The GDSE Framework: A Meta-Tool for Automated Design Space Exploration

Tripti Saxena and Gabor Karsai
Institute for Software Integrated Systems
2015 Terrace Place
Nashville, TN 37203
{tsaxena,gabor}@isis.vanderbilt.edu

ABSTRACT

Existing Design Space Exploration (DSE) frameworks are tailored specifically to a particular problem domain and cannot be easily re-used between domains. Typically these frameworks translate the DSE problem to a single formulation of the problem (e.g. ILP or CSP) and then solve it to retrieve satisfying alternatives. In order to compare the efficiency of different formulations/techniques on a given problem, the domain-expert has to manually reformulate the problem in another constraint language, which is time-consuming. In order to overcome this lack of reusability and flexibility in the current frameworks, we present here the Generic Design Space Exploration (GDSE) framework that allows the designer to solve DSE problems from different domains. Rather than using one strict formulation of the design problem, the framework supports a higher level formulation that can be mapped to different low level encodings. The main contributions of this framework are: 1) a generic representation which can be used to express any DSE problem, and 2) a flexible exploration technique which supports several exploration techniques.

1. INTRODUCTION

Complex embedded systems have a large number of choices in terms of selection of software components and hardware architectures for implementation. A set of possible design alternatives forms a *design space*. A valid design alternative must satisfy all constraints like throughput, latency etc. Each choice of software/hardware can have an impact on the functional and/or para-functional properties of the final implementation. *Design Space Exploration*(DSE) is the exploration of these design alternatives before actual implementation of the system. The goal is to search through a space of possible design alternatives and select designs that not only satisfy a given set of constraints, but are also optimal with respect to one or more objectives. This selection process is further complicated because the different design objectives might conflict with each other (e.g. area vs. latency).

Moya and Moya [13] identify three aspects of DSE: (i) Design Space Representation, (ii) Evaluation methods, and (iii) Exploration algorithms. In most existing frameworks all the three aspects are tightly coupled with domain-specific details. This leads to two main limitations: (1) **Lack of reusability**: DSE problems exist in different domains like signal processing, software product lines, Web server configuration, distributed software architecture, business processes, etc. Most DSE frameworks solve specific DSE problems within a domain. Although same exploration algorithms can be used to solve different DSE problems, the framework cannot be reconfigured to represent these problems. For instance, framework for software product line configuration cannot be used to solve software/hardware synthesis problems. (2) **Lack of flexibility**: A particular DSE problem can usually be solved in several ways, for example, product-line configuration problem can represented as a Constraint Satisfaction Problem (CSP) with finite domain constraints [3] or it can be represented as a boolean satisfiability problem [10]. The efficiency of a particular solving technique depends on the objective function, kind of constraints, as well as the size of problem instance. For example, an Ordered Binary Decision Diagram (OBDD) based solver, like [15] is better for pruning out infeasible configurations in a software product line, but CSP solvers are more efficient for searching an optimal configuration. Existing frameworks do not support this flexibility of choosing a different solver for every DSE problem instance.

In order to overcome these limitations we need a generic framework that can be reconfigured to represent DSE problems from any domain and also supports a set of exploration techniques that can be selected at run time. The main requirements of this generic framework:

R1. *Need for a generic representation:* The challenge is to have generic language that can be used to represent DSE problems from different domains. This language should allow the designer to project only those aspects of the design that are interesting from the DSE standpoint. Broadly, the representations used in the DSE frameworks can be categorized into Enumerative and Parametric representations. An enumerative representation of design space supports explicit enumeration of design alternatives whereas a parametric representation supports specification of ranges, which can be discrete or continuous. As the focus of this work is discrete space exploration, we restrict our focus to discrete ranges in parametric representation. Neema [15] uses a hierarchical version of the enumerative representation to structure the design space into a tree, where each design alternative is a path from the root to the leaf. Feature models [11] used to model the variability in product-lines are another example of enumerative representation. This representation is useful when the design points are not regular and it is difficult to parameterize the variations. On the other hand, parametric representation is used when the design variations are abstracted into single or multiple parameters. The cross-product of the domains of the configuration parameters forms a design space. A design alternative may be obtained from the design space by supplying appropriate values for the configuration parameters. A generic representation should support both representations of design spaces.

