A GENERIC FRAMEWORK FOR DESIGN SPACE EXPLORATION

By

Tripti Saxena

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August, 2012

Nashville, Tennessee

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Gabor Karsai, who has done the best possible job in advising me during this work. Needless to say, this work would not be possible without his constant support and encouragement. Thanks Gabor for believing in me, even when I didn't completely do so myself. Also many thanks to Dr. Sandeep Neema for introducing me to the area of design space exploration and for the financial support during the initial years of my graduate studies at Vanderbilt. I would also like to thank Professor Douglas Schmidt, Professor Christopher Jules White, and Professor Bharat L. Bhuva for serving on my dissertation committee. I appreciate their guidance and advice.

This work was supported by GM, Vanderbilt, DARPA, and NSF under the Producible Adaptive Model-based Software and Lattice QCD SCIDAC: Optimization of Fault Mitigation for Large Commodity Clusters projects. Therefore, I would like to extend my gratitude to all for their generous financial support that has allowed me a lot of freedom to work on the topics I liked. I would especially like to thank Dr. Abhishek Dubey for all the patience while working together on the Fault Mitigation project.

I would like to thank Professor Aniruddha Gokhale for his encouragement during the time spent at Vanderbilt. I also want to thank my colleagues at ISIS, in particular Joseph Porter for his help with different topics from graphs to parsers, as well as numerous invitations for Christmas dinner. Many thanks to Daniel Balasubramanian for his constructive criticism and for many interesting discussions, both work and non work related. Thanks to all my friends in Nashville, in particular Jia Bai, Shweta Sharma, Pan Pan, Pooja Varshney, Aparna Barve, and all the others that I have not mentioned. A special word of appreciation for my friend Kalid K. Waren, thanks for being there for me through the thick and thin of grad life.

Last, but certainly not least, I would like to thank all the members of my family, in particular my parents Ramji and Rajni and my sister Divya, for their unconditional support, their advice and friendship. I also want to thank my husband Hiral and my in-laws Usha and Rajni for sharing many of my experiences during the final year at Vanderbilt.

Thank you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A software system is formed by a set of intercommunicating hardware and software components. Design of software systems involves numerous design choices, such as selection of software components, hardware architecture, assembly of the components, etc. The design choices are made to ensure that the final implementation of the system meets a set of design requirements (*constraints*) and is most suitable with respect to a quality metric such as cost or dependability. However, as the complexity of the software system rises, the designers are faced with the challenge of identifying designs that meet the design requirements in the presence of increasing number of components.

A *design space* of functionally equivalent design alternatives is created by the combination of existing design choices in the design process. Evaluating each design alternative in this space is a complex task as the design space can become large very quickly, especially if arbitrary combinations of design choices are allowed. For example, assume an application with t computational tasks is to be mapped on to a platform consisting of p general-purpose processing blocks. This means that each task can potentially be mapped to any of the blocks leading to a design space of p^t design alternatives. The complexity further increases if there are multiple conflicting requirements that the design solution must satisfy, for example memory consumption and power dissipation. In this case, all objective values of a design point must be compared with the corresponding objective values of every other design.

Design Space Exploration (DSE) is the process of searching through the design space to identify design solutions that satisfy all design requirements and are most suitable based on quality metrics such as cost, memory use, etc. DSE is a challenging problem, often found in the application areas (domains), such as embedded systems [94], software product-line engineering [17], digital signal processing [85, 87], mobile ad hoc networks [129] etc. In the past the designers have chosen to interactively search using a sequence of decisions. Each decision prunes out a set of design alternatives and reduces the size of the space. However, design is a complex activity where a unidirectional series of decisions do not always work. The designer typically might have to retract a previously taken design decision to explore other regions of the design space. Moreover, as the size of the space increases, manual exploration becomes impractical and automated approaches to DSE become imperative.

A common approach to automate DSE is to write a customized algorithm [66, 94] to evaluate design alternatives, and search the space for a particular problem. Although customized algorithms work efficiently for the problem under consideration, this approach is not only time consuming but also hard to maintain if the problem changes over time due to addition, deletion or modification of objectives and constraints. In this case reusing the same algorithm requires the user to deeply examine the code to identify the problem-specific parts and rewrite these parts. This task is often time consuming and erroneous. Consequently, there is a strong need for a more systematic approach to automate exploration that can be easily maintained and reused.

The need for a disciplined approach to automate DSE has motivated the designers to use well-defined search methods drawn from artificial intelligence and operations research. Examples of these methods are constraint programming [83], mathematical programming [109], metaheuristics algorithms [22], boolean satisfiability [37] etc. Given the variety of search methods, they can broadly be classified into exact and approximate algorithms [123], where exact algorithms exhaustively evaluate all design alternatives in the design space, and return a guaranteed optimal solution, whereas approximate algorithms generate high quality solutions in reasonable amount of time. The choice of a particular method depends on the motivation for DSE.

A key advantage of using a search method to automate DSE is the generic scope of these methods. For example, constraint programming has been used to solve combinatorial problems in many domains from hardware design (placement and layout), financial decision (portfolio management), computational biology (DNA sequencing), etc [114]. Another advantage of using standard search methods is the availability of well-tested and documented libraries [30, 122, 127]. These libraries provide constructs for specifying and solving a problem with the supported search method. This enables the designer to reuse code from the libraries to develop an algorithm to automate DSE in a relatively short amount of time. Moreover, the resulting algorithm is more reliable and maintainable in the face of evolving problems. Over the years, research in the field of artificial intelligence has lead to creation of efficient software packages (solvers) [2, 3, 44] based on search algorithms. Solvers support a declarative syntax to model a search problem, which is relatively easier to use as compared to libraries that require an imperative coding language.

Given the range and power of search methods, an on-going challenge is to use these methods to automate DSE. A typical approach for solving any new search problem using general purpose search methods is to formulate an abstract mathematical model of the problem. This model is then processed by the solver to return a solution. However, formulation of mathematical models for real-world problems is a challenging and time consuming step that relies heavily on the experience and expertise of the designer. While creating the mathematical model for a given problem, the designers may look at models of similar problems for inspiration before they can effectively formulate their problem. The task of creating a mathematical model for a domain specific *DSE problem* is even more challenging, as the conceptual gap between the informal problem description and the mathematical model is too large even for the experts.

With these considerations in mind, the thesis of this dissertation is to develop a systematic approach and architecture that can reduce the complexity of leveraging standard search methods to automate exploration of design space. Such an approach should facilitate modeling of DSE problems at a high-abstraction level, that can be solved without requiring expert knowledge of the search methods and the tools used in the actual search.

I.1 Motivation

A *DSE problem* is essentially a search problem, where the goal is to find a valid and optimal solutions given a design space, quality metrics and a set of validity constraints. Literature survey reveals a vast body of work that exists to address the challenge of solving DSE problems. Most existing work is focused on solving a given DSE problem from a given domain. This work is motivated by the limitation of the existing domain-specific approaches for automated DSE.

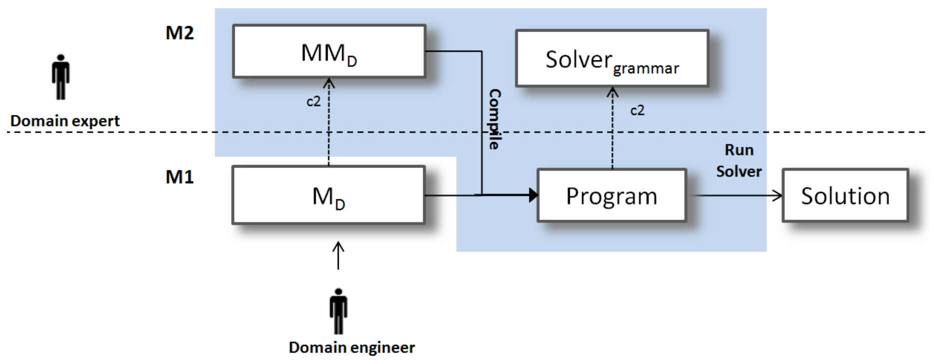


Figure 1: Generic Architecture of a DSE Framework

A common approach for solving a class of DSE problems in a domain is to integrate a general purpose solver with a domain-specific design environment to create an environment that supports automated DSE for the modeled problems. Figure 1 shows the generic architecture of domain-specific DSE frameworks. These frameworks consist of 2 salient features: (a) a domain-specific modeling language MM_D, and (b) a model compiler. The language captures the main concepts of the class of DSE problem under consideration, such that the language can be used to specify problem instances belonging to the class. For example, MILAN [14], a domain-specific DSE environment for embedded systems, supports a language for resource and application modeling. A problem model M_D can be created by instantiating the concepts in the language. For example a problem model in MILAN consists of a list of resources and tasks. The model compiler translates the problem model M_D, into a format that can be processed by the general-purpose solver. This compiler is created by the *domain experts*, who have in-depth knowledge of the domain as well as the search methods. The domain-expert tries to identify a recurring pattern used in formulating a class of DSE problems in the domain and uses this pattern to build the model compiler. This compiler is customized to work only for the problems in the class.

In order to use the framework, a *domain engineer*, who has minimum knowledge of search methods, creates a model M_D for an instance of of the DSE problem. The problem model M_D includes the design space and validity constraints that the solutions should satisfy. The framework automatically processes the data from M_D using the solvers to return a solution. A number of such frameworks exist [15, 20, 84, 124]. An obvious advantage of this approach is that it enables the domain-engineers to model and solve DSE problems using high level domain-specific notations with minimal knowledge of the search methods.

An obvious concern with the framework-oriented approach is the time and effort required to build a DSE framework. Building a set of reusable classes and an efficient compiler require a number of iterations, where the classes and compiler are customized. Once the framework has reached a level of efficiency, the framework is packaged as a black-box framework that can be used by the domain-engineers without knowledge of the compilation or the search algorithm involved to perform the actual exploration. However, any changes in the requirements of the problem require a non-trivial amount of effort to change the compiler and classes to ensure that the instances conforming to the new problem are modeled and solved correctly.

Another concern with the existing approach is the complexity of writing validity constraints. Existing frameworks require the domain-engineers to specify validity constraints using a constraint specification language, for example OCL [49]. The efficiency of the search depends on the formulation of these constraints. Writing the constraints so that the resulting mathematical model is efficient can be a daunting task, especially for an engineer who does not have any expertise in optimization methods. Ideally, the responsibility of writing constraints should be given to the domain-experts who have the technical knowledge of how to write constraints that work best for the class of DSE problems under consideration.

However, a major limitation of existing DSE frameworks is that they are tailored towards solving *a single class of DSE problems* in *a given domain*. This domain specificity comes from three sources: (a) domain-specific modeling language that is used to model the DSE problem; (b) the model compiler that is highly-specialized for the class of DSE problems that the framework is intended to solve; and (c) the general purpose solver that is selected on the basis of the requirements of the problems.

The domain-specific modeling concepts in the language supported by the design environment precludes the use of the DSE framework to model similar problems from other domains. For example, resource allocation problems have been found in embedded system [61], network design [117], product-line engineering [131], yet no single framework can model and solve all three problems, even though the general purpose solvers that are integrated with the framework are powerful enough to solve them.

The problem-specificity of the model compiler restricts the modeler from using the framework to solve another DSE problem *within the same* domain. For example placement and routing problems in VLSI design. Even though the modeling notations supported by the framework can be used to model both the problem and the general purpose solvers can solve both problems, the model compiler makes the framework incapable of handling instances of both problems as it is customized for a given problem only.

Lastly, a DSE problem may evolve over time and addition of new constraints and objectives can render the existing solvers inefficient. For example, the objective might initially be to retrieve valid design alternatives. This is successfully done using a model compiler that takes the design space model and translates it to a format that can be solved by constraint satisfaction solvers, like MiniSat[37], to retrieve satisfactory solutions. However, this approach fails when one or more objectives are added to the problem and the goal is to retrieve a set of optimal solutions.

Given the time and effort required to develop frameworks for automated DSE and the power of general purpose solvers, it would be advantageous to have a unified framework to *model* and *solve* DSE problems from a variety of domains, so that it can be exploited by a large number of users. This framework should retain support for high-level modeling, so that the engineers can model and solve DSE problem easily.

I.2 Challenges

Developing a generic DSE framework that supports modeling and solving DSE problems from a variety of domains has the following challenges:

The first challenge is to develop a language that can capture design spaces of problems from a wide range of domains. A survey of the literature showed two categories of design space representations: (1) *enumerative* representation, and (2) *partial model* representation. An enumerative representation explicitly includes all alternatives in the design space. An example of this representation is found in product-line configuration problems [20, 90], where the design alternatives are organized in a tree structure (feature model) and a design alternative is a set of paths from the root to a leaf. On the other hand a partial model representation [57, 68] includes an incomplete model, where the design space is the set of all well-formed models created by completing the model. A generic framework should be able to support both representations. An additional requirement on the modeling language is that it should be easy to use both for the domain-experts and engineers. Therefore, the generality of the representation should not add to the complexity of using the language.

Another challenge is to develop a language to specify the constraints in the DSE problem. Constraints are essentially validity conditions that each valid design alternative should satisfy. These constraints are required to guide the exploration of the design space towards valid solutions. Typically in DSE frameworks, high-level constraints are expressed along with the model of the design space. These constraints are refined to lower-level languages like propositional logic or relational algebra to guide the solver towards valid solutions. A generic framework should therefore support a constraint specification language that can specify a wide range of constraints, while being easy to refine to lower-level languages. Existing frameworks, especially the MDE-based framework [90] use high-level constraint specification languages, like Object Constraint Language (OCL) [49] or its variants to specify constraints. OCL is expressive and user friendly, but the intricacies of OCL can complicate the automated analysis of arbitrary OCL constraints [82]. On the other hand a number of formal specification languages like Alloy[56], B[6] have been used to specify consistency constraints on UML model. These languages have a formal mathematical basis and are therefore easy to translate to lower-level languages because of their mathematical foundation, but there is a steeper learning curve in writing correct constraints using these languages. Therefore, the challenge is to develop an expressive constraint specification language that is user friendly like OCL but is easy to analyze like the formal specification languages.

A third challenge is to be able to solve a wide range of DSE problems. Most existing frameworks are built on a single search method that is used to solve the DSE problem. This is a limitation because the constraints and objectives of the exploration might change as the problem evolves over time. A generic framework supports modeling of DSE problems with a variety of constraints and in order to solve all modeled problems, the framework should support a set of optimization and constraint satisfaction methods that can solve a rich set of DSE problems.

I.3 Thesis Statement and Contributions

The objective of this work is to create a *generic framework* for automated design space exploration that can be used to model and solve DSE problems from a variety of domains. The focus is on reducing the development cost involved in solving design problems with discrete design spaces. In order to meet this objective, we developed a model-based DSE framework that can be configured to create domain-specific DSE environments. The key idea is that the domain-experts can *meta-program* the generic framework to work for a class of DSE problems in a given domain. This configuration results in a domain-specific DSE environment that can be used by the domain-engineers to model and solve an instance of the DSE problem. The main feature of the framework it its ***generic scope***, such that it can be used to model and solve exploration problems from a variety of domains. The framework includes:

- generic reusable classes, that can be specialized to configure the framework to support modeling a class of domain-specific DSE problems.
- a simple yet expressive constraint language that is used to specify a variety of validity constraints.
- solver support to solve the problems modeled using the framework, such that it is possible to select a solver according to the requirements of a given problem model (multi-objectives, constraint satisfaction etc.)

This framework is intended to be exploited by as many users as possible and to that effect, most design decision of the framework were taken to enable ease of use by both the engineers and experts. The goal of this work is to reduce the development cost involved in solving new DSE problems.

The most important contribution of our approach is the generic modeling support of the framework. Our approach generalizes the existing approaches by decoupling the DSE aspects of a DSE Problem (objective, constraints etc.) from the domain-specific concepts (task, processor etc). The domain-independent DSE concepts are captured as a set of abstract classes that are generic enough to be applicable to a wide range of domains. These classes embody an abstract model of a DSE problem. The domain-experts inherit/specialize these classes to configure the framework for a class of specific domain-specific DSE problems. The abstract classes include: (a) classes for modeling the design space; (b) constraints classes to model the validity constraints for a design alternative; and (c) objective classes to capture the goal of the exploration. Together these classes are referred as the *Abstract Design Space Exploration Language* (ADSEL).

A framework that supports generic modeling of DSE problems also requires an expressive language to specify the conditions (constraints) that each valid design solution should satisfy. Another contribution of our approach is the expressive and easy to use *Constraint Specification Language* (CSL) to specify these constraints. This language can be used by the experts to specify validity constraints. Additionally our approach supports constraint templates, where the domain experts specify constraint templates that can be easily used by the domain engineers by plugging in values. This simplifies the time-consuming and error prone task of writing constraints. CSL is expressive without being excessively verbose and can be used to write boolean, set and arithmetic constraints.

The third contribution of this work is the development of the *Intermediate Language* (IRL), a solver-independent abstraction layer that decouples the problem specification from the search method used to solve the problem. The IRL supports simple set theoretic concepts like sets, functions and relations that can easily be refined to concepts in lower-level mathematical languages (example constraint programming). A model in the IRL is refined to a set of solver-specific models. The framework supports two solver-specific models: (1) CP Model for *mono-objective* optimization using constraint programming and (2) MOP model for *multi-objective* optimization using meta-heuristics. The solver-specific models are processed by the respective solvers in conjunction with the data generated from the problem model to return valid design solutions. The refinement of DSE problem definitions, specified using ADSEL classes and CSL constraints is automated using a set of translators. This approach of using multiple solvers empowers the domain-engineer to choose a search technique to solve a problem instance. This is in contrast to the current approaches that hard code a single search technique that is used to solve all problem models belonging to the class.

I.4 Outline

The rest of the dissertation is structured as follows:

- **Chapter** II introduces the necessary background knowledge. It presents a summary of the most commonly used search methods for automation of DSE, and then presents a survey of selected DSE frameworks.
- **Chapter** III presents a detailed view of the architecture. The framework includes a series of operations that cover the entire search process, starting with problem specification, translation to lower abstractions, and retrieval of solutions. We discuss the languages and translators included in the framework in detail.
- **Chapter** IV: We validate the scope of our approach by a set of case studies and summarize the extent of the success.
- **Chapter** V summarizes the results and discusses future work.

CHAPTER II

BACKGROUND

A large body of work exists on automated DSE in a variety of domains. This chapter presents background material on search techniques used for automating design space exploration. The search techniques are drawn from research in the fields of artificial intelligence and operations research. In order to solve a search problem using a standard search technique, the problem is formulated as an abstract mathematical model and then a search procedure is used to process this model to generate solutions. We evaluate each technique on the basis of the mathematical model required and search algorithms supported. Although many classifications are possible, we classify the search methods into three classes: (1) constraint satisfaction, where the objective is to retrieve one or more valid solutions, (2) mono-objective optimization, where the objective is to find a single global optima, and (3) multi-objective optimization, where the goal is to retrieve a set of comparable solutions.

This chapter also presents a selected set of domain-specific and generic DSE frameworks that are relevant to our work.

II.1 Methods for Constraint Satisfaction Problems

II.1.1 Constraint Programming

Constraint Programming (CP) [102] is a paradigm for solving combinatorial search problems. It has been widely used to solve practical problems from network design [7], VLSI design [107], scheduling [71, 99], product-line configuration [18], etc. Solving a problem using constraint programing involves two steps: (1) *modeling*, and (2) *solving*. We discuss each of the steps in detail in this section.

Modeling involves formulating a problem as a Constraint Satisfaction Problem (CSP). A

CSP model consists of a set of variables, a set of finite domains for the variables and a set of constraints restricting the values of the variables. Formally, an instance of a CSP is a triple

$$\langle X, D, C \rangle \tag{II.1}$$

where

- X is a set of decision variables $X = \{x_1, ..., x_n\}$,
- D is a set of possible values that each variable can take and consists of a *finite* set D_i of possible values for each variable x_i,
- C is a set of constraints over the set of variables, which consists of mathematical or symbolic constraints over the variables.

The goal of solving a CSP is to find a valuation: $V \rightarrow D$, which maps a variable to a value such that all the constraints in C are satisfied. Typically, the interest is to find whether a solution exists, and if it does then find one or all solutions of the problem. In order to formulate a problem P as a CSP model, a set of variables and domains are selected to represent the entities in P and then constraints are written in terms of the variables to represent the restrictions on solution in P.

Solving a CSP involves assigning values to each of the decision variables. A *constraint solver* is a software package that takes in a CSP and determines an assignment of values to the decision variables. A typical finite domain constraint solver uses a two step process to solve a CSP: (1) Propagation and (2) Search [102].

Propagation is a means to infer implications of a constraint on the variables in its scope and prune out inconsistent values from the variable domains. For example given a simple CSP problem,

- variables V = a, b

- domains D = (0..4),(1..5)
- constraints C = a $\geq$ b

A constraint $a \leq b$ can infer that the value '5' can be removed from the domain of b and value '0' can be removed from the domain of a as these values do not satisfy the constraint. A set of consistency algorithms are used depending on the number of variables in a constraint. Node consistency algorithms prune inconsistent values from the domain of a single variable using unary constraints. Similarly, arc consistency algorithms prune out inconsistent values from domains of two variables using binary constraints. K-consistency and Path consistency algorithms are used to remove inconsistent values till no more values can be removed from the variable domains.

However, consistency techniques alone may not be able to narrow down the domain to a single value. In general consistency algorithms are combined with systematic search to derive complete solutions to a problem. A search is a systematic way of traversing the space of complete assignments to find a solution to the CSP. A search tree is constructed based on variable and value ordering to systematically assign values to variables. The *variable ordering* specifies the order of selecting the decision variables while creating the search tree and the *value ordering* specifies the order of selecting a value in the variable domains. These two heuristics form the core features of the search. At each step of the search, a value is assigned to a variable. This triggers the propagation, where consistency checks are performed till no more values can be removed. This step is iteratively performed till a solution is obtained.

Several commercial and free constraint solvers exist today [2, 11, 30, 122]. Each constraint solver supports a declarative constraint programming language to specify the CSP. This language includes constraint patterns that are often found in combinatorial problems, called *global constraints*. The constraint solvers support specialized consistency algorithms for propagation of global constraints, thereby reducing the search time.

II.1.2 Boolean Satisfiability

Boolean Satisfiability (SAT) is the problem of determining if there exists an assignment of values to boolean variables in a propositional logic formula such that it evaluates to true. In the past, SAT has been used to solve planning problems [65] and FPGA layout problems [87]. Solving a problem using boolean satisfiability involves two steps: (1) *modeling*, and (2) *solving*.

Modeling involves formulating the problem as a propositional logic formula. A propositional logic formula consists of a set of boolean decision variables, boolean literals and a set of logical connective like $\wedge$, $\vee$, $\implies$, $\neg$ to form boolean expressions. The propositional logic formulas should be in *Conjunctive Normal Form (CNF)*, which is a standard format accepted by most SAT solvers. A CNF consists of a conjunction of clauses, where each clause is a disjunction of literals and each literal in turn is a boolean variable or a negation of it. The goal of solving a SAT problem is to determine the existence of an assignment such that all clauses are satisfied. Two variants of the SAT problem are MAX-SAT and weighted MAX-SAT, where the goal is to find an assignment that maximizes the number, or weighted sum of satisfied clauses.

Solving a SAT problem involves finding a single assignment for decision variables that makes the formula true. Existing SAT solvers use Davis-Putnam-Logemann-Loveland (DPLL) [32] algorithm or a variant of it to solve SAT problems. Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a systematic backtracking search algorithm, which starts by choosing a literal in the formula, assigning a truth value to it, simplifying the formula and then recursively checking the simplified formula. The basic DPLL algorithm suffers from the same drawbacks as the backtracking algorithm (late detection of conflicts, redundant work due to lack of record of conflicting values of variables). The improved variants of DPLL algorithm incorporate clause learning and non-chronological backtracking. For example, Chaff [86] is a conflict-driven SAT solver which integrates the improvements over the standard DPLL algorithm, and as a result, performs very well on practical SAT benchmarks.

Another approach for solving SAT problems uses approximate algorithms to find solutions. Approximate algorithms are incomplete algorithms that are used to obtain high quality (near optimal) solutions to the combinatorial optimization problems in polynomial time. These algorithms do not guarantee finding a globally optimal solution. In order to solve a SAT problem using approximate algorithm, the problem is formulated as an optimization problem, where the goal is find a solution with maximum number of satisfied clauses. A local search algorithm (Section II.2.3), is an approximate algorithm that starts with a randomly generated truth assignment and then selects variables according to some heuristics for bit flip. Depending on the heuristic used to choose the variable, different local search algorithms exist for SAT. One of the first local search algorithms used for solving SAT problems was GSAT [111]. An improved version of the basic local search algorithm for SAT is WalkSAT [52] where some noise is added to the strategy. A comprehensive survey of the different local search algorithms for SAT problems is presented in [55]. and second class of SAT solvers use variants of local search algorithms, for example GSAT [111].

II.1.3 Symbolic Constraint Satisfaction

Constraint Programming solves satisfaction problems by searching through the search space to find assignments that satisfy constraints. In order to retrieve all solutions, CP solving techniques will require time exponential in the number of decision variables. Neema [89] came up with an Ordered Binary Decision Diagram (OBDD)[23] based technique to retrieve all satisfying solutions in a single step by pruning out all inconsistent assignments. OBDDs represent boolean functions as directed acyclic graphs and use graph algorithms to perform operations on these boolean functions [24]. Since binary values and boolean operations can be used to implement a variety of mathematical domains like sets, and finite domains, OBDDs and symbolic boolean manipulation can be used to solve set manipulation problems as well. Neema formulated the design space pruning problem as a set manipulation problem, where the initial design space set consists of all possible (feasible and infeasible) design points. This set DS is encoded as a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$, where $n = \lceil log_2 |DS| \rceil$. Similarly the constraints are also represented as boolean functions and thereafter as OBDDs. The main set manipulation operation (intersection) is performed by conjunction of design space OBDDs with the constraint OBDDs. This can be interactively performed by the designer.

This approach suffers from scalability issues in the presence of numerical constraints because the underlying formalism is boolean algebra and booleanization of the numerical domains leads to large OBDDs that take prohibitively long time to prune the trees. Eames [34] developed a hybrid technique based on OBDD-based and finite domain CP to perform sequential pruning by first applying boolean constraints using OBDD based technique and then using CP to apply numerical constraints.

The BDD based technique was initially developed for synthesis of adaptive computing systems, but they have also been used for design of embedded systems [14], reconfiguration of sensor networks [69],and architecture synthesis [106].

II.1.4 Summary

The focus of this class of methods is to retrieve all valid solutions of a given problem. Broadly, methods for solving constraint satisfaction problems can be divided into two categories: (1) Propositional logic based techniques and (2) Constraint programming [102].

A propositional logic formula consists of a set of boolean variables and a set of logical connective like $\wedge$, $\vee$, $\implies$, $\neg$ to form boolean expressions. A propositional logic techniques takes a propositional logic formula as input and determines whether there is a variable assignment that satisfies the formula. In this section we discussed two propositional logic techniques: Boolean Satisfiability [79] and Symbolic Constraint Satisfaction

[89]. In order to either of these techniques to automate DSE, the variables and input parameters have to be represented as boolean variables, each constraint is mapped in to one or more formulas and the conjunction of these formulas is fed into the solver. However, these technique do not scale well in the presence of integer decision variables with large domains. Encoding integer constraints as propositional formulas can easily lead to combinatorial explosion in the size of the formula [26]. These techniques are good for coarse grained design space exploration.

On the other hand constraint programming based techniques take CSP (Section II.1.1) as input to determine an assignment for the variables in the CSP. Constraint Programming does not impose any restriction on formulation of constraints making it easier to formulate a problem as a CSP. For finite domain constraint programming, both integer and boolean variables can be used. Moreover, most existing solvers based on constraint programming also support modeling and solving optimization problems using the branch and bound algorithm.

II.2 Methods for Mono-Objective Optimization

An optimization problem is a problem of finding an optimal solution from a set a possible solutions. A constraint optimization problem (COP) can be formally defined as a tuple [125]

$$\langle X, f, D, C \rangle \tag{II.2}$$

where,

- X is a set of decision variables $X = \{x_1, ..., x_n\}$;
- D is a set of possible value that each variable can take.
- $f(X)$ is the objective function
- C is a set of constraints on the values of the variables.

This definition in essence is a CSP $\langle X, D, C \rangle$ with an optimization function f. Let S be the set of all possible solutions, then $f : S \Rightarrow \mathbb{Z}$ assigns a value to each solution $s \in S$ indicating its worth. The objective function defines a total order relation between any two solutions in the search space. A solution $s \in S$ of COP is therefore a solution of CSP for which $f(s)$ is minimum or maximum depending on the requirements of the problem.

A *combinatorial* optimization problem is essentially an optimization problem where X consists of discrete decision variables and finite domain search. In this section we focus on the techniques used for solving single objective combinatorial optimization problems.

II.2.1 Branch and Bound

Branch and bound [75] is an exact algorithm that is used to solve optimization problems to retrieve globally optimum solution. The design space is explored by building a tree, where the root of the tree represents the problem and the leaf nodes represent solutions to the problem. Internal nodes of the tree represent subproblems, such that the size of the subproblem reduces as we move from the root to the leaf. The tree is dynamically construct by iteratively *branching* and *pruning*. The branching strategy divides the problem into two mutually exclusive subproblems that can in turn be divided into smaller subproblems. Many branching strategies can be applied such as breadth-first, depth-first and best-first. The pruning strategy is used to prune out subproblems that will not lead to optimal solutions. This is done by computing a lower bound of the subproblems. If the lower bound is greater than the best solution so far then the subproblem can be pruned out.

This method has been used to solve scheduling problems [58, 81]. Branch and bound algorithms have been used to solve mono-objective optimization problems formulated as a CSP or a linear program.

II.2.2 Mathematical Programming

Mathematical programming [109] facilitates the modeling and solution of a broad class of constrained optimization problems. A commonly used model in mathematical programming is the Linear Programming (LP). A linear program is an optimization problem that seeks to minimize a cost function subject to a set of linear constraints. A linear program in its standard form can be formulated as:

$$Minimize : c^T.x \tag{II.3}$$

$$Subject\ to : A.x \leq b \tag{II.4}$$

$$x \geq 0 \tag{II.5}$$

where A is a matrix, and c, b are vectors of known coefficients. A, c and b are given and x represents a vector of decision variable whose value is to be determined. Equation II.1 represents the objective function that combines the different variables to express a goal. The objective function has to be maximized or minimized. Equation II.2 represents the linear numeric constraints that are used to express bounds on possible solutions. A solution of a linear program is binding of each decision variable to a value such that all the domain constraints and bound constraints are satisfied.

An Integer linear programming (ILP) extends the concepts of linear programming by adding integrality constraints to the linear programs. A Mixed Integer Linear Program (MILP) relaxes the integrality constraint of ILP and can have variables which have one or more variables which take real values. MILP has been used modeling hardware/software partitioning [92], network design [28], real-time scheduling [70]

Search Methods: The following search methods have been used for solving linear programming models:

1. *Simplex Method* [88] is the most popular solution technique for solving linear programs developed in 1947. However, Simplex method has an exponential time complexity. Karmakar's projective scaling method [63] is a polynomial time algorithm for solving linear programs.

2. *Branch and bound* [75] is used in combination with linear relaxations to solve ILP's. Using branch and bound an ILP solver decomposes the problem into smaller subproblems recursively. The integrality constraints are replaced with lower and upper bounds on the variables, thus transformation it into an LP problem. The bound obtained by LP relaxation is used to discard subproblems that have a bound than the best known solution. An advanced version of branch and bound is the branch and cut algorithm [96].

3. *Metaheuristics*: Local search algorithms have also been used to solve large instances of ILPs [128]. The local search algorithms are discussed in detail in Section II.2.3.

Linear programming has proved efficient in solving a variety of exploration problems like scheduling [100], hardware-software codesign [93].

II.2.3 Local Search Algorithms

Local search algorithms are one of the most widely and successfully applied approximate algorithms. A template of a local search algorithm is shown below:

Local Search Template

```
LocalSearch( s)
    let currentSolution = s
    While not Termination Criterion Do
      GenerateNeighbourhood(currentSolution)
      If no better neighbour then
        Stop
      Else currentSolution = improvedSolution
    Output Local Optima
```

A local search algorithm starts with a given solution of the problem and iteratively tries to find a better solution in the neighborhood of the current solution. In case a better solution is found it replaces the current solution. This step is iteratively performed till no better solution can be found in the neighborhood. The disadvantage of using the local search algorithm is that it can get stuck at a local minima. In order to overcome this drawback several improvements have been proposed. One approach is to restart local search from a new, randomly selected solution. This approach is applied in Iterated Local Search (ILS) [78], Greedy Randomized Adaptive Search Procedure (GRASP) [40, 41, 101]. Another possible approach is to accept worse solutions, thus escaping the local optima. This approach is applied in Simulated Annealing [74], Tabu Search [46].

II.2.3.1 Simulated Annealing

Kirkpatrick [67] applied the annealing concept from physics to solving combinatorial optimization problems. Annealing is based on the principle of mechanics whereby a substance is heated and then slowly cooled to get a strong crystalline structure. If the substance is not heated to the right temperature or the cooling is too quick then the process, then the resulting solid is weak and brittle. Thus, it is imperative that the cooling is done at the right rate.

Simulated Annealing (SA) algorithm simulates the energy changes in the system till it converges to an equilibrium state. The optimization problem is analogous to the system, where the system state represents a solution and the energy of the system represents the objective function. Starting from an initial solution, SA incorporates significant randomization while traversing the state space. In each iteration a random neighbor is selected. If the objective value of the neighbor is better the move is always accepted. If not, then the move is accepted with a probability P, which is a function of the current temperature (a control parameter) and the difference in the objective value (ΔE). Initially the temperature is high and the probability of accepting non improving solutions is also high, but as the simulated temperature decreases according to the cooling schedule, fewer such moves are accepted. The search stops after stopping condition is met, typically when the probability becomes negligible or the temperature reaches a certain threshold.

Different versions of SA have been used for design space exploration, for example a multi-objective simulated annealing algorithm [97], a parallelized simulated annealing algorithm [50], where a number of instances of the algorithm are run in parallel.

II.2.3.2 Tabu Search

Tabu Search (TS) [46] algorithm was developed by Glover. Like SA, TS also accepts nonimproving solutions to escape from the local optima. In contrast to random moves used in SA, the neighborhood is explored in a deterministic manner in Tabu search. Like the basic local search algorithm, a better neighbor replaces the current solution, but the search continues even after reaching the local optima by accepting non-improving solutions. This can lead to cycles if the solution accepted has been previously traversed. TS avoids cycles by maintaining a list of recently visited solutions/moves (tabu list) and discards all those neighbors present in the list. TS also uses certain conditions, called aspiration criteria to accept tabu moves if it generates a better solution among the set of solutions possessing a given attribute. Advanced techniques like medium-term memory and long-term memories are commonly used to handle intensification and diversification of the search.

Tabu search has also been used for design space exploration of embedded systems [35, 116].

II.2.4 Summary

Mono-objective optimization techniques deal with finding a globally optimal solution with respect to an objective. In general mono-objective optimization problems are easier to solve as compared to multi-objective optimization problems, because of the total ordering of the solutions in the search space. Mono-objective optimization problems have been solved using both exact and approximate algorithms. In this section, we presented both exact techniques: mathematical programming, branch and bound, and approximate techniques: Simulated Annealing and Tabu Search.

In general exact techniques perform better for tightly constrained problems, while approximate algorithms perform better for unconstrained optimization problems. However, approximate techniques are mostly used when exact techniques are unable to provide solution in acceptable amount of time. This is true for large instances of combinatorial optimization problems. The approximate algorithms are generally used where reasonably good solutions obtained in polynomial time are preferred over globally optimum solutions.

II.3 Methods for Multi-Objective Optimization

Unlike mono-objective optimization problem, many existing DSE problems have multiple objectives that conflict with each other, such that no single design solution is optimal with respect to all objectives. Thus we need to find a set of solutions that trade off between the different objectives. These problems are known as multi-objective optimization problems and consist of at least two objectives. The goal of solving multi-objective optimization problems is retrieve a set of Pareto-optimal solutions. A Pareto-optimal solution has a property that a given objective cannot be improved without deteriorating the values of other objectives. A solution is Pareto-optimal if improvement of one objective will make it worse with respect to other objectives. An optimal solution for these design space exploration problems involve search of a set of solutions, referred to as *Pareto Optimal solutions*. Gries [48] presents a definition of dominance and Pareto-optimality:

Definition 1. ***Pareto-criterion of dominance****: Given k objectives to be minimized without loss of generality and two solutions (designs) A and B with values (a_0, a_1,..., a_{k-1}) and (b_0; b_1; ... ; b_{k-1}) for all objectives, respectively, solution A dominates solution B if and only if*

$$\forall_{0<i<k} i : a_i \leq b_i \tag{II.6}$$

and

$$\exists j : a_j < b_j \tag{II.7}$$

This means that a dominating solution is better in at least one objective, while being the same with respect to other objectives.

Definition 2. ***Pareto-optimal solution****: A solution is called Pareto-optimal if it is not dominated by any other solution. Non-dominated solutions form a Pareto-optimal set in which neither of the solutions is dominated by any other solution in the set.*

The solutions in the Pareto-optimal set cannot be ordered totally using the dominance relation presented in Def. 1. All the solutions are equally viable and a further evaluation is required to compare the solutions in the set. A wide variety of DSE problems found in literature [13, 38, 45] require simultaneous optimization of more than one objectives, such that valid solutions should also satisfy a set of constraints.

II.3.1 Evolutionary Algorithms (EA)

Metaheuristics are general purpose algorithms mainly used to obtain solutions to large combinatorial search problems in polynomial time. The metaheuristics model search problems after the evolution of species. They can be applied to any optimization problem. Evolutionary algorithms (EA)[12, 42] are a class of population-based metaheurisitcs, which involve iterative improvement of a population of initial solutions. Figure 1 shows a template of an evolutionary algorithm.

An Evolutionary Algorithm is a population based metaheuristics that is based on the iterative improvement of a *population* of solutions rather than a single solution. Table 1 illustrates a generic template of an EA. With an EA, a set of initial solutions (population $P(0)$) is created and then evolved over a series of generations to reach a final solution. The evolutionary process involves generation of a new population of solutions and replacement of the current population. First the $Evaluate(P(t))$ function associates a fitness value to

Evolutionary Algorithm Template

```
    Generate( P(0) )
    $t$ = 0;
    **While** not Termination Criterion **Do**
      Evaluate($P(t)$)
      $P'(t)$ = Selection($P(t)$)
      $P'(t)$ = Reproduction($P'(t)$); Evaluate($P'(t)$);
      $P(t+1)$ = Replace($P(t)$, $P'(t)$)
      t = t + 1
    **End While**
    **Output** Best individual or population found
```

Table 1: Evolution Algorithm Template [123]

each solution of the population. The $Section(P(t))$ function selects solutions with better fitness value. A simple selection strategy is to randomly select k individuals from the population and then the best solution from these k candidates is selected. These candidate solutions then act as parents to generate new offspring by variation. The $Reproduction(P'(t)$ function, which basically includes the variation operators first makes small changes in the selected candidate solutions and then combines them such that the offspring inherits the characteristics of both parents. The $Replacement$ function selects the best solutions from the old population and offspring to promote their propagation in the evolution. This process iterates till it satisfies a stopping criteria. When the evolutionary process ends the solutions with the highest value is output as the best solution.

II.3.2 Scalar Approaches

Multi-objective optimization problem can be solved using mono-objective optimization methods, such as mathematical programming by transforming multi-objective problems to mono-objective optimization problems. This is done by using an aggregation function to combine the different objective functions f_i into a single objective function:

$$f(x) = \sum_{i=1}^{n} w_i f_i(x), \;\; x \in S$$

combing the objectives where the weights $w_i \in [0...1]$ and $\sum_{i=1}^{n} = 1$

II.3.3 Summary

In this section we discussed two approaches for multi-objective optimization. The scalar approach is used to combine mono-objective optimization techniques to solve multi-objective optimization problems by aggregation of objectives into a single objective (cost). The decision maker should have a good understanding of the problem being solved in order to come up with an appropriate weighted sum of the objectives, so that useful results can be obtained. The quality of solutions obtained using scalar techniques depends on the weights. Moreover, depending on the weighted sum of objective, certain regions of the design space unreachable by the search. In order to overcome this limitation, a search might have to be repeated several times with different weights, such that different regions of the search space can be explored. However, a disadvantage of this technique is that it does not retrieve all Pareto-optimal solutions in the presence of a non-convex Pareto front. Another approach for using mono-objective optimization techniques for solving multi-objective optimization problems is to use multi-runs, each with one objective to retrieve a single optimal solution. However, even this approach does not retrieve all possible Pareto-optimal solutions. Scalar techniques are therefore used for solving multi-objective optimization problems with convex Pareto fronts.

Contrary to scalar techniques, the Evolutionary Algorithms (EA)s (and other population based metaheuristics) can retrieve Pareto-optimal solutions in a single run. Some of the popular algorithms are Nondominated Sorting Genetic Algorithm (NSGA) [33] and Strength Pareto Evolutionary Algorithm (SPEA2) [135]. Moreover, Evolutionary Algorithms also work better for solving large instances of unconstrained optimization problems.

II.4 Design Space Exploration Frameworks

A large body of work exists on representation and solution of DSE problems, especially in embedded systems [94] and software product-line engineering [17]. Numerous domain-specific frameworks exist that use a combination of solvers and techniques described in section. We sample a few domain-specific and generic exploration frameworks and compare the capabilities in an attempt to highlight the limitations that exist and list out requirements for a generic framework.

II.4.1 Domain-specific DSE Frameworks

Design space exploration techniques have been widely studied in the areas of electronic design automation and high level synthesis. Gries [48] presents a survey of some of these techniques and frameworks used for early design space exploration in embedded system. Most commonly, DSE frameworks in embedded systems are used for component selection, resource allocation, routing and scheduling.

II.4.1.1 Metro-II

Metro-II [31] is a DSE framework developed at UC Berkeley. It is an integrated design environment for platform based design of embedded systems. The framework supports system level design starting from functional specification to a mapped and optimized implementation. Metro-II was developed to overcome the limitations of Metropolis [15] with the goal of automating mapping of parallel heterogeneous embedded systems. In Metro-II, modeling starts with a separate specification of application and platform. This is followed by a mapping phase where the exploration is performed based on the constraints and objectives. Mapping is the process of associating the functional blocks with architectural models such that the services used by the functionality are bound to those provided by the architecture. The problem is formulated as a covering problem, where each process is covered by exactly one architectural resource. The mapping should satisfy the performance constraints and optimize the relevant metrics. An MILP formulation is used in the back-end to automate the process and a constraint language name Logic of Constraints (LOC) is used to specify behavioral and structural constraints to be satisfied by the solution.

Metro-II is a domain-specific DSE framework for embedded systems synthesis that translates the DSE problem model to an MILP mathematical model.

II.4.1.2 MILAN

Model based **I**ntegrated sim**L**u**A**tio**N** (MILAN) [14] is a framework developed by researchers at USC and Vanderbilt/ISIS. It is mainly used for used for system level DSE for heterogeneous embedded systems. The goal is to explore the space of possible design alternatives created as a result of several architectural parameters such as voltage, operating frequency, configuration, etc. and find a design solution that is optimal with respect to throughput and energy consumption. The framework uses multi-layered design space exploration starting with symbolic constraint satisfaction (described in Section II.1.3). This is used to prune the design space in order to remove infeasible design points. The design points obtained after pruning are iteratively evaluated using HiPerE, a performance estimation tool which uses trace-based simulation to calculate estimate time and energy performance of the system. The design points with lower performance are discarded and the promising solutions are them simulated using low-level simulators.

This framework is more scalable because instead of exploring a large design space at once, it performs multi-step exploration. The main drawback is that the lower-levels of explorations are tightly coupled with estimation tools.

II.4.1.3 MULTICUBE

MULTICUBE Explorer [113] is a multi-objective design space exploration framework to optimize MP-SoC architectures. This framework was developed as a part of the MULTICUBE project [4] and was aimed for solving design time problems. The framework uses a parameterized representation of the SoC architecture and explores the design space created by a combination of architectural parameters. The goal is to find the best tradeoff in terms of different objectives (energy, area, etc).

MULTICUBE is different from other frameworks because it is more flexible. It supports a common interface for simulation engines and optimization algorithm, such that the designer can plug in his optimization algorithm. The framework provides a set of meta-heuristics algorithms to compute an approximated Pareto-optimal set. The exploration process is made even more efficient by using neural networks to predict the cost of a design point instead of performing the actual simulation.

II.4.1.4 FeAture Modeling Analyzer (FAMA)

FAMA is a framework for the automated analysis of feature models. It integrates some of the most commonly used logic representations and solvers for the analysis. The automated analysis is a two-step process:

1. The feature model is translated to a logic (such as propositional logic) representation.
2. Off the shelf solvers are used to extract information from the logic representation.

At present CSP, SAT and OBDD solvers are integrated in the back-end and the framework has been used to compare the efficiency of these techniques in performing a predefined set of operations on the feature models.

This framework provides more solver flexibility than other frameworks found in literature survey. The solver flexibility comes from the support for multiple solvers in the back end. This allows the modeler to choose a different solver for every instance model. This flexibility is not present in most other frameworks where only one search technique is supported in the back end. The modeler also has an option to use to automatic selection, which allows the framework to automatically select the solver to use according the operation requested.

II.4.1.5 Software Product Lines Online Tools (S.P.L.O.T)

SPLOT is a Web-based system which uses SAT and OBDD solvers to analyze and configure software product lines. The tool provides interactive configuration services to users. This framework, like FAMA uses different solvers to perform different operations like counting of valid configurations is done by OBDD solver, whereas selection of a particular configuration is done by invoking a SAT solver.

SPLOT supports more than one solver in the back-end but does not give the modeler the flexibility to choose the solver. The choice of the solver used for a particular operation is hard-coded in the execution engine. For example, operations like auto-completion, and configuration are performed only by SAT solvers, whereas product search depending on feature selection by the modeler are solved using OBDD based solvers.

II.4.2 Generic Frameworks

II.4.2.1 DESERT

Design Space Exploration Tool (DESERT) [90] 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 which boolean constraints describing interaction of design choices. The design tree and constraints are symbolically encoded, using OBDDs. (for more details refer to Section II.1.3). The designer can interactively choose the hard constraints and prune the design space. The main advantage of this approach is that it is exhaustive and is very useful for performing coarse grained design space exploration. However, it does not scale well in the presence of continuous finite domain variables [89].

DESERT-FD [34] is a DSE framework developed to overcome the limitations of DESERT. It uses a combination of OBDD and a Finite-Domain (FD) solver to perform DSE. Both DESERT and DESERT-FD frameworks are domain-independent frameworks and can be used to represent and explore design spaces in any domain.

II.4.2.2 PISA

A **P**latform and Programming Language **I**ndependent Interface for **S**earch **A**lgorithms (PISA) [21], is an interface specification and tool developed at E.T.H, Zurich. The focus of the tool is to perform multi-objective optimization. The interface provides a lot of solver-flexibility because it allows separating the problem-specific and problem independent part of an optimization problem. The search algorithm is a separate process that communicates with the optimization problem. The problem representation part can be arbitrarily combined with any ready-to-use multi-objective optimization algorithms. Although the domain-engineer does not need to implement the search algorithms himself, he still needs to write the domain-dependent evaluation and consistency checking functions.

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. It is language and platform independent, but a significant coding effort is required to write the evaluation functions.

II.4.3 Discussion

Table 2 shows a summary of the frameworks surveyed in this chapter. The frameworks are compared on the basis of the modeling and solution support provided by them. The modeling support includes the the design space representation and constraint specification language. The solving support highlights the search method used by the framework.

The frameworks used for product-line configuration, in general support an enumerative design space representation, where each alternative is explicitly modeled. The design alternatives are organized in a compact representation, called the feature model. DESERT also uses an enumerative representation to organize alternatives hierarchically. On the other hand, the frameworks used for automated DSE in embedded systems, support partial model representation of the design space, where an incomplete model represents the design space. Each alternative in the space is a model obtained by completing the partial model.

All, except two frameworks surveyed are domain-specific. This means that the same framework cannot be used to model and solve exploration problems from other domains. This is also because the design space is represented using domain-specific concepts. Most frameworks (except PISA) provide a constraint specification languages to specify declarative constraints. The Metro-II framework uses logic of constraints, a language used for specifying temporal constraints. The constraint languages supported by the frameworks range from high-level languages like OCL to low-level languages like propositional logic. In contrast to the low-level languages that are tied to the search techniques, high-level languages provide a user friendly solver-independent syntax to specify validity conditions. However, most frameworks surveyed in this chapter do not provide any support to the modeler for constraint formulation.

Most frameworks (except FAMA) are built on a preselected solver, such that all problem instances modeled using the framework are solved by the same solver. This is also because different frameworks are built on the requirements of the class of DSE problems it is intended to solve. For example, some frameworks are used to obtain design points which satisfy constraints (DESERT), yet there are others which are used for multi-objective optimization. However, pre-selection of solver prevents the modeler from solving both satisfaction and optimization problem in the same framework.

		Frameworks						
Property		FAMA	SPLOT	METRO-II	MILAN	MULTICUBE	DESERT	PISA
Model	Design Space	Feature Model	Feature Model	Mapping Model	Mapping Model	Mapping Model	Enumerative	bit string
	Notation	graphical	graphical	graphical	graphical	graphical	graphical	textual
	Constraint Language	Propositional	Propositional	LOC	OCL	linear expr	OCL	C(code)
	Domain	product-line	product-line	embedded	embedded	embedded	generic	generic
Solve	Technique	Exact	Exact	Exact	Exact	Approx.	Exact	Approx.
	Kind	SAT, CSP, OBDD	SAT, OBDD	ILP	OBDD	GA	OBDD	GA
	Flexibile	yes	no	no	no	yes	no	yes
	Problem Size	small-large	large	medium	large	large	large	large
	Solutions	feasible	feasible	optimal	optimal	good	feasible	good
	Objectives	satisfy	satisfy	multiple	single	multiple	satisfy	multiple

Table 2: Summary of Frameworks

CHAPTER III

A GENERIC FRAMEWORK FOR DESIGN SPACE EXPLORATION

This chapter presents a generic framework for automated design space exploration. This work is motivated by the limitation of existing approaches for automated DSE, which focus on solving problem instances belonging to a *single* class of DSE problems from a given domain. As an example consider the problem of assigning of real-time tasks to multi-processor during embedded systems synthesis. This is a common DSE problem, where given a set of task and a set of resources, the goal is to find an assignment of tasks to processors, such that the solutions satisfy certain non-functional properties (cost, memory, etc). A large body of work exists to automate the exploration of design space for the task to processor mapping problem. Approaches range from use of ad-hoc algorithms [66, 95] to using frameworks [31, 34, 90, 132] built on integration of constraint solvers etc into domain-specific modeling environments.

The use of frameworks to automate DSE is better than other approaches found in literature as they support reuse of code and design. The framework-oriented approach is built on a model compiler that transforms an arbitrary model in the domain-specific modeling language along with a set of constraints to a suitable formulation, for example an integer linear program that can be processed by an optimization solver to retrieve valid and perhaps optimal solutions. The results of the solver are integrated back into the domain-specific environment. This approach allows the domain-engineer to model DSE problems at a higher abstraction level using familiar notations, and then solve them without any particular knowledge of the search method.

However, like other approaches, existing DSE frameworks are also tailored to solve a class of domain-specific DSE problems. This is due to the tight coupling of the model compiler and the modeling language with the domain-specific aspects of the problems being solved. Literature survey reveals that similar kind of DSE problems exist in different domains. For example the design of synchronous optical networks [117] is a DSE problem, where a set of client nodes have to be installed on one or more rings given the traffic and demand constraints. This problem, like the task to processor mapping problem in embedded systems, is also a resource allocation problem. Although the general-purpose solvers are powerful enough to solve both problems, the domain-specificity of the existing frameworks restricts their use to modeling and solving only one of the two problems. Given the time and effort required to develop frameworks for automated DSE and the power of general purpose solvers, it would be advantageous to have a unified framework for DSE that can be exploited by as many users as possible.

This chapter introduces a *generic framework* for automated DSE that can be used to *model* and *solve* DSE problem from different domains. The framework adopts Model Integrated Computing (MIC)[121] as its core technology. MIC paradigm is a development methodology that centers around the use of models in system development activities. MIC facilitates model-based development by definition and implementation of a *domain-specific modeling language* (DSML), a language tailored to a particular domain. Use of models raises the level of abstraction and enables the modeler to reason about the problem in terms abstract concepts. Moreover, use of DSMLs in particular improves the modeling experience through the use of domain-specific concepts. This is a much desired feature for the generic DSE framework, where use of a DSML enables the domain-engineer to formulate the DSE problem using domain-specific concepts.

The generic framework is based on three distinctive features as compared to existing framework: (1) a set of reusable classes that capture concepts common to DSE problems across domains, (2) a simple constraint language and constraint templates that enable easy constraint specification, and (3) a solver-independent abstraction level that enables translation of a well-formed problem model to a number of solver-specific formats that can be processed by the different solvers. The key idea is whenever a new class of DSE problems in a given domain is to be solved, the domain-expert associates the DSE aspects captured by the reusable classes with the existing domain-specific concepts in the DSML of the domain to create a new DSML. This new language can be used to specify models of the problem using domain-specific notation. The DSML is used to configure the the generic framework to create a domain-specific DSE environment, that can then be used by domain-engineers to model DSE problem instances. The model compiler is domain independent, therefore, any problem modeled in the framework is correctly transformed to a solver-dependent format.

In order to achieve generic modeling of DSE problems, an alternative approach would have been to use the Unified Modeling Language (UML) [103], for modeling the DSE problems. The UML provides a single unified language that can be used to describe all domains. A common approach to use the UML is to tailor the existing models using profiles. A UML profile is an extension of the UML language for an area of application, for example the SysML [43] is a UML profile for System Modeling. A UML profile for specifying DSE problems would be generic enough to support DSE problems from different domains. However, this approach requires the domain-engineers to learn a new language, in this case the UML profile, that lacks the familiarity of a domain-specific modeling language. This in turn will add to the complexity of modeling the DSE problems. Therefore, in order to achieve generic modeling support without losing out on the niceties of using domain-specific modeling languages, the modeling support of the framework is based on a clear conceptual separation of the domain-specific concepts and the DSE aspects of the problem. This separation allows the use of the framework to model any DSE problem.

In order to solve the problems modeled using the framework, a generic solver support is included in the framework. Existing approaches for automated DSE integrate one general-purpose solver that is most suited for their needs. This choice of the solver is based on the objective of solving the problem (global optimal solution, all valid solution) as well as the type of constraints (linear, non-linear). However, in a generic framework that supports solution of DSE problem instances with arbitrary combinations of objective and constraints, integrating a single solver will not suffice. Thus, in order to have an extensible solver support, a solver-independent abstraction level was introduced in the framework. Any problem modeled in the framework is automatically translated to a model at this abstraction level. This model of the problem can then be translated to different solver specific formats and solved according to the objectives and constraints. At present, two solvers, one based on constraint programming [16] and one based on evolution algorithms [47] have been integrated. Although these two solvers are capable of solving a wide range of problems, any more solvers can easily be integrated into the framework given the simplicity of the solver-independent abstraction level. Unlike existing approaches, where the choice of the solver is predefined, the domain-engineer can select a solver based on the requirements of the instance.

In this chapter we give and overview of the architecture and discuss the languages involved, in detail. We use the following definition while describing the concepts in the framework.

Definition 3. ***Conceptual Model***: *A conceptual model MM_{CM} models a class of DSE problems P in a domain. MM_{CM} includes the concepts, relationships, validity constraints that the solution should satisfy and objective of solving P.*

Definition 4. ***Problem Model***: *A problem model M_{CM} models a problem instance P_i belonging to P. A problem model M_{CM} conforms to the abstract syntax captured by the conceptual model MM_{CM} of P.*

Essentially, the concepts in the conceptual model are common to all problem models belonging to the class of DSE problems under consideration.

III.1 Overview of the Architecture

As previously mentioned, the generic framework adopts MIC, that facilitates model-based development by systematic definition and implementation of DSMLs and a suite of meta-programmable tools including (Generic Modeling Environment (GME) [76], Universal Data Model (UDM), Graph Rewriting and Transformation(GReAT)) to reduce the effort required in creating domain-specific modeling environments, as well model transformation and synthesis.

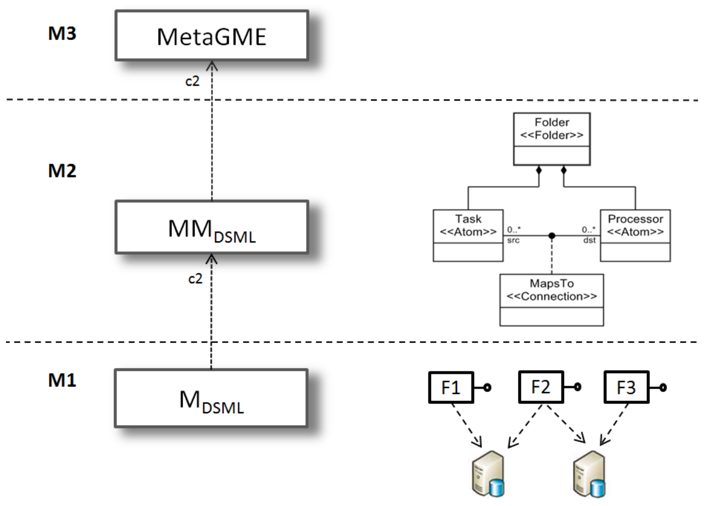


Figure 2: Modeling layers in MIC

A key tool in the MIC metaprogrammable tool suite is the GME [121], a configurable toolkit for creating domain-specific modeling environments. The configuration of the tool is accomplished through *metamodels*, a definition of the syntactic and semantic aspects of the domain in a stereotyped UML-style class diagram. Figure 2 shows the metamodel MM_{DSML} of a task to processor mapping problem. This metamodel is used to automatically generate the target domain-specific environment. The generated domain-specific environment is then used to build domain models M_{DSML} like the one shown in Figure 2, that conform to the language defined by MM_{DSML}. The GME is meta-programmable, therefore, the same environment is used by the language designer to capture the metamodel of a language and the engineers to create model instances of the language. The model can essentially be seen as an instance of the metamodel. The GME provides a predefined metamodeling language, MetaGME that is used to specify metamodels.

The generic DSE framework is based on the meta-programmable tool GME. The reconfigurability of GME enables the domain-experts to configure the generic DSE framework for every class of DSE problem. The framework includes three key elements. First key element of the framework is the ***Abstract Design Space Exploration Language*** (ADSEL), that consists of abstract modeling concepts common to all DSE problems. These concepts might include objectives, constraints, global variables, components. Another key element is the ***Constraint Specification Language***(CSL), an expressive language that is used to write constraint definitions denoting the conditions that the solution model should satisfy. The ADSEL and CSL together form the crux of the generic modeling support in the framework. The third key element of the framework is the ***Intermediate Language*** (IRL), a solver-independent language used to capture a high-level mathematical model of the class of DSE problem under consideration. This mathematical model is then refined to lower-level solver specific formats.

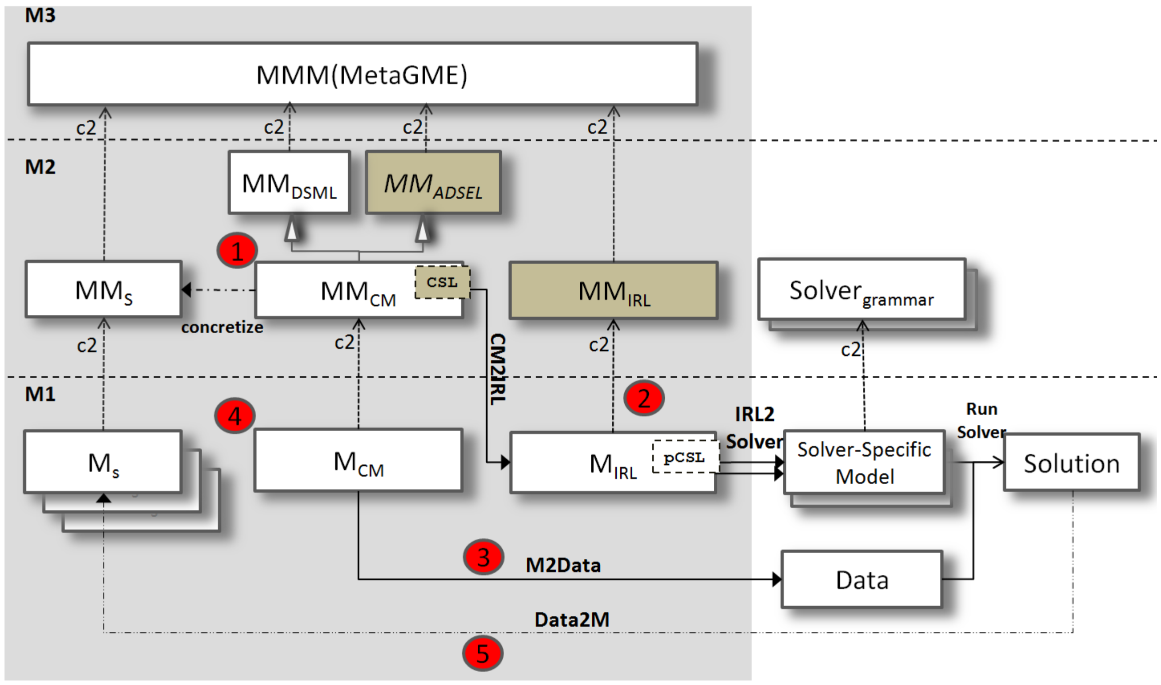


Figure 3: Architecture of the generic framework

Figure 3 shows a detailed view of the architecture of the framework. Given a DSE problem model M_{CM}, the goal of using the framework is to retrieve a set of design solution models M_S. The process of modeling and solving a DSE problem consists of 5 main tasks: (1) Configuration, which involves configuring the framework for a DSE problem in a domain by creating the conceptual model MM_{CM}, of the DSE problem; (2) Search Problem Generation, which involves translation of the conceptual model MM_{CM} to create solver-specific models that can be processed by the solvers; (3) Search data generation, which involves generation of instance specific data from the a DSE problem model M_{CM}; (4) Model and search, which involves creation of a DSE problem model M_{CM} and performing the search; and (5) Solution Model, which involves taking the results of the solver to create a solution model M_S. Task (1) is performed by the domain expert, once for a DSE problem, while task (4) is performed by the domain-engineer for every problem model M_{CM} of the DSE problem. Task (2) and (3) are automated in the framework and task (5) is performed manually using simple steps.

1) *Configuration*: In order to use the framework to solve a class of DSE problems in a domain, the domain-expert has to configure the environment to create a domain-specific DSE environment that supports modeling problem instances belonging to the class. For example, to solve task to processor mapping problems in the embedded system domain, the domain-expert configures the framework to create an environment where the domain-engineer can create a specific DSE problem model consisting of a list of tasks and processors and solve it to retrieve possible mappings. In order to achieve this, the domain-expert starts with definition of DSML for the embedded systems domain, shown as MM_{DSML} in Figure 3, that captures all the key concepts of the domain, for example task, processor, etc. Based on MM_{DSML}, the domain expert creates a new DSML that essentially captures the conceptual model of the task to mapping problem by associating DSE aspects from the ADSEL classes with the existing concepts in MM_{DSML}. This association is achieved by using *Template Instantiation* technique of metamodel composition [36]. As the ADSEL consists of only abstract classes, the template instantiation method proposes inheriting a pre-existing concept *Class m* in MM_{DSML} from an abstract ADSEL *Class a*, such that *Class m* also inherits the characteristics of *Class a* in the resulting new DSML, captured by the metamodel MM_{CM}. For example, the MM_{CM} for the task to processor mapping problem includes *Task* as an *ADSEL component*. The ADSEL classes and their use is discussed in detail in Section III.3. The constraint classes in the MM_{CM} are annotated with constraint definitions specified using the Constraint Specification Language (CSL) constraints that should be satisfied by valid solution models M_S. Once the development of MM_{CM} is complete, it can be used to automatically configure GME, to create a domain-specific DSE environment. The configuration is performed once for every class of DSE problem solved.

2) *Search problem generation*: This task involves translation of the conceptual model to a set of solver-specific formats. This translation is performed in two stages. The first stage, illustrated by the *CM2IRL* transformation (Figure 3), translates the conceptual model of the DSE problem to a high-level mathematical model M_{IRL} conforming to the Intermediate Language (IRL). The second stage transforms M_{IRL} to solver-specific formats. The *CM2IRL* is a domain independent model transformation, that is it remains the same for all MM_{CM}. The transformation rules only focus on those elements of MM_{CM} that are a specialization of the *ADSEL* classes. The *CM2IRL* also rewrites the CSL constraint expressions in the conceptual model MM_{CM} to lower-level constraint expressions in M_{IRL}. These processed CSL constraints, referred to as *pCSL* constraints, are written in terms of elements of M_{IRL}.

In order to translate the model M_{IRL} to lower-level solver specific models a set of translators are used. Currently, the framework supports two solver-specific formats for solving the DSE problem. Figure 4 shows the translators and the models produced. The translator *IRL2CPL* is used to generate a model M_{CPL}, referred to as the ***CP Model***, which is basically a constraint satisfaction problem in a Constraint Programming Language (CPL). For this work, we chose Minizinc [91], a solver-independent medium-level CPL that is used

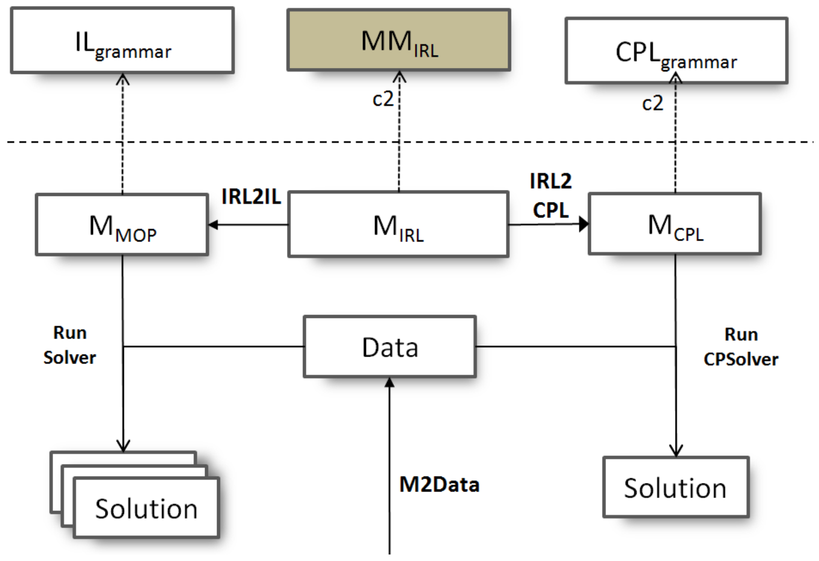


Figure 4: Generation of solver-specific models

to express combinatorial search problems. The M_{CPL} along with the data can be used to retrieve valid design solutions or a globally optimal solution with respect to a single objective. Alternatively, the model M_{IRL} can also be refined a model M_{IL}, referred to as the ***MOP Model***. This model is basically a program implemented in an imperative language that is used in conjunction with an evolutionary algorithm to produce a set of Pareto-optimal design alternatives (Section II.3). Currently, the model M_{IL} is implemented by a Java program and interacts with an evolutionary algorithm program using text-based interface. The MOP model is described in detail in Section III.7.

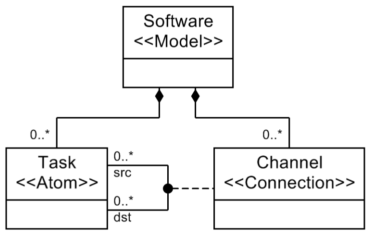
3) ***Search data generation*** is automated by the *M2Data* transformation. It takes the problem model M_{CM} created by the domain-engineer and generates textual data that is used in conjunction with the solver-specific models: M_{CPL} and M_{IL} to generate solution model M_S (or a set of solution models). This transformation is generic and uses a definition of MM_{CM} to generate the right data. The data is generated in two similar each corresponding to the two solver-specific models.

4) ***Model and Search*** is iteratively performed by the domain-engineer when the system is being designed. The domain-engineer creates a DSE problem model MM_{CM} and selects the search method to be used. The solver-specific model corresponding to the selected search method is then executed and solutions (if any exist) are returned in the solver output format, usually a plain text format.

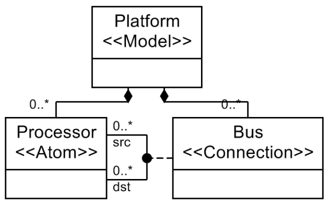
5) ***Solution modeling*** This output is used to create solution models M_S that conforms to metamodel MM_S. The metamodel MM_{CM} is used to create *design spaces* and simple modifications are required so that the metamodel can be used to represent a simple design alternative instead of a design space. These modifications depend on the kind of design space representation used in MM_{CM}. A simple modification can be to concretize an abstract element in MM_{CM} to generate MM_S. The solution to model transformation is performed manually as present.

III.2 Running Example

In the following sections we discuss each of the languages used in the framework in detail. As a running example, we used the task to processor mapping DSE problem found in embedded system [61]. Given an application consisting of a set of tasks and a platform of connected processing elements, the goal of the platform mapping problem is to find a mapping of the tasks on to the processing elements.



(a) Application



(b) Platform

Figure 5: Embedded Systems DSML

Application Specification : Let T be the set of all tasks in the application, i.e., $T = \{t_1, t_2, ...\}$.

Platform: Let P be a set of all processing elements in the platform, i.e, $P = \{pe_1, pe_2, ..., pe_m\}$. Each processing element pe_j has 2 attributes $\{cost_j, powd_j\}$ where $cost_j$ gives the cost and $powd_j$ gives the power dissipation of processing element.

Constraints: All resulting design solutions must satisfy the following constraints

(i) Each task must be mapped to exactly one processor.

(ii) A task $t1$ must not be co-located with $t2$, modeled using a a $NoConflict$ constraint.

(iii) A task must be mapped to one processing element from a set of possible choices, modeled using a $Mapping$ constraint, for example $IDCT$ module can be mapped to a processing elements from a set $\{muP1, muP2\}$.

Problem Objective: For a given application and a processing element library, the goal is to find design solutions that satisfy the constraints, and are Pareto-Optimal with respect to cost and power dissipation. Both CP and MOP models for the running example are tested in this chapter. Figure 5 shows the DSML of an embedded system that embodies domain-specific aspects of the DSE problem.

III.3 The Abstract Design Space Exploration Language (ADSEL)

The generic modeling support of the framework is based on a set of reusable classes that capture the common aspects of DSE problems, for example the components, objectives, constraints etc. These classes together are referred as the *Abstract Design Space Exploration Language* (ADSEL). In this section we discuss the rationale of creating the classes and the definition of the language in detail.

III.3.1 Rationale

One of the key research challenges of our work was to determine a set of abstract modeling concepts that can capture generic aspects of DSE problem. These concepts should be generic enough to be applicable to a wide range of domains. We evaluated a set of DSE problem from different domains to identify common modeling concepts used to capture the problems. The goal is to separate out these common concepts into a pattern metamodel that can be reused for modeling DSE problems. We consider two DSE problems: a task to processor mapping problem [59], and a configuration of software product-lines [131] in our evaluation. In each case we followed 3 steps: (a) Identify key elements of the DSE problem, this includes the design space, the solution of the problem, the constraints that each valid solution should satisfy and finally the objective of the exploration; (b) create a metamodel to reflect the concepts; (c) create a problem model and a solution model to ensure all concepts were captured. We discuss each of the examples in detail. The language definition is specified as a metamodel in GME and uses the GME metamodeling syntax which is well documented in [64]. Concepts like inheritance and containment are similar to those in UML. The stereotypes (ex: «`Model`», «`Atom`», «`Connection`») express the binding of the abstract syntax to the concrete syntax implemented by the GME environment.

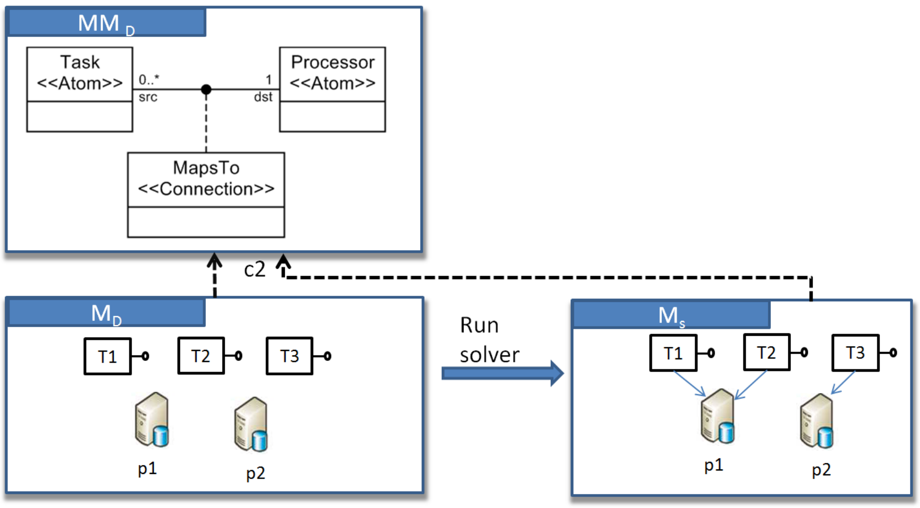


Figure 6: Task to Processor Mapping Problem

1) ***Task to Processor Mapping Problem:*** This is a simplified version of the running example presented in Section III.2.The goal of this version of the task to processor mapping problem is to find a mapping between $MapsTo : T \rightarrow P$, given a set of tasks T and a set of processors P. Key aspects of the problem are: (a) design space: a set of all possible mappings; (b) constraint: each task $t \in T$ is mapped to exactly one processor $p \in P$; (c) objective: find valid mapping. Figure 6 shows a possible metamodel MM_D for capturing the key aspects of the problem. This includes classes: *Task* and *Process* to capture the inputs, and an association *MapsTo* to model the mapping relation. The cardinality of the association captures the constraint. This metamodel can be used to model both the design space model M_D and solution model M_S.

2) ***Software Product-line Configuration*** is a DSE problem found in software product line engineering, where given a set of possible features that can be included in a product, the goal is to select a product variant that contains a subset of features. Key features of this class of problem are: (a) design space: all possible subsets of the set of features. (b) constraints: consistency constraints, like feature A is selected if feature B is selected, composition constraints, (c) objectives: find a product variant with the lowest total cost.

There are two alternate ways to capturing the design space in this problem. One approach is to have a *Feature* class and then use associations to model the constraints on selection on selection of the feature. Another, more concise representation is to hierarchically organize the alternative subsets in a tree like structure, called feature models [62], where the composition constraints are transformed to containment relationships to create a tree-like structure. We use the feature model approach to develop a metamodel to capture the problem, shown in Figure 7. It consists of a *Primitive* and *Compound* classes to model the leaf and internal nodes of the feature model tree. The *Feature* class has two attributes: `select` boolean attribute to model selection of class in a product variant and `cost` attribute. Besides the composition constraints, this problem includes two kinds of constraints: cross tree constraints that relate selection properties of two features,

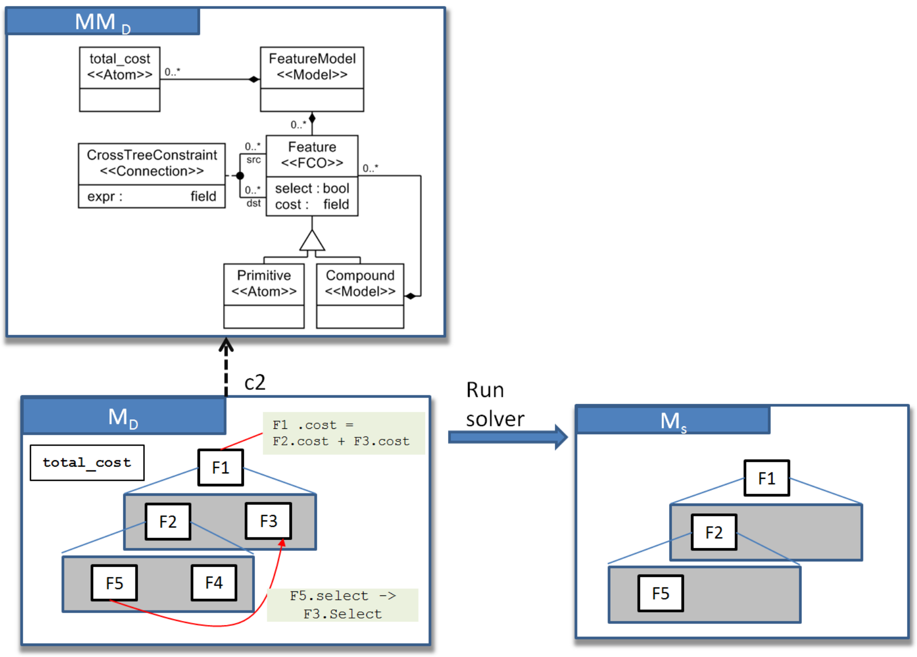


Figure 7: Feature Configuration DSE Problem

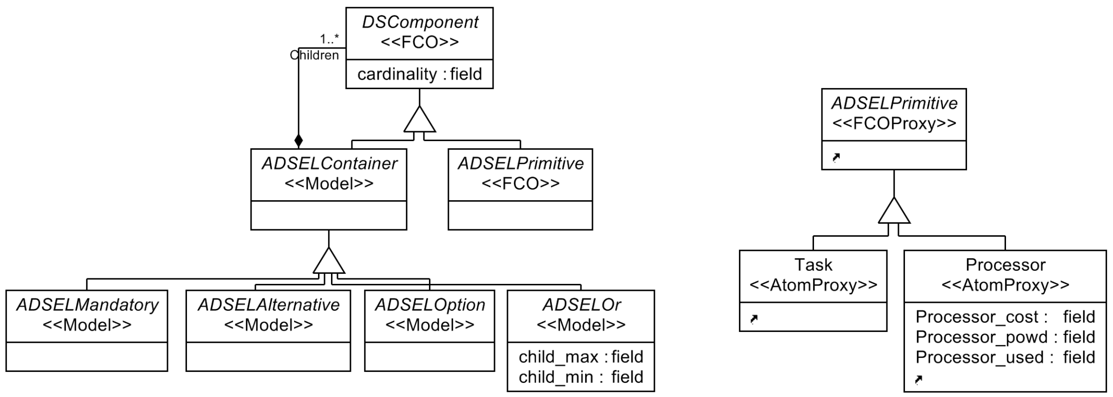
for example '`F1.select->F2.select`', as shown in the figure. This constraint can be specified as an association, where the exact constraint definition is captured as expression attribute of this association. Another constraint is the dependency constraint where the cost of a feature is determined based on the cost of the composed features. For example '`F1.cost=F2.cost+F3.cost`'. Such a dependency relation is more naturally as an assignment statement. Besides this, a concept is required to capture the attributes of a design alternative in the design space. For example, `total cost` of a product variant. Figure 7 shows an example design space model with the constraints and a solution derived based on the information included in the model.

As a result of the evaluation we conducted, we came to the conclusion that the following concepts are required to model the DSE problems in an MDE-based framework: (i) *Components* that form the core of design space representation in the problem models, for example *Task* and *Processor* classes in task to processor mapping problem and *Features* in the configuration problem; (ii) *Component Attributes*, that can capture the properties of the components. The values of these attributes may be included in the problem model, like `cost` attribute of *Feature* class in configuration example, or maybe calculated by the solver and included in the solution model, for example `select` attribute; (iii) *Association*, that are used to express relations between the core components in the design space, for example *MapsTo* relation, (iii) *Constraints*, that are used to capture conditions that must be satisfied by the solution models, for example the *Cross-Tree* constraints in the configuration example; (iv) *Global Variables* that are required to capture properties associated with a design point in the design space, for example `total_cost`. Besides these concepts that capture the design space, we require a concept to capture the objective of solving the DSE problem. These classes are separated out into a template that is reused associate DSE characteristics with any DSML .

III.3.2 The ADSEL Definition

The ADSEL language definition is specified as a metamodel in GME. In this section we discuss each concept of ADSEL in detail. As a running example to explain the concepts we use the DSE problem illustrated in Section III.2.

Based on the evaluation in Section III.3.1, the ADSEL Metamodel consists of (i) *Component Types*, which are entities used to model the design space in a given problem, (ii) *Association Types*, which model relationship between the component types, (iii) *Constraint Types*, which model the constraints to be satisfied by valid design alternatives, (iv) *Global Variables*, which capture properties of a design solution, for example, `totalcost`. These properties are required to compare the different design alternative during the search, and (vi) *Objective Types*, which captures the goal of the exploration. In the following, we describe each of category in detail and use our simple mapping problem to show the use of these classes in creating the conceptual model for the mapping problem.



(a) ADSEL Component Types (b) Mapping Component Types

Figure 8: Component Types

III.3.2.1 Component Types

The design space in a problem model M_{CM} is represented using a set of *Component* objects. This design space can be enumerative, where the design alternatives are organized hierarchically in a tree like structure as done using *feature models* [62]. Alternatively, the design space can be represented as a partial model, where the space is a represented as a model template and each design alternative is a well-formed model created by completing the model template. The Component classes provide generic means to support both representations. The Component classes are classified into *Primitive* and *Container* class, as shown in Figure 8(a). The *Primitive* class represents a fundamental unit of composition atomic concepts that cannot be decomposed any further. A *Container* component represents a decomposable model concept, that can contain one or more components (*Primitive* or *Container*), shown by containment relationship *Children*. The components form the building blocks for modeling a design space, for both enumerative and partial models.

An additional set of container components, namely, *Mandatory*, *Alternative*, *Option* and *Or* are provided for the convenience of modeling enumerative design spaces. The *Mandatory* class models composition, which means all the child objects are included if the parent is included in a solution model. The *Alternative* class models a choice point where exactly one child object is included. The *Option* class models a choice point where one or none of the child objects is included. Finally, the *Or* class models a choice point where any number of child objects between the `child_min` and `child_max` can be included if the parent is included in a configuration. Essentially only *Primitive* and *Container* component types are enough to create a hierarchy. In such a case additional constraints will have to specified to express the composition constraints. Therefore, *Mandatory*, *Alternative*, *Option* and *Or* are provided for user convenience. Figure 8(b) shows the components in the conceptual model of our simple mapping problem, where the *Task* and *Processor* are modeled as primitives.

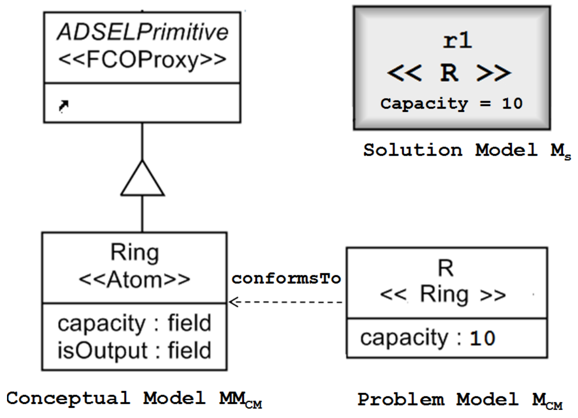


Figure 9: Output Component

A *Component* class by default models an input component where the component objects are included in a DSE problem model M_{CM}, created by the domain engineer. However, DSE problems require a concept to model classes where the component objects are retrieved as a result of the search. This concept is modeled using an *Output* component, captured by `isOutput` attribute, where `(isOutput=true)` for an output component. An output component is a singleton, where a single object of the component is included in the design space model. The DSE properties of this singleton are shared by all the objects returned by the search.

Certain DSE problems require an ordered set of objects, such that each object can be

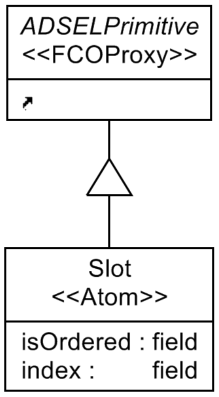
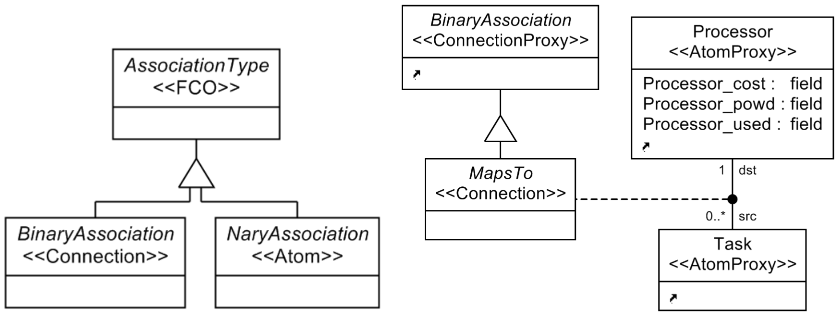


Figure 10: Ordered Component

accessed using a unique index number in the range `[1..cardinality]`. where the `cardinality` gives the total number of objects included in problem model. We model such a set using an *Ordered* component class, reflected by the `isOrdered` attribute. Figure 10 shows an example of an ordered component *Slot*. Each object of *Slot* can be accessed using the `index` property.



(a) ADSEL Association Types (b) Mapping Association

Figure 11: Association Types

III.3.2.2 Association Types

An *Association* class models a relationship between the *Component* instances . There are two kinds of associations classes, as shown in Figure 11(a): (1) *BinaryAssociation* class, which models a relationship between two component types, and (2) *Nary Association* class, which is used to model a relationship between more than two components. The association end points specify the role and multiplicity. The roles are specified using the key words '**`src`**' and '**`dst`**'. In case of Nary associations where there is more than one source or destination contexts, the rolenames '`<`**`src`**`|`**`dst`**`>[N]`' are used, where `N` is a natural number used to uniquely index a context in case of more than one source(destination) components. The multiplicity of the association denotes the number of instances that can participate in the relationship. For example, Figure 11(b) shows a binary association between *Task* (role name: src) and *Processor* (role name: dst), where one instance of *Task* instance can be associated to exactly one instance of a *Processor*, but the *Processor* instance can be associated to zero or more *Task*s. We can reason about the set of all *Association* instances in the model by use of the association name. For example, *MapsTo* refers to the set of all association instances in the problem model. A particular instance of the association can be referenced by function style notation where the association ends points are passed as arguments. For example, '`MapsTo(t,p)`' refers to the association instance between *Task* *t* and *Processor p*. The associations in ADSEL are navigable in both directions although the directionality is implied by the role names. The source to destination navigation in this case is written as '`MapsTo(t,_)`', where *t* is the name of the *Task* instance and returns the *Processor* instance. Reverse navigation is achieved by writing '`MapsTo(_,p)`' where '`p`'p is the *Processor* instance id.

Association classes can be abstract or concrete. An abstract association class models an output relation between components, that is included in the design solution returned by the search. A valid design solution should satisfy the cardinalities of the association. Figure 11(b) shows an abstract *MapsTo* association class. The cardinalities of the association suggest that each *Task* instance in the design solution should be associated to exactly one *Processor* instance by a *MapsTo* association.

All *N-ary* association can essentially be modeled using a set of binary association. Therefore, for the remaining thesis, we restrict our focus to the use of binary association.

III.3.2.3 Properties

The attributes of the DSML elements (for example, cache type) represent the natural characteristics of these model elements but may or may not be useful for design space exploration. In order to express attributes relevant for design space exploration (for example, cost) we associate DSE properties with components and associations. A property specification includes the following parts:

- `Property Type`, which can be {PARAMETER, DECISION, METRIC}. A parameter property is set before the search process whereas a decision property is a result of the search. A metric property depends on other properties of the component, global variables or associations navigable using the parent component.

- `Value Type`, which can be {INT, BOOL, STRING, FLOAT}. Only a parameter property can take string values.

- `Domain`, which can be a single value or set or range of values that the variable can take and depends on the `ValueType`.

- `Assignment Statement`, which specifies the CSL assignment statement (arithmetic or boolean) used calculate the value of a metric property.

In our mapping example, the *Processor* component has three properties, namely, `cost`, `powd` and `used` as shown in Figure 8(b), where `cost` represents the cost of the processor, `powd` represents the power dissipation of the processor and `used` models a condition to check if at least one *Task* object is associated to the *Processor* object. The `cost` and `powd` properties are of parameter type, where the value of the properties for each *Processor* object is included in the DSE problem model. These properties remain constant for the duration of the search. The `used` property is modeled as a metric type property, where the value is calculated using the assignment statement in Figure 12, where '**`$self`**' refers to the *Processor* component. Each property of the component is accessed using a '`.`' operator. The '**`card`**' operator gives the cardinality of the set of *Task* objects associated with the *Processor* using the *MapsTo* association. The statement is written using the constraint specification language discussed in Section III.4.

```
$self.used <-> ( card(MapsTo(_,  $self)) >= 1)
```

Figure 12: Assignment statement for cost property

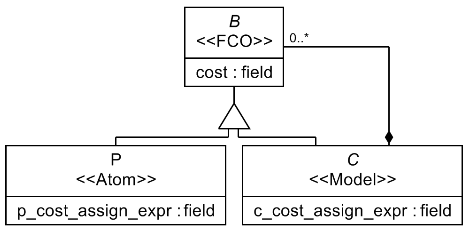


Figure 13: Property Inheritance

Property Inheritance: Properties of a component are inherited by all derived component classes. For example, Figure 13 shows component classes *C* and *P* that are derived from the base component class *B*, where the base component class has `cost` property of decision type. This property is inherited by the derived components and can be accessed like any other property that belongs to the component. For example '`C.cost`', is a valid arithmetic expression. It is possible to constrain the domain of an inherited property. A decision property is the least constrained, thus an inherited property of decision type can be constrained further by using an assignment statement that is a function of other properties of the component. For example, in the feature example, the `cost` decision property of component *B* is inherited by component *C* and is constrained by the following assignment statement,

```
  $self.cost = sum(c  in Children($self, _)  )  (c.select * c.cost)
```

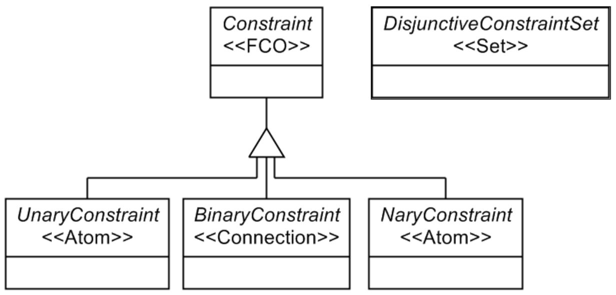
A decision or a metric inherited property can be constrained to a single value input by the domain engineer. For example, the `cost` property in Figure 13 is also inherited by *P* and is constrained to a single value entered by the domain engineer. The is done by the following assignment statement:

```
$self.cost = param($self.cost)
```

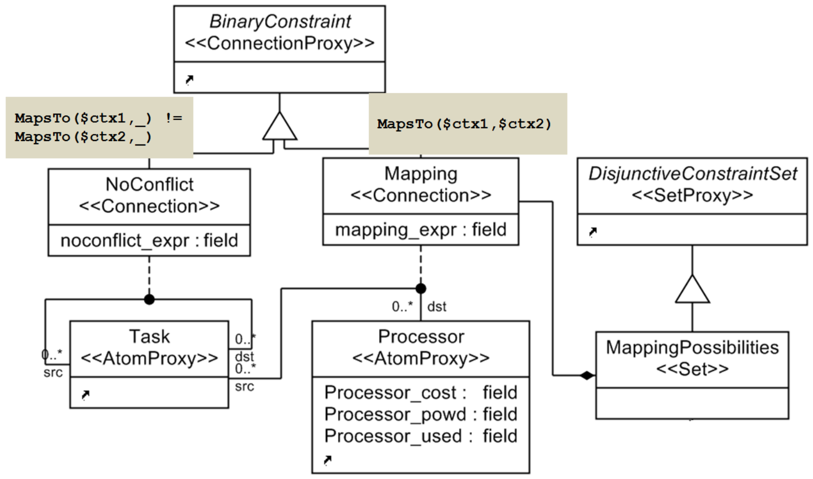
where the function '**`param`**' is used to convert a decision (metric) property to parameter property such that the value of the property is equal to the value included in the DSE problem model.

III.3.2.4 Constraints Types

Constraints are used to model conditions that each valid design solution must satisfy. A constraint is expressed in terms of one or more context component classes and is applicable to a particular component object or all component objects included in the DSE problem model. Based on the number of context components, the ADSEL constraints can be of three types: *Unary*, *Binary* and *Nary* constraints (Figure 14(a)). Each constraint class can be concrete or abstract, where an abstract constraint class models a constraint that should hold true for all instances of the context like an OCL invariant or at least one unknown instance of the context. On the other hand a concrete constraint class models a constraint



(a) ADSEL Constraint Types



(b) Mapping Constraints

Figure 14: Constraint Types

that has to be explicitly instantiated and applies to a single component object in the problem model. All constraint classes have an attribute `<name>_expr`, which captures the constraint definition using the CSL, a simple constraint specification language discussed in Section III.4. We discuss each of the constraint classes in detail.

1. A *Unary Constraint* models a single context constraint, where the context is a *Component*. For example a constraint on the memory property of a component *Module*, say '`Module.memory <= 128`' can be modeled as a unary constraint with the *Module* component as context. The CSL constraint definition is given by '**`$ctx1`**`.memory <= 128`'. If the constraint class is *abstract*, then the constraint definition applies to all *Module* objects in the problem model. If the constraint class is concrete, then it is

explicitly instantiated in the design model M_{CM} and constrains the memory property of an object of the *Module* component.

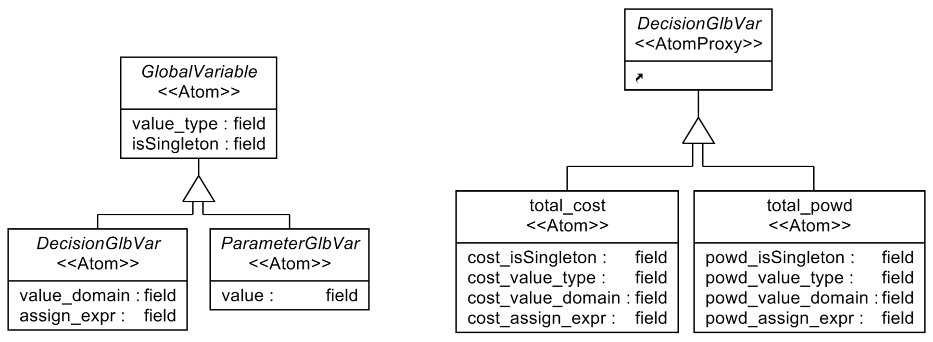
2. A *Binary Constraint* models a dual context constraint used to specify a condition relating two components (and their properties). For example, a selection constraint in the product configuration problem presented in Section III.3.1 relating the selection property of a feature *A* to a feature *B*, such the selection of feature *A* implies selection of feature *B*. This constraint can be easily modeled as a binary constraint *BC* with *Feature* class as both source and destination contexts. The CSL constraint definition is given by '**$ctx1**`.sel -> `**$ctx2**`.sel`', where '**$ctx1**' and '**$ctx2**' refer to source and destination *Feature*s. An *abstract* binary constraint *BC* is applied to all combinations of source and destination instances. A concrete binary constraint is explicitly instantiated in the design space instance model. The contexts of the constraint have rolename '**$ctx**`[N]`', where `N` is a natural number uniquely index each context component. Figure 14(b) shows two binary constraints in our example mapping problem. The *NoConflict* binary constraint models the condition that the context *Task* instances should not be mapped to the same *Processor* and the *Mapping* constraint models that a *Task t* is mapped to *Processor p*. The CSL expression of the mapping constraint is given by '`MapsTo(`**$ctx1,$ctx2**`)`'.

3. *NaryConstraint* models constraints with more than two context components.

By default a valid configuration should satisfy a conjunction of all constraints, except in case of *DisjunctiveConstraint Sets*. A *DisjunctiveConstraintSet* models a set of constraints that are composed using a disjunctive operator. It can contain a set of unary, binary or nary constraints, such that exactly one constraint in the set should be true. For example, in our simple mapping problem a task can be mapped to exactly one out of a possible set of processors. This is represented by a disjunctive set of *Mapping* constraints. Figure 14(b) shows a disjunctive constraint set *MappingPossibilities* that consists of a set of *Mapping* constraints out of which exactly one constraint should hold.

III.3.2.5 Global Variables

Global Variables are essentially DSE properties that are associated with a design solution and cannot be expressed as properties of a particular component or association. For example, `total_wire_length` is a global variable that is used to model the length of nets for a particular placement solution. This property is used to compare the different placement alternatives during the search. Like properties, global variables have a `value_type` attribute that specifies the type of values the variable can take. Additionally, all global variables have an `isSingleton` attribute, which models whether exactly one or more instances of variable can be included in the problem model. By default the global variables are singletons (`isSingleton=true`).



(a) ADSEL Global Variables

(b) Mapping Global Variables

Figure 15: Global Variables

There are two types of *Global Variables*, shown in Figure 15(a):

1. a *Decision* global variable models a property of the design solution that is determined by the search. A decision variable cannot be of `float` or `string` type. The `domain` attribute captures the possible values of the variable and the `assign_expr`

attribute captures the assignment statement used to calculate the value of the variable depending on values of other properties.

2. a *Parameter* global variable, which models a property that is shared by all solution in the design space. The `value` attribute is used to capture the value of the variable instance.

Figure 15(b) shows two global variables in the mapping example that are used to calculate the total cost and total power dissipation of a design solution. The *total_cost* is an integer variable that can take all non-negative values. The assignment statement is given:

```
        $self = sum(p in Processor)  (bool2int(p.isUsed)*(p.cost))
```

III.3.2.6 Objective Types

This captures the goal of solving the DSE problem. There are two kinds of objective functions (shown in Figure 16): (i) *Satisfy* objective is used to perform constraint-based DSE where the goal is to find design alternatives that satisfy all constraints, and (ii) *Optimize* provides a placeholder for specifying the cost functions used to compare the alternatives in the design space.

The *Optimize* objective contains one or more *Objective Variables* that are essentially global decision variables or decision properties of components. The `property_name` attribute gives the name of the property under consideration. Each objective variable has a `weight` attribute which is used for aggregation when a CP solver is used for multi-objective optimization. In the mapping example for instance, the objective is to find a design that satisfies the constraints and has minimum cost and power dissipation. The objective variables are *total_cost*, *total_powd*.

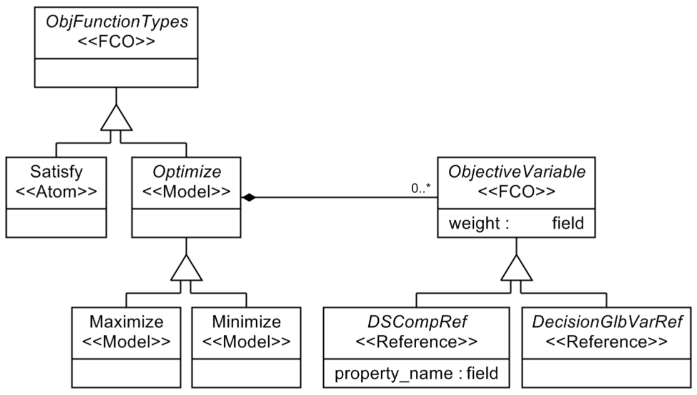


Figure 16: Objective Types and Objective Variables

III.4 The Constraint Specification Language (CSL)

In DSE, constraints are often used to specify logical conditions that the design solution should satisfy. These conditions are often in the form of bounds on certain properties of a design solution. For example, "`the memory consumption of the system should be less than 100`". They can also be in the form of a relationship between concepts that the design solution should honor. For example in a task to processor mapping problem [51], a typical constraint can be "`Always deploy Task A and Task B on the Processor p`". These constraints are used by the search methods to prune out design alternatives that do no satisfy them and move towards valid solutions. The efficiency of the search and the correctness of the returned solution is highly dependent on the way the constraints are formulated. Therefore, correct formulation is of the essence while solving DSE problems.

The generic framework presented in this dissertation is a model based framework, where the constraint definitions are specified in conjunction with the conceptual model of the problem. We had four requirements for a constraint specification language . Firstly, we required a *declarative* constraint specification language that can be used to annotate the concepts in the conceptual model of the problem. Secondly, keeping in line with the generic scope of the framework, we required an *expressive* constraint specification language that can adequately specify a wide range of constraints, from simple relational constraints, to complex structural constraints that are specified on graph-based model of the system. Thirdly, given the importance of constraint formulation for search, we require a constraint specification language that supports a *simple syntax* so that the modeler can specify and debug complex constraints. Moreover, the expressiveness of the language should not add to the difficulty of using the language. Finally, the constraints specified in the conceptual model are translated to constraint expressions in lower-level constraint language supported by the solvers. Therefore, the constraint language should be such that the constraint expressions can be easily translated to a number of solver-specific formats.

Existing model-based frameworks (including GME) use textual constraints to include information about the static and dynamic semantics of the language that cannot be included in the graphical model. This includes constraints that specify state invariants, guards in state machines, constraints in sequence diagrams, and pre and post condition of operations. In the generic framework, the constraints are used to specify conditions that must be satisfied by valid design solutions. In essence these constraints are analogous to invariants that must hold for all valid models that conform to the language. Therefore, we focus on how invariants are specified using constraint specification languages like the Object Constraint Language (OCL) [39] that are used in conjunction with UML, as well as formal specification languages like, Alloy [56] and Z [120], that are used for analysis of the UML models.

The OCL is a standard language that is widely accepted for writing constraints on UML models. It is based on first order logic and is used to specify constraints for various types of models. It was developed with the aim of being easier to use for the modelers as compared to formal specification languages like Z. OCL is tightly integrated into the UML and contains features for navigating across models by using association roles. This makes is easier to specify structural constraints in OCL. The ease of use and expressiveness made OCL a good candidate for the generic framework. As a result an initial attempt included using an extended subset of OCL, similar to [89], to specify the constraints in DSE problems.

Figure 17 shows an example constraint in OCL specifying that two *Ring* objects should be associated with disjoint set of nodes using the association *RingNode*.

```
OCL context: Ring
OCL expression:
   Ring.allInstances ->forall (p, q | p.name != q.name implies
   p.RingNode->intersection (q.RingNode)->isEmpty)
```

Figure 17: Example constraint expression in OCL

Although OCL is expressive and user friendly, there are two disadvantages of using OCL as a constraint specification language for our framework. The first disadvantage is the use of single-context constraints. In OCL, all constraints are written in context of a *single* class and in order to reason about other classes (parent, children) a chain of navigations are used. This can sometimes make simple OCL constraints verbose and hard to read [126]. Moreover, the OCL constraints use quantifier stacking that adds to the complexity when multiple levels of navigations are used. Both these problems can be eased by avoiding navigation chains by use of multi-context constraints that require only one step navigation. The second disadvantage of using OCL comes from the complexity in translating arbitrary OCL constraints to logic [25, 27]. This complexity arises from the expressiveness of OCL, which makes specifying of constraints easy for the user but the analysis of those constraints hard. This can complicate refinement of arbitrary constraints to lower-level constraint languages (constraint logic) supported by the optimization models. A third minor issue is the lack of support for creating constraint templates, that can be created by the domain-expert and used by the domain-engineer by plugging in values.

In contrast to the OCL that is tightly coupled with UML, textual formal specification languages like Alloy [56], B[6], Z[120] are used to specify the entire model along with the constraints. These languages were built to support automated analysis of the UML models. Alloy is a specification language based on first-order relational logic. Everything in Alloy is a relation, including sets and scalars, where sets are considered 1-tuple unary relation with each 1-tuple in the unary relation representing an element of the set. Similarly scalars are considered a singleton (size 1) unary relation. This provides uniformity in syntax and semantics. For example $e_1 + e_2$ represents union of sets if e_1 and e_2 are sets. The invariants in Alloy are written as $facts$ and always hold . The biggest advantage of Alloy is the support for automated analysis provided by *Alloy Analyzer*, that analyzes the model for consistency and if the model is consistent returns a model. Alloy analyzer is based boolean satisfiability and is used for model search in small design spaces because of the combinatorial explosion as the complexity of the model and constraints increase. Therefore, we cannot use Alloy analyzer for our framework, but Alloy can still be used as a stand alone language for specifying constraints.

```
    all p, q | p.name = q.name -> no (p.RingNode & q.RingNode)
```

Figure 18: Example constraint expression in Alloy

A large body of work exists that refines UML model along with OCL constraints to formal specification language for consistency checking [8, 9, 77], but the languages are rarely used in conjunction with UML models to specify constraints. One instance where a formal specification language (B) is used as an action and constraint language with UML models is the UML-B profile [118]. One obvious reason for this lack of integrated use is that formal specification languages can textually capture the entire model, so there is essentially no dependence on the UML models. A stronger reason is the semantic difference between formal specification languages and UML. For example, Alloy supports only integers as opposed to UML, that supports integers, strings, booleans etc. Another difference between UML and Alloy in the uniform treatment of sets and relations. In UML, sets and scalars are viewed as distinct types whereas in Alloy everything is a relation. In the generic framework we need a constraint language that can use the declarations in the conceptual model and specify invariants in context of one or more classes. In such a case writing constraints which make no distinction based on types can create confusion. For example, if the constraint expression includes a 'e_1 in e_2' is used, it is hard to figure out whether the expression means that e_1 is a set that is a subset of set e_2 or whether it means that p is a scalar that is an element of set e_2. Consequently, we need a language that can be used to specify readable constraints using notation that is familiar to the modeler rather than requiring him to learn a new language.

In pursuit of this goal, we have developed Constraint Specification Language (CSL), a simple user friendly language used to specify constraints. The CSL is expressive enough to specify complex structural constraints. The CSL is based on first-order logic and set theoretic concepts. The goal was to have the bare essential constructs for specifying readable constraints and yet be easy to translate to a CLP constraints or an imperative boolean functions. The CSL can be used to specify multi-context constraints to avoid navigation chaining in the constraint definition. Additionally, the CSL supports quantifier chaining instead of stacking to maintain the simplicity of the constraints. The generic framework support constraint templates, such that the domain-engineer can *write* constraints by providing values for the template parameters.

```
forall($ctx1, $ctx2)(
    ($ctx != $ctx2) ->
    (RingNode($ctx1,_) intersect RingNode($ctx2,_) == {})
)
```

Figure 19: Example constraint expression in CSL

III.4.1 Use Cases

The CSL expressions can be used in two ways in the conceptual model: (1) Specify constraint definitions, (2) Specify assignment statements for metric properties/global variables. We discuss the format of each in this section.

Notation : We use the following syntactic convention given in Appendix B.1. Throughout this section, Boolean expressions are denoted with uppercase letters from the beginning of the alphabet (A, B, C, etc.); integer expressions are denoted with lower-case letters from the ending of the alphabet (x, y, z, etc). Domains are written in capital letters (e.g. D1, D2) and the elements of the domain are in lower-case letters (eg. d1,d2).

Constraint Definition Format: A CSL constraint definition is always written in the following format:

$$[\quad (\ \langle \mathit{Quantifier} \rangle\ \underline{(}\,\langle \mathit{Context} \rangle \underline{)}\ )^*\ ]\underline{(}\quad \langle \mathit{ConditionalExpression} \rangle\quad \underline{)}$$

where the *Quantifier* represents either '**`forall`**' or '**`exists`**', *Context* is a context class that are accessed using the context-keyword (for example **`$ctx1`**). The scope of this context is limited to the quantified *ConditionalExpression*. The quantifier and thereby the quantified-context is optional in the constraint expression, but there can more than one conjuncted by '$\cdot$'.

```
    Context: $ctx1 = Node , $ctx2 = Node , $ctx3 = Processor
    CSL:   forall(<$ctx1,$ctx2> in Edge) . forall( $ctx3)(
                 MapsTo($ctx1,$ctx3) ->  MapsTo($ctx2, $ctx3)
              )
```

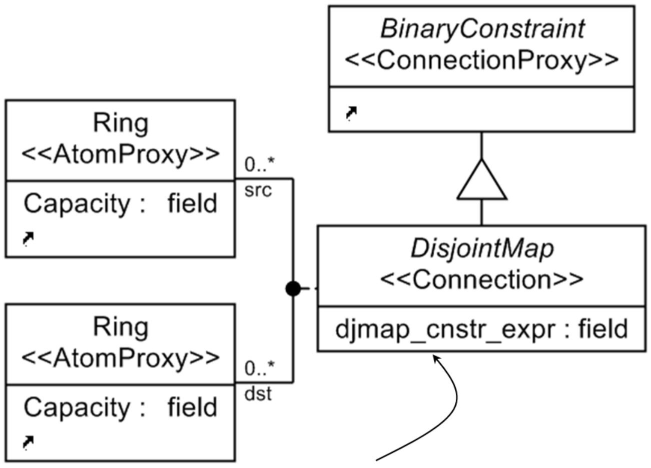
Figure 20: Example constraint in CSL

Figure 20 shows an example CSL expression of an nary-context constraint definition, where the constraint has three contexts. The first two contexts refer to *Node* components and the third context refers to the *Processor* component. The expression specifies that if two components are connected by an *Edge* association, then both the Nodes should be mapped to the same processor. The quantifiers are chained outside the conditional expression.

The use and type of quantifier in the constraint definition depends on whether the constraint class is concrete or abstract. A *concrete* constraint class, which models a constraint template does not include quantifiers or a list of formal parameter. An *abstract* constraint class models a constraint expression that should hold true for all instances of the context or at least one unknown instance of the context. The constraint definition in this case includes quantifiers to express the scope of the constraint. A universal quantifier (‘**`forall`**’) specifies that the constraint is applicable to all instances of the quantifying-context. An existential quantifier (‘**`exists`**’) specifies that the constraint is applicable to at least one instance of the context component. The universal and existential quantifiers are always specified *outside* the CSL expression and cannot be included in the conditional expression.

A CSL constraint is always written in terms of contexts of the constraint class. These context classes are accessed in the constraint definition by use of keyword ‘**`$ctx`**[*N*]’, where *N* is the context index number. For example, the source and destination classes of *DisjointMap* (Figure 21) are accessed using the keyword ‘**`$ctx1`**’ and ‘**`$ctx2`**’ respectively. Another context keyword, ‘**`$self`**’ is used to access the constraint class. These are reserved keywords in CSL and cannot be used as variable names. The context in the constraint definition can be a single context keyword, for example ‘**`$ctx1`**’, ‘**`$self`**’, etc or it can be a tuple of context-keywords, where the context components are also the ends points of an association, for example ‘**`<$ctx1, $ctx2> in`** `MapsTo`’, where *MapsTo* is an association between the context classes referred by ‘**`$ctx1`**’ and ‘**`$ctx2`**’.

Figure 21, shows a typical use-case for a CSL constraint, where *DisjointMap*, a binary constraint is used to represent a constraint between the *Ring* objects. The constraint class is abstract, which means that the constraint is applicable to *all or any* pair of instances.



```
forall($ctx1).forall($ctx2)
   ( ($ctx1 != $ctx2) ->
        (RingNode($ctx1,_) intersect RingNode($ctx2,_) == {})
   )
```

Figure 21: CSL Constraint

Informally, the constraint specifies that if two *Ring* instances are different then the set of nodes mapped onto them are disjoint. The CSL constraint definition is specified as the value of `cnstr_expr` attribute. The *RingNode* is a binary association from *Ring* to the *Node*, such that '`RingNode(r,_)`' is used to specify the forward navigation of the association to retrieve *all Node* instances associated with *Ring r* using the *RingNode* association. In the constraint definition this association is accessed using the end points. This notation makes it easy to express this constraint is a succinct syntax.

CSL Assignment Statement Format: CSL is also used to specify assignment statements that are used to set the value of a metric property.

$$\langle Context\rangle\ \langle AssignOp\rangle\ \langle AssignmentExpression\rangle$$

where the *Context* represents either a metric property, accessed by using '**`$self.`**$\langle property\rangle$', or a decision global variable accessed by '**`$self`**'. The *AssignOp* can '**`=`**' if the property (variable) is an integer, or '**`<->`**' if it is boolean. The *AssignmentExpression* can be a boolean expression or an arithmetic expression depending on the type of the variable. It can also be a special method supported by CSL. For example, '**`param`**'.

III.4.2 The CSL Definition

In this section we discuss the features of the CSL in detail. The CSL expressions include context keywords, which always refer to the components in the conceptual model. The expression can use the context properties, the associations that can be navigated using the context components, and all the global variables defined in the conceptual model. Local variables in a CSL expression can only be introduced as quantified variables whose scope is limited to the corresponding quantification.

Notation: Throughout this section, classes in the conceptual model are denoted using *italics*, for example *MapsTo*, *Task*, etc. The domain corresponding to these class are in `typewriter` text, for example `MapsTo`. The keywords and operators included in the CSL are in bold letters.

III.4.2.1 Data Types

The two basic data types supported in the CSL are **Integer** and **Boolean**. The advanced data type supported in the CSL is **domain**, which is essentially a set of *objects* of a component class, or a set of tuples of component objects associated by an association class. The domain is referred to by the name of the component (association). For example '`Task`' in a CSL expression refers to a set of *Task* objects, where the *Task* component is declared in the conceptual model. Corresponding to the data types, the literals in CSL includes integers literals, example '**`1,4`**' and boolean literal, for example'**`true`**', '**`false`**', and set literals, for example '**`{}`**' that denotes an empty set.

III.4.2.2 Domains

Domains are used to model finite sets in the CSL. There are two types of domains: the component domain and the association domain.

Association domain is specified by the name of the association class. For example, '`MapsTo`' is an association domain and refers to a set of tuples of objects, where each tuple is a pair of *Task* and *Processor* objects connected by the *MapsTo* association. Note that '`MapsTo`' cannot be used in a CSL expression unless *MapsTo* association is included in the conceptual model.

Component domain is specified by the component name. For example, '`Task`', in the CSL represents a set of *Task* component objects. A component domain can also be derived from an association domain using '**`sources(`**$\langle Assoc \rangle$**`)`**' or '**`targets(`**$\langle Assoc \rangle$**`)`**' method to retrieve a set of source or destination context objects. A component domain can also be retrieved by association navigation. For example '`MapsTo(p,_)`' returns a component domain with *Processor* instances associated to *Task t* and similarly '`MapsTo(_, p)`' returns a component domain with *Task* instances.

Domains are essentially sets, and therefore it is possible to use set operators '**union**' and '**intersect**' on two domains consisting of *same type* of component objects.

Table 3: Set Operators yielding Domain Expressions

Boolean Operators	
`D1 intersect D2`	$D1 \cap D2$
`D1 union D2`	$D1 \cup D2$

III.4.2.3 Basic Expressions

Every expression in CSL is either a boolean or integer. In this section, we cover the basic expression that are either atoms or are constructed by unary or binary operators. Complex expressions such as the quantified expression are discussed in Section III.4.2.4.

Table 4: Unary and Binary Operators yielding Boolean Expressions

Boolean Operators

`not(A)`	negation of A	$\neg A$
`A /\ B`	A and B	$A \bigwedge B$
`A \/ B`	A or B	$A \bigvee B$
`A->B`	A implies B	$A \rightarrow B$
`A<->B`	A is equivalent to B	$A \leftrightarrow B$

Relational Operators

`x == y`	equality	$x = y$
`x <= y`	less or equal	$x \leq y$
`x >= y`	greater or equal than	$x \geq y$
`x != y`	inequality	$x \neq y$
`x < y`	less than	$x < y$
`x > y`	greater than	$x > y$
`d1 == d2`	equality	$x = y$
`d1 != d2`	equality	$x = y$

Set Operators

`D1 seq D2`	D1 equals D2	$D1 \subseteq D2 \wedge D1 \subseteq D2$
`D1 sneq D2`	D1 not equal to D2	$D1 \neq D2$
`d1 in D1`	d1 is an element of D1	$d1 \in D1$

Basic Boolean Expressions are either atoms or expressions composed by operators that yield Boolean expressions. A boolean atom can be constants (‘**`true`**’or ‘**`false`**’). It can also be a Boolean property of a context, for example ‘**`$ctx`**`1.select`’, ‘**`$self`**`.isUsed`’, etc. Boolean atom can also be a property of an element of the domain, for example ‘`p.isUsed`’, where‘`p`’ is a local variable representing element of a domain. The domain (component or association) should have the property that is being accessed by the variable. For example

'`Edge(`**`$ctx1,$ctx2`**`).select`', accesses the'`select`' property of an instance of *Edge* association. A boolean atom can also be a condition to check the presence of an element in the domain. For example '`MapsTo(t,p)`' is an atomic boolean expression, which checks the presence of an instance of *MapsTo* association between '`t`' and '`p`'. Similarly, '`c in Channel`' is a check on the presence of an element '`c`' in the domain '`Channel`'. Table 21 summarizes all unary and binary boolean, relational and set operators that yield Boolean expressions. A and B are arbitrary Boolean expressions, x and y are arbitrary integer expressions and $D1$ and $D2$ are domains of the same type. These operators are standard in most solver-independent modeling languages.

Table 5: Unary and Binary Operators yielding Integer Expressions

Binary Operators

`x - y`	subtraction of A	$x-y$
`x + y`	addition	$x+y$
`x * y`	multiplication	$x*y$
`x / y`	division	x/y

Unary Operators

`-x`	negative x	$-x$
`abs(x)`	absolute x	$\|x\|$
`card(D)`	cardinality of domain set	
`bool2int(A)`	boolean to integer	

Basic Integer Expressions are either atoms or expressions composed by operators that yield integer expressions. An integer atom can be a constant, (for example 0,1). An integer atom can be property of the context or property of an element in the domain (component or association). Table 5 shows the operators and methods that yield integer expressions. The CSL supports the '**`card`**' method that returns the number of elements in the domain, for example '**`card`**`(MapsTo)`'. The CSL also supports a '**`bool2int`**' method that converts a boolean expression to an integer (0,1) depending on the value. An integer atom can also be the minimum (or maximum) element of an ordered domain set.

III.4.2.4 Quantified Arithmetic Expressions

Quantifications are a compact means of representing a variable number of constraints. The general syntax is

$$\langle Quantifier\rangle \underline{(}\langle Quantifying-Var\rangle \; \texttt{in} \; \langle Domain\rangle \underline{)(}\langle ArithmeticExpression\underline{)}$$

where *quantifier* represents either '**`sum`**', '**`min`**' or '**`max`**', *quantifying-variable* is a temporary variable that is an element of the *domain*, for example '`c` **`in`** `Children($self,_)`'. The scope of this local variable is limited to quantified expression. Only arithmetic quantifiers can be included in the conditional expression and these quantifiers can be nested.

III.4.2.5 Special Methods

'**`param`**' is a unary operator on a decision (metric) component property that is used to constrain the property to a single value that is input by the modeler. A detailed grammar specification of CSL can be found in Appendix B.

III.5 The Intermediate Language (IRL)

Existing approaches for automated DSE integrate one general-purpose solver that is most suited for their needs. The choice of the solver is based on the requirements of the DSE problem that the framework is intended to solve. The requirements include the objective of solving the DSE problem (global optimal solution, all valid solution), and types of constraints (linear, non-linear). In a generic framework, supporting a single solver is not sufficient to solve problems with arbitrary combinations of objective and constraints. In this case set of model compilers would have to be written to support solution of every problem modeled in the framework. However, developing even one model compiler that directly transforms a domain-specific model of the problem to solver-dependent format is a hard and time consuming task. Therefore, writing a number of them is not a feasible approach.

In order to simplify the compiler development and maintain solver flexibility we developed a solver-independent abstraction level using an Intermediate Language (IRL). This abstraction level includes simple set theoretic concepts like sets, functions and relations that can be used to specify a high-level mathematical model of the DSE problem. The goal is to automatically transform the conceptual model MM_{CM} of a DSE problem to a model M_{IRL} in the IRL. This transformation distills out the domain-specific aspects of the DSE problem and refines the DSE aspects to create a simple mathematical model. The model M_{IRL} is then automatically transformed to solver-specific models that can be processed by the solvers to retrieve solution. The IRL simplifies the compiler development by requiring only one translator from the conceptual model to IRL to be built and maintained. The IRL is at an abstraction level where the concepts can capture all the information in the conceptual model in a solver-independent way using concepts like sets, functions, relations, constraints and predicates, making the transformation of IRL models to solver-dependent formats relatively simple.

Besides simplifying the compiler development, the IRL has two advantages. Firstly, it enables solver flexibility, such that a single DSE problem model M_{CM} can be solved with different solvers depending on the objectives. Secondly, the IRL provides an extension point in the framework that can be used to add more solvers to the framework. At present the framework supports two solver-dependent models: (a) a constraint programming [16] based (CP) model that is used to solve satisfaction and mono-objective optimization problems, and an (b) evolutionary algorithm [47] based (MOP) model used to solve multi-objective optimization problems. Although these two models solve a wide range of problems, more solvers can easily be integrated into the framework given the simplicity of the IRL. This will allow the domain-engineers to compare the performance of difference solvers in a solving a particular instance of the problem.

While defining the IRL, the main challenge was to find a reasonable middle ground between the graphical conceptual model and the solver-dependent models that can take any from ranging from a logic program to an language code. Our main research goals were to define a language such that - (a) every conceptual model created in the framework can be transformed to well-formed model in the IRL, and (b) a model can be transformed to a set of solver-dependent models. In order to achieve the first goal, the initial version of the IRL [108] focused on ease of model generation from the conceptual model. This version of the language was created by removing all the visual and other non DSE related details from the conceptual models. This resulted in a language that had a one-to-one mapping with the ADSEL classes and ensured every conceptual model could be refined to a well-formed model in IRL. However, it made the refinement of the IRL model to the different solver-specific models a challenging task. Another version of the IRL was similar to a high-level constraint programming language. This version worked well while solving DSE problems using constraint programming, but made translation of conceptual models to other solver-specific formats, like evolutionary algorithm program hard.

Finally, in order to achieve our goal, we started with the most basic notions from set-theory: sets, functions, relations and operations on them. This notation has been used to describe the semantics of UML class diagram [110]. As all conceptual models are essentially annotated UML class diagrams, this notation is rich enough to specify the conceptual model, which ensures that every conceptual model in the framework can be transformed to a well-formed model in this language. Moreover, the a subset of the set theoretic concepts included in the IRL have been translated to constraint logic [54] and boolean logic [24] giving us the confidence that a model in the IRL can be translated to logic programs with ease.

The IRL is a declarative language, defined as a metamodel in GME. In this section we discuss the parts of an IRL model and the IRL definition. The rules used to transform an IRL model to a CP model and an MOP model are given in Appendix D and Appendix E respectively.

III.5.1 The IRL Model

A model M_{IRL} in the IRL can be formally defined as a 4 tuple:

$$M_{IRL} = \langle I_{IRL}, O_{IRL}, C_{IRL}, Obj_{IRL} \rangle \tag{III.1}$$

where

- I_{IRL} is a set of fixed input data that remains constant during the search. This includes an *input parameter* whose value is read from the data generated from the DSE problem model M_{CM} and constants whose value is included in the model itself. The input parameters and constants are generated from concrete components and associations in the conceptual model MM_{CM} of the DSE problem.

- O_{IRL} is a set of *decision variables* whose value is determined during the course of the search based on the constraints and the input parameters. A decision variable may also be calculated based on the value of other decision variables as well as input parameters and constants. Decision variables are generated from abstract components and associations in the conceptual model MM_{CM} of the DSE problem.

- C_{IRL} is a set of logical expressions written in terms of the input parameters, decision variables and constants in the IRL model. These constraints must be respected during the search. These constraints are generated from constraint classes in the conceptual model MM_{CM} of the DSE problem.

- Obj_{IRL} is a set of all decision variables that *can* be used in the optimization function. The actual objective of the search is specified in the design space model M_{CM}

and contains a subset of the objective variables. Moreover, the design engineer can change the objective of solving a DSE problem, every time he invokes the search method.

IRL is an internal format of the framework, each model M_{IRL} is automatically generated from the conceptual model MM_{CM} of a DSE problem using the *CM2IRL* translator (Figure 3). Appendix C presents the rules for translating each concept in the conceptual model, including the CSL constraints to corresponding concepts in the IRL model.

III.5.2 The IRL Definition

In this section we discuss three features supported by the IRL: the data types, the constraint types and finally the constraint expressions.

III.5.2.1 Data Types

The IRL is a strongly typed language where every variable (parameter or decision) has a data type. Figure 22 shows the data type hierarchy supported by IRL. There are three main data types in IRL: (1) Predefined types, (2) Set Types, and (3) Compound Types.

Basic Types: The *irlPredefinedType* models common primitive data types, namely Boolean, Integer, String and Float. A parameter variable can be of any data type. The float and string type parameter variables are used as arguments for function definitions only. A decision variable is restricted to be of boolean or integer type, since the framework is used to solve only discrete exploration problems. It is possible to assign a value to a decision variable using an *assignment expression*, a processed CSL expression built on variables in the IRL model.

The *irlSetType* represents the types of sets supported in IRL: (1) *irSet*, which models a simple set containing values from a set *sdomain*, such that *irlSet* $\subseteq$ *sdomain*, where *sdomain* is a set literal or a set constant. For example *irlSet A* with domain set $\mathbb{Z}$ represents a set of integers; (2) *irlIntervalSet* is an integer set whose elements are in a range `[1...max]`; Both parameter and variable can be of the supported data types, and (3) *irlCompoundSet*, which models a set formed by union of other simple or compound sets. This set type is similar to a `struct` in C programming language.

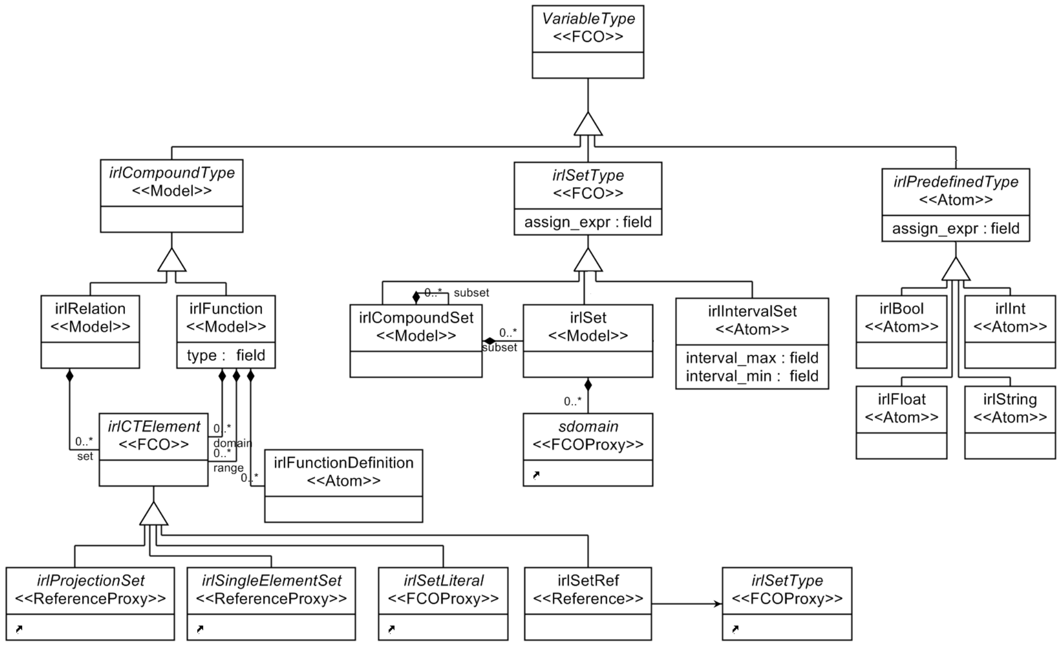


Figure 22: Data Types in the Intermediate Language

Compound Types are used to model functions and relations on sets (can be literals or constants). An *irlRelation* is used to a model a relation $R \subseteq A \times B$, where A and B are sets. These sets can be existing simple sets (*irlSetRef*) in the IRL mode, or projections of the compound types, single element sets, or set literals).

An *irlFunction* is used to model a function $F : A \rightarrow B$, where A and B are sets with composition roles `domain` and `range` respectively. These sets are obtained from similar sources like the relation sets. In our framework, we consider only total functions, such that for a function $F : A \rightarrow B$, every element $a \in A$, there is exactly one $b \in B$ such that $(a,b) \in F$.

If B is a powerset of a set C, then the *irlFunction* models a function $F : A \to \mathscr{P}(C)$. If there is more than one set with `domain` (`range`) role, then the *irlFunction* models a function $F : A \times B \to C$, where A and B are sets with `domain` role. Additionally, a function in the IRL model can be `injective`, `surjective` or `bijective`, captured using `type` enumerated attribute. The type of the function imposes additional constraints on the function. A function can also include a *Function Definition* that models a process with inputs and outputs, where the `domain` set represents the inputs and `range` set represents the output of the function.

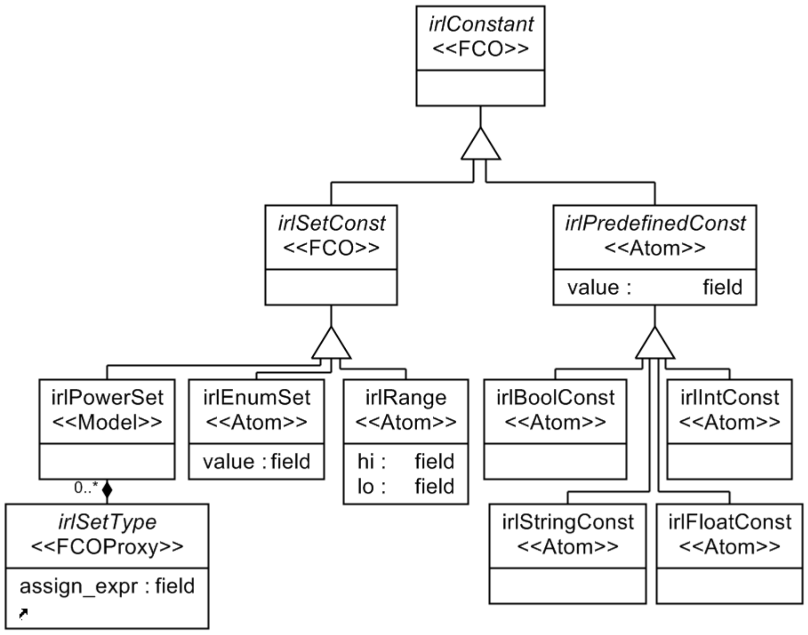


Figure 23: Constants in the Intermediate Language

III.5.2.2 Constants and Literals

The constants declared in an IRL model remain fixed during the course of the exploration. The difference between constants and parameters is that the value of the parameters is read in from the data generated from the design space instance. The IRL supports basic constants of integer, boolean, float and string types. Besides this, the IRL supports four set constants: (1) *irlRange*, a set that can be expressed either as range, for example `[1..4]`, (2) *irlEnum*, a set whose value is specified by enumerating the elements, for example as `{1,2,3}`, (3) The *Power Set*, models power set of a set in the IRL model;

The IRL supports four set literals, namely set of all integers $\mathbb{Z}$, set of all booleans $\mathbb{B}$, set of all natural numbers $\mathbb{N}$, and Kleene star applied to character set $\mathbb{S}$.

III.5.2.3 Data Type Conversion

The IRL also supports type conversion methods to cast functions, relations and predefined types to sets. For example, a method *ImgSet* is used to retrieve the image set of a function. Similarly, *DomSet* and *RngSet* methods are used to retrieve the domain and range sets of a function respectively. The *irlRelationProjection* method is used derive projection for any set in a relation using an index number, assuming each of the sets in the relation is associated with an index number. A variable of a predefined type can be converted to a *Single Element set*.

III.5.2.4 Constraint Types

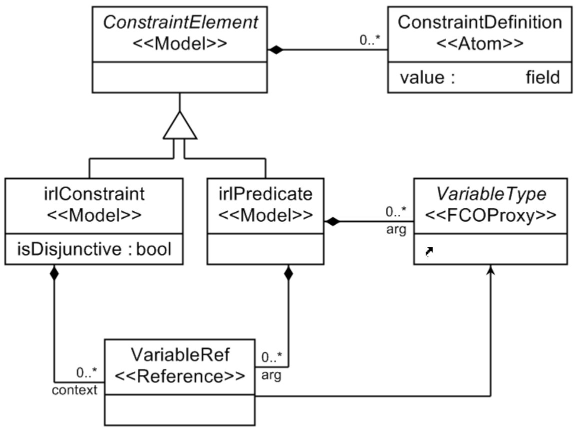
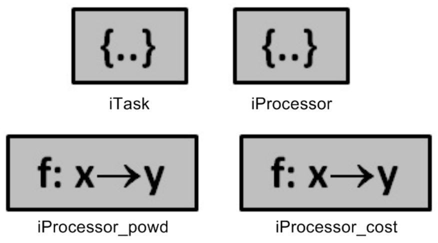


Figure 24: Constraint classes in the Intermediate Language

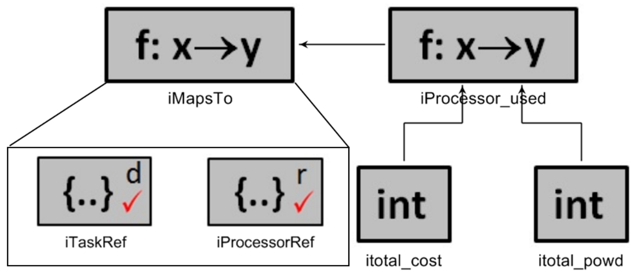
The constraints in IRL are classified into constrains and predicates shown in Figure 24.

An *irlConstraint* represents a global constraint expression that is applicable to all (or at least one unknown) instances of the context. The context of a constraint is a decision variable or an input parameter of set type. An *irlPredicate* models a constraint template that is instantiated in the instance model and thus expresses an explicit constraint between context instances. An argument of a predicate is always a parameter variable. These constraint classes are annotated with logical expressions to capture the constraint definition. The expressions are written in terms of the decision variables, inputs parameters in the IRL model. These expressions are referred to as ***processed CSL (pCSL)*** constraints.

III.5.3 IRL Model of the Simple Mapping Problem



(a) IRL Inputs



(b) IRL Outputs

Figure 25: The IRL Model of the Simple Mapping Problem

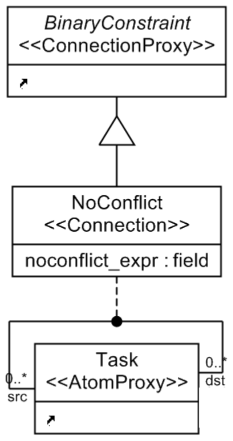
Figure 25 shows the IRL model automatically generated from the conceptual model MM_{CM} of the mapping problem (refer to Appendix A) using the *CM2IRL* translator. Appendix C presents the rules for translating each concept in the conceptual model to concepts in the corresponding IRL model. This translation also includes the processing of CSL expression to generate logic expression in terms of sets, functions and relations (pCSL).

The *Task* and *Processor* components in MM_{CM} are transformed to input parameters *iTask* and *iProcessor*, that are sets of strings. By default all components in the conceptual model are refined to input parameters unless they are output components. The parameter properties `cost` and `powd` of the *Processor* are refined to parameter functions $iProcessor\_cost : iProcessor \rightarrow \mathbb{Z}$ and $iProcessor\_powd : iProcessor \rightarrow \mathbb{Z}$ (Figure 25(a)).

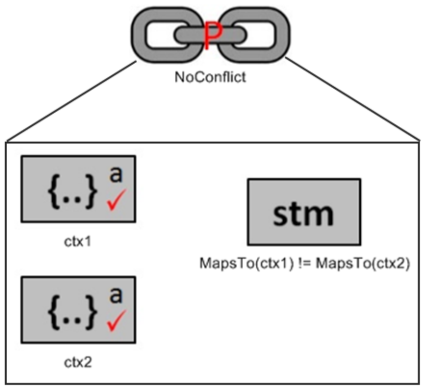
The *MapsTo* association between the *Task* and *Processor* components is translated to *iMapsTo*, a decision variable of function type in the IRL model (Figure 25(b)). By default all concrete associations are translated to input function variables and all abstract associations are translated to decision variables in the IRL model. The `isUsed` boolean property of the *Processor* component generates a decision function variable $iProcessol\_isUsed : iProcessor \rightarrow \mathbb{B}$. The value of this property is calculated based on the assignment statement:

```
    iProcessor_isUsed(ctx)<-> (  (card( iMapsTo(~ctx) )  > 1 )
```

This assignment statement is generated from the corresponding CSL assignment statement of the `isUsed` property included in the conceptual model of the problem. The original statement in CSL is rewritten in terms of sets, functions, relations and their operations. Appendix C.5.1 presents a summary of the rewriting rules used to translate CSL to pCSL expressions. As shown by the assignment statement, the value of the *iProcessor_isUsed* function depends on the value of the function *iMapsTo*. This dependence is also graphically modeled as a connection, shown in Figure 25(b). These dependencies create a Directed Acyclic Graph (DAG) that can be used to identify independent search variables for propagation in the CP model. The DAG is also used to identify the order of variable assