optimal solutions. As far as we are aware of, Tabu search has not been used to exhaustively explore the design space.

Planning in AI solves the problem of finding a path between a pair of source and destination points on a search space. There is a wealth of literature on algorithms and heuristics to prune parts of the space that need not be explored. Some of these techniques, such as simulated annealing [31] and rapidly-exploring random trees [20], use random sampling to increase diversity during the exploration. Our random sampling approach was inspired by them.

Partitioning a large search space into equivalence classes has been employed in other areas of software engineering. In testing, the space of the test input can be partitioned based on a certain notion of equivalence, and only a single test case from each class needs to be executed [25]. In model checking [6], *state abstraction* can be used to group related states together, thereby reducing the size of the space that the model checker needs to visit. As far as we are aware of, our work is the first one to apply the partitioning method to explore distinct solutions in a design space.

In software product lines, researchers have studied techniques to explore and analyze the space of product configurations based on feature models [2, 3]. These works focus on managing variability in features, whereas constraints in FORMULA describe non-functional, architectural properties such as scheduling and security requirements. Constraint solvers use symmetry breaking predicates to avoid searching through solutions that are isomorphic to each other [8]. These predicates operate on low-level representations such as Boolean propositions, and do not take into account high-level domain knowledge.

6.3 DSL Specification Languages

A number of tools exist for specifying DSLs, with various degrees of automated analysis. We have already mentioned the work of DESERT [26]. Additionally, the Atlas Model Management Architecture (AMMA) [29] uses the OMG's meta-object facility (MOF) [27] as the DSL specification language. Abstract state machines (ASMs) are used to define the behavioral semantics of DSLs. These tools are built on top of the Eclipse Modeling Framework (EMF).

The KerMeta [24] framework provides a MOF compliant specification language. At the time of writing, KerMeta provides static type analysis and run-time checking of pre/post conditions. Additional formal methods are provided by exporting to other tools. The KerMeta language is inspired by the programming language Eiffel and is object-oriented in nature. It provides its own imperative language for specifying DSL behavioral semantics. The AToM^3 [9] framework is integrated with the Maude theorem prover [7]. AToM^3 focuses on behavioral and transformational DSL semantics. It uses the term rewriting formalism of Maude to evaluate LTL queries on models.

7 Discussion and Conclusion

In conclusion, we have presented an approach for exploring the design space of complex systems in our framework, FORMULA. Our framework combines results from domain-specific languages, symbolic execution, and automated theorem proving in order to quickly move from a specification of a design space to a set of distinctive solutions. Our exploration algorithms go further than finding non-isomorphic solutions; they attempt to quickly visit a diverse set of solutions across the design space. We have presented initial data to show the efficacy of our approach.

In the improved exploration algorithm (*ExploreII*), the predicate $\neg C$ can be considered to be a *complete* symmetry breaking predicate [8]—complete because it *guarantees* that the solver will not find any instance that is isomorphic to the ones that have already been discovered. But computing a full symmetry breaking predicate is generally expensive. In our case, the computation of a homomorphism from a sample to the ones in *valid* can be costly; no polynomial algorithms are known for this problem. An alternative approach is to use a *partial* symmetry breaking predicate, which provides a weaker guarantee but is much cheaper to compute [8, 23, 30]. We are currently investigating a way to integrate this partial approach into our framework.

We are also interested in studying other mechanisms to differentiate solutions in design spaces. In case studies for embedded systems, we encountered scenarios where a partial order over records would also be useful mechanism to distinguish solutions. This also raises interesting theoretical and practical questions on how to combine various differentiation mechanisms, and how to fairly sample and efficiently encode more general equivalence classes into an SMT solver.

The opportunities to combine DSE with rapid prototyping and optimization appear promising. The DSL approach in general, and FORMULA in particular, allow behavioral semantics to be assigned to DSLs. Thus, it is possible to automatically simulate and profile designs as they are sampled. Either the user or a utility function could rank solutions, and then these results could be used to further prune the design space. In this instance, pruning would mean adding more constraints to the SMT formula φ to refine the design space.

References

1. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.
2. D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

3. D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *CAiSE*, pages 491–503, 2005.
4. J. A. Bland and G. P. Dawson. Tabu search and design optimization. *Computer-Aided Design*, 23(3):195–201, 1991.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
8. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.
9. J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering, 5th International Conference (FASE)*, pages 174–188, 2002.
10. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
11. P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. In *Design Automation for Embedded Systems*, volume 2, pages 5–32. Kluwer Academic Publishers, 1997.
12. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Dordrecht, the Netherlands, 1998.
13. M. Gries. Methods for evaluating and covering the design space during early design development. *Integration*, 38(2):131–183, 2004.
14. X. Hu, G. W. Greenwood, S. Ravichandran, and G. Quan. A framework for user assisted design space exploration. In *DAC*, pages 414–419, 1999.
15. E. K. Jackson, E. Kang, M. Dahlweid, and D. S. T. Santen. Components, platforms and possibilites: Towards generic automation for mda. In *Embedded Software*, 2010.
16. E. K. Jackson, D. Seifert, M. Dahlweid, T. Santen, N. Bjørner, and W. Schulte. Specifying and composing non-functional requirements in model-based development. In *Software Composition*, pages 72–89, 2009.
17. E. K. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 2008.
18. S. Kakita, Y. Watanabe, D. Densmore, A. Davare, and A. L. Sangiovanni-Vincentelli. Functional model exploration for multimedia applications via algebraic operators. In *ACSD*, pages 229–238, 2006.
19. S. Kanajan, H. Zeng, C. Pinello, and A. L. Sangiovanni-Vincentelli. Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment. In *DATE*, pages 548–553, 2006.
20. S. M. LaValle and J. J. K. Jr. Randomized kinodynamic planning. *I. J. Robotic Res.*, 20(5):378–400, 2001.
21. L. Mandel and M. V. Cengarle. On the expressive power of ocl. In *FM'99*, pages 854–874, London, UK, 1999. Springer-Verlag.
22. P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *ISSTA*, pages 61–72, 2007.
23. I. McDonald and B. M. Smith. Partial symmetry breaking. In *CP*, pages 431–445, 2002.

24. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems, 8th International Conference (MoDELS)*, pages 264–278, 2005.
25. G. J. Myer. *The Art of Software Testing.* Wiley, 2004.
26. E. Neema, J. Sztipanovits, and G. Karsai. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.
27. Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.
28. G. Polya and R. C. Read. *Combinatorial Enumeration of Groups, Graphs, and Chemical Compounds.* Springer-Verlag, New York, 1987.
29. D. D. Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. RR 06.02, April 2006.
30. I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007.
31. P. J. van Laarhoven. *Simulated Annealing: Theory and Applications.* Springer, 1987.
32. T. Wiangtong, P. Y. K. Cheung, and W. Luk. Comparing three heuristic search methods for functional partitioning in hardware-software codesign. In *Design Automation for Embedded Systems*, volume 6, pages 425–449. Kluwer Academic Publishers, 2002.