R2. *Need for an expressive constraint language:* Constraints are required to the capture the requirements that each valid design alternative should satisfy. We need a simple, yet expressive constraint language that can capture arithmetic, boolean as well as set constraints in the design space. The existing approaches like Integer Linear Programming (ILP) [17] and OBDDs support only a subset of the constraints. ILP can only handle linear arithmetic constraints while OBDDs can be used to express integer arithmetic constraints but are rather inefficient for doing so. Besides being expressive, the constraint language should be easy to use with the representation.

R3. *Need for solver independence:* Existing frameworks typically map this conceptual model of the problem to one formulation. For example DESERT maps the design space model to an OBDD formulation. Ideally, it should be possible to map the same conceptual model of the problem to different formulations and determine which formulation and solver works best for the problem at hand. This is also helpful when the same DSE problem can be solved using different formulations. Therefore, a generic framework to be able to support multiple solvers in the back end so that the modeler can experiment and determine which technique is the most appropriate for a particular problem.

R4. *Need for automation:* Encoding a complex design problem directly in a constraint language is a time-consuming task, which often requires detailed knowledge of the constraint language. This requires a number of iterations of formulation and testing. In order to increase the usability of the framework, a minimum level of automation is required. The generic framework should automate the translation of the high level design space model to the solver specific design model so that the modeler can extensively experiment without the need to learn a new language.

In this paper, we present the Generic Design Space Exploration Framework (GDSE), a meta-tool which can be configured to represent a DSE problem from any domain. In order to solve the DSE problem, the framework maps it to a model in Minizinc [16], a solver independent medium level language used to express combinatorial problems. The remainder of the paper is organized as follows: In Section 2 we present a case study which is used as running example to highlight the features of the framework. Section 3 provides an overview of the framework and a description of the generic representation and generic exploration, Section 4 discusses the related works, and finally, in Section 5 we conclude and discuss the future work.

2. BACKGROUND

2.1 Generic Modeling Environment

The GDSE framework adopts Model Integrated Computing (MIC) [18] as the core technology. MIC is based on the use of domain-specific modeling languages, metamodel composition, model transformation, and model synthesis; and it is supported by a suite of meta-programmable tools, including the Generic Modeling Environment (GME) [18]. GME is graphical modeling environment that allows the designer to capture the syntactic and semantic aspects of the application domain in a stereotyped UML-style class diagram, called the metamodel. The metamodeling syntax of GME is well documented in [12]. GME is meta-programmable, thus the same environment that is used to define domain *metamodel* is also used to build domain-specific *model* instances. This is done by using the metamodel specification to configure GME to present a domain-specific graphical modeling environment. Besides this, GME also supports creation of plug-ins and addons that operate on the models. The GDSE framework is built on GME, and uses the meta-programming feature of GME to reconfigure representation and satisfy requirement **R1**. A translator is plugged-in GME to translate a design space instance to a solver independent constraint problem, thus fulfilling requirement **R4**.

2.2 Minizinc

In order to solve the DSE problem, the framework maps the DSE problem instance to a model in Minizinc [16], a solver-independent medium level language used to express combinatorial search problems. The Minizinc model can be mapped to different solving techniques and solvers like constraint programming, mathematical modeling, etc. The Minizinc tool distribution includes a pre-defined translator which converts the Minizinc model to a low-level format (Flatzinc). It also provides translators to transform Flatzinc model to different solver specific languages. Moreover, Flatzinc is also supported by other external solvers like Gecode [19] and Eclipse [2]. The GDSE framework uses Minizinc distribution [1] to enable solver independent DSE and fulfill requirement **R3**.

2.3 Illustrative Example

We use a software product line configuration problem as a running example to highlight the salient features of the GDSE framework. A *Feature model* provides a compact visual representation of all the products of a software product-line in terms of features. A software product-line configuration involves selection/deselection of features such that the resulting set of products satisfy all constraints (for example, cross-tree constraints like feature A requires feature B, or performance constrain like product.cost <= 100).

3. THE GDSE FRAMEWORK

The GDSE framework decouples the representation, evaluation and exploration aspects of DSE, such that it is possible to configure the representation and use several exploration techniques to perform DSE. The framework consists of two components: (1) Generic representation, and a (2) Generic Explorer. The generic representation can be configured to represent DSE problem from any domain and thus satisfies the requirement **R1**. The GDSE Framework uses an extended-subset of the Minizinc language to express constraints. This constraint language is expressive and can express boolean, finite domain and set constraints, thus satisfying requirement **R2**. The flexible explorer supports a set of exploration techniques and satisfies the requirement **R3**. The framework includes translators to automate the configuration of representation and automatically transform the

DSE problem instance to a solver independent constraint problem specification, thereby satisfying requirement **R4**. Figure 1 shows an overview of the framework.

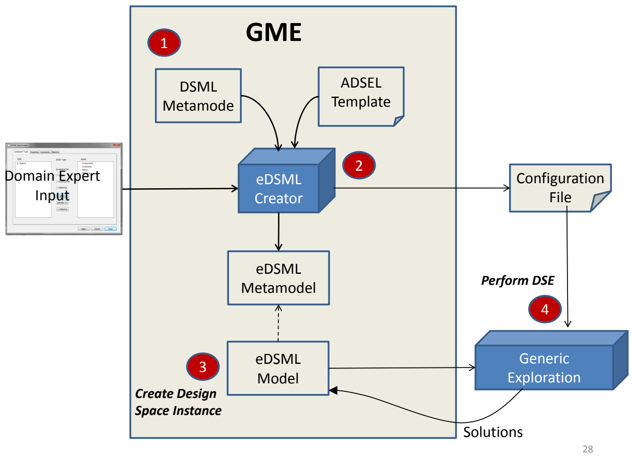

Figure 1: Overview of the GDSE Framework

3.1 Generic Representation

The meta-programming of the GDSE Framework involves two simple steps (1 and 2 shown in Figure 1):
Step 1: Create a Domain-Specific Modeling Language (DSML). This step is specific to each individual domain (see [18] for details). The DSML captures entities and their relationships in the domain. This step has to be performed once for every domain. Figure 2 shows the metamodel for capturing feature models.
Step 2: Extend the DSML to capture the design space, constraints and objective of a particular DSE problem. We call this extended version of the original language as eDSML. The extension is done by composition of DSML metamodel with a generic language template called Abstract Design Space Exploration Language (ADSEL), which is an abstract metamodel fragment that specifies a set of roles (the classes) representing common DSE concepts. The eDSML metamodel which is obtained as a result of this composition is then used to configure GME. This domain-specific version of the GDSE tool can then be used by domain-engineers to create an instance of design space with constraints and objective functions.

3.1.1 Abstract Design Space Exploration Language (ADSEL)

ADSEL is the core of reconfigurable representation in the GDSE framework. It allows association of DSE characteristics to elements of an existing DSML metamodel. This is done by metamodel composition of original DSML with ADSEL, where the extended DSML metamodel (eDSML) can instantiate (concretize and replicate) concepts in ADSEL while still using elements from original DSML. ADSEL Metamodel consists of three parts: (i) *Component Types*, which provide elements to create an enumerative or parametric design space, (ii) the alternatives in the design space interact with each other and this interaction is captured using *Constraints*, (iii) the *Function* captures the goal of the exploration, and (iv) the *Metrics* are used to hold any temporary values required during the exploration process. For example, `totalcost` is a variable required to represent the cost of the design point. We describe these in more details as follows:

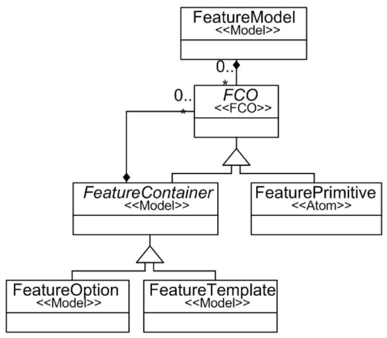

Figure 2: Metamodel for Feature Modeling

Component Types: ADSEL supports representation of both enumerative design space, where all alternatives are explicitly enumerated. The design space is a combinatorial product of the design alternatives. Figure 3 shows the component types of the ADSEL metamodel. The design space can be structured hierarchically with the *ExplicitDSElement*, where *Primitive* is a basic element representing a fundamental unit of composition and *Mandatory*, *Alternative*, *Option*, and *Or* are internal nodes. The *Mandatory* class models composition, which means all the objects contained in a *Mandatory* object are included in the design if the parent is included. The *Alternative* class models a choice point where each child object represents one alternative and exactly one is selected if the parent is selected. *Or* models a generalization of *Alternative*, where if the parent is selected then a set of child objects between `min` and `max` cardinality are selected. The *Option* class models option such that if parent is selected then zero, one or any combination of child objects can be included. *ParametricDSElement* is used to create parametric design space. If should contain at least one property that is a decision variable, which is fixed as a result of the exploration. The design space is contained in *DesignConfigurations* class.

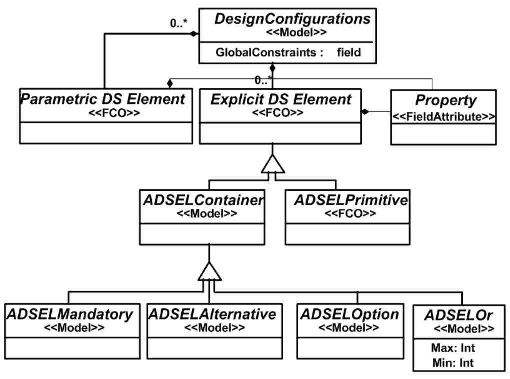

Figure 3: ADSEL Component Types

Properties: ADSEL properties are used to capture the parameters, that are of interest for DSE and set before exploration, as well as decision variables, which capture the dimensions of the design space and are set as result of the exploration. A Property Class includes attributes: (i) `name`, (ii) `propertyType`, which specifies whether the property is a `decision variable` or a `parameter`, (iii) `valueType`, which captures the type of value the property variable can take (scalar type like `INT` or `BOOL`, or `CUSTOM` for user customized domains), (iv) `domain`, which captures the possible values the variable can take (exact value in case of `parameter` variable, or a range in case of `decision` variable, or Set of values in case of `CUSTOM`, and finally (v) `composition`, which represents a recursive formula applied during the design space exploration to calculate system-level property values for enumerative design space. For example, if the property composition is `Add`, the property value of the *Mandatory* object is the sum of the property values of all the contained objects.

Constraints: ADSEL specifies two constraint classes, shown in Figure 4: (i) *GraphicalConstraint*, which models a dual context constraint used to impose a relation between two objects in the design space tree. (example, A requires B), and (ii) *TextualConstraint*, which models single context constraint (example, `A.memory` $\leq$ 128). Besides these constraints, ADSEL also specifies *Globalconstraint*, which is applicable to all alternatives in the design space (example, `forall(m in Module) (m.resourcetype != 0 ))`. The constraint objects have an `expr` which captures the constraint definition, written in extended subset of Minizinc language.

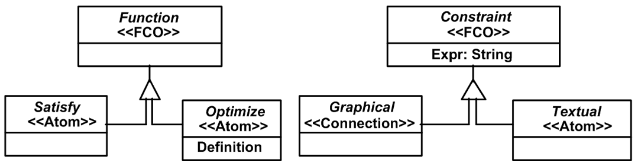

Figure 4: Metamodel of ADSEL

Objectives: These capture the goal of the exploration process. *Satisfy* is used to perform constraint-based DSE where the goal is to find design alternatives that satisfy the constraints. *Optimize* provides a placeholder for specifying the cost function used to compare the alternatives in the design space. `Definition` is used to specify which properties/metrics are maximized/minimized.

3.1.2 Metamodel Composition

Metamodel composition enables the reuse of an existing DSML specification to rapidly construct an extended version with DSE characteristics superimposed onto the original language elements. We focus on the Template Instantiation [8] method of metamodel composition, where we instantiate the elements of the abstract metamodel template (in our case ADSEL) in the extended DSML metamodel (eDSML). The composition of DSML with ADSEL is done by creating new inheritance relationships from the ADSEL elements to the pre-existing elements in the original DSML, which forces them to play the roles of the ADSEL elements. These new inheritance relationships are included in the eDSML metamodel. Figure 6 shows the eDSML metamodel (called eFeature) resulting from the composition of *FeatureMeta* with ADSEL.

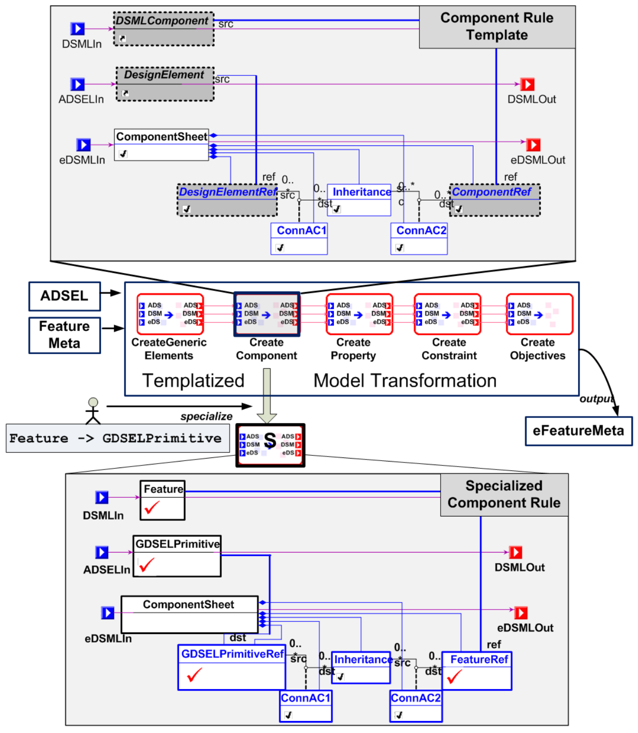

Figure 5: Metamodel composition using Templatized model transformation

The classes with dotted outlines are part of ADSEL and the one with solid lines belong to the *FeatureMeta*. The *FeaturePrimitive* class plays the role of *ADSELPrimitive* class from the template. If the original DSML metamodel does not contain a suitable class for playing a given role then a new class with place-holder name can be added in the eDSML metamodel. For example, the *FeatureMeta* does not contain a class to play the role of proscribed by *Mandatory* class, so *NewMandatory* is added.

We have automated the metamodel composition process with the help of a templatized model transformation. The tool support for metamodel composition helps the domain-experts to quickly generate the eDSML metamodel and configure GME. Figure 5 shows a graphical representation of the templatized model transformation, which consists of sequence of transformation rules starting with a rule that creates generic elements in eDSML (for example, *DSComponent*, *DSContainer*). This rule is applicable to all DSMLs and does not need to be specialized. The domain-expert specializes the template by supplying a map through the User Interface (UI). The map indicates which DSML component is included in the eDSML metamodel and which ADSEL element role it should play. For example, the domain-expert enters the map `Feature -> ADSELPrimitive`, which specifies that the `Feature` is included and it should play the role of the `ADSELPrimitive`.

The transformation template rule (shown in Figure 5 above the templatized model transformation) has place holders for elements from DSML and ADSEL, which are replaced by the actual elements in the specialized transformation rule

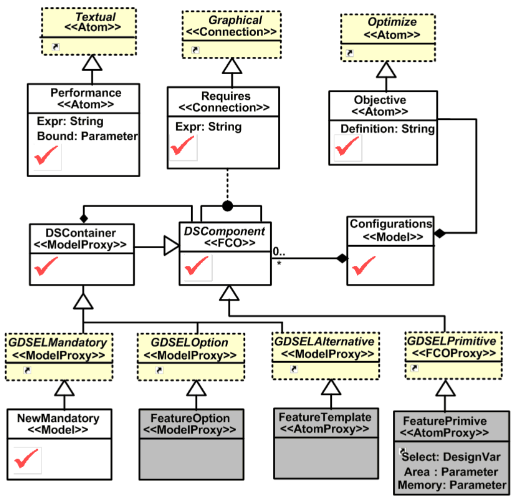

Figure 6: Autogenerated extended-Feature Metamodel

(shown in Figure 5 below the templatized model transformation) depending on the map. For example, the `Feature -> ADSELPrimitive` map replaced the *DSMLComponent* and *DesignElement* abstract classes shown in the *Component Rule template* with *Feature* and *ADSELPrimitive* classes respectively. The specialized model transformation consists of a sequence of specialized rules, where there is one specialized rule for each (DSML, ADSEL) pair provided by the domain-expert.

Figure 6 shows the autogenerated *eFeature* metamodel, where the dotted outlined elements are references to ADSEL classes, the dark elements are references to *FeatureMeta* classes and the elements with tick marks are new classes. The *FeatureMeta* was created to capture the product lines but did not have elements which can capture the constraints and objectives required for product-line configuration. Therefore, new classes (*Performance*, *Requires*) are added to capture the constraints. A default constraint expression is specified by the domain-expert. For example, default `expr` for *Requires* constraint is `$src.Select` $\rightarrow$ `$dst.Select`). When the domain-engineer creates an instance of this constraint in his model, he does not need to write any constraint expression.

Once the specialized transformation is executed on the DSML, the extended version is automatically generated and the GME is configured using this eDSML. The domain-engineer can then model his DSE problem and perform DSE with minimal efforts.

3.2 Generic Exploration

In order to find valid design configurations that satisfy constraints and are optimal with respect to a certain cost function, the GDSE framework has a generic exploration engine which supports several solvers. The main aim of this flexible solver backend is to overcome the limitations imposed by a single formulation exploration and enable the designer to experiment with different solvers. In order to do so, we propose a two-stage translation of eDSML design space models into low level constraint satisfaction problem.

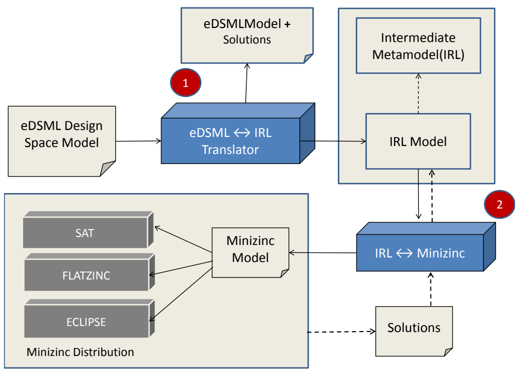

Figure 7: Generic Exploration of GDSE Framework

```
Minizinc Code Snippet
int:  n = 13;%-----variables ---
array [1..n] of var 0..1:  Sel ::  is_output;
array [1..n] of var int :  Memory::  is_output;
 ...
constraint%-----requires constraints -
  (Sel[1] == 1) -> (Sel[2] == 1);
 ...
constraint %-----Performance constraints -
  (Memory[4] <= 2048);
 ...   constraint %-----Property constraints -
  (Sel[3] == 1) -> (Memory[2] == Memory[3]);
...
solve ::  int_search(Sel, first_fail,indomain_min,
    complete )satisfy;
```

Figure 7 shows the details of the generic exploration in the GDSE Framework. After the design space model has been created, the domain engineer invokes the exploration backend with a solver he wants to use. The generic exploration is supported by two solver independent formats, the InteRmediate Language (IRL) format and the Minizinc. The first step in the backend transforms the design space model to a model in IRL. This IRL is also expressed as a metamodel in GME, which distills all the non-DSE related information. In future we intend to use IRL to extend solver support, for example convert IRL models to take advantage of metaheuristics techniques like simulated annealing, hill climbing. At present a domain-independent transformation sanitizes the eDSML model to get IRL models. Another domain-dependent transformation is written to convert IRL models to eDSML models in order to reflect the solutions back in our original design space model.

The second step transforms the IR model to a model in Minizinc, which can be mapped to different solving techniques and solvers like constraint program, mathematical modeling, etc. A snippet of a Minizinc model generated corresponding to an instance of product-line configuration problem has been included is shown in Table 1. In the current framework, the designer can choose the solver from {Flatzinc, LazyFD, Gecode} solvers. Many other solvers like Eclipse Interval Solver [2], MiniSAT [9] solver and SMT solver [6] can also be used but have not been tested yet. The designer can also choose to feed back the solution models into the design space model, if a small number of solutions are obtained.

4. RELATED WORK

DESERT [15] is a DSE framework that is aimed at performing early design space exploration in embedded system design. Design alternatives are represented hierarchically as an AND-OR-LEAF tree and the Object Constraint Language (OCL) is used to express constraints. The design tree and the constraints are symbolically encoded and OBDDs and space pruning is performed by conjunction of design space OBDD with the constraint OBDDs. The primary advantage of this approach is that it is exhaustive and is useful for pruning large spaces, although the approach does not scale well in the presence of continuous finite domain variables [14]. **DESERT-FD** [7] is a DSE framework developed to overcome the limitations of DESERT. It contains a hybrid solver which combines symbolic constraint satisfaction of DESERT with finite domain constraint satisfaction. DESERT-FD also supports a property composition language that allows the user to write custom property composition functions. Both DESERT and DESERT-FD frameworks are domain independent frameworks and use a single encoding mechanism to perform DSE. These frameworks can be used to express DSE from different domains but do not allow the user to experiment with different solving techniques.

PISA [4] is a framework that uses an evolutionary multi-objective search algorithm to perform DSE. The search algorithm is implemented as set of separate communicating processes. PISA is better than other frameworks in terms of flexibility since it supports separation of the specification of the DSE exploration problem from the exploration algorithm. The framework proposed in this paper differs from PISA because it supports different solving techniques in the back end.

EXPLORA [5] is a Java based tool which enables generic DSE by providing support to integrate different optimization algorithms, cost functions and synthesis tools. The advantage of this tool is that it can perform DSE at different abstraction levels but the reconfiguration is done programmatically.

5. CONCLUSIONS

In conclusion, we presented a novel framework for enabling DSE. Our framework leverages domain specific modeling and metamodel composition technique to create a meta-tool capable of expressing and solving a DSE problem in any domain. The configuration tool support allows the domain-experts to quickly configure GME, such that instances of the DSE problem can be created by domain-engineers and solved using a suite of solvers supported by the backend.

6. REFERENCES

[1] Minizinc distribution. `http://www.g12.cs.mu.oz.au/minizinc/`, 2000-2004.

[2] K. R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse.* Cambridge University Press, New York, NY, USA, 2007.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Reasoning on Feature Models. In *LNCS, Adavanced Information Systems Engineering: 17TH International Conference, Caise 2005*, page 2005. Springer, 2005.

[4] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - A Platform and Programming Language Independent Interface for Search Algorithms. pages 494–508. Springer, 2003.

[5] F. Cieslok, H. Esau, and J. Teich. EXPLORA - Generic Design Space Exploration during Embedded System Synthesis. In *DIPES '00: Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, pages 215–226, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[6] B. Dutertre and L. D. Moura. The YICES SMT Solver. Technical report, 2006.

[7] B. K. Eames, S. K. Neema, and R. Saraswat. DesertFD: A Finite-Domain Constraint based tool for Design Space Exploration. *Design Automation for Embedded Systems*, 2009.

[8] M. Emerson and J. Sztipanovits. Techniques for Metamodel Composition. In *OOPSLA Ű 6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[9] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] M. Janota. Do SAT Solvers Make Good Configurators? In *SPLC (2)*, pages 191–195, 2008.

[11] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[12] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of the IEEE*, pages 145–164, 2003.

[13] F. Moya and M. Moya. Evaluation of Design Space Exploration Strategies. *EUROMICRO Conference*, 1:1472, 1999.

[14] S. Neema. *System-Level Synthesis of Adaptive Computing Systems.* PhD thesis, Vanderbilt University, May 2001.

[15] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In *EMSOFT*, pages 290–305, 2003.

[16] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[17] A. Schrijver. *Theory of Linear and Integer Programming*, chapter 15.1 : Karmarkar's Polynomial-Time Algorithm for Linear Programming, pages 190–194. John Wiley & Sons, New York, NY, USA, 1986.

[18] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *Computer*, 30(4):110–111, 1997.

[19] G. Tack. *Constraint Propagation - Models, Techniques, Implementation.* Phdthesis, Saarland University, Germany, 2009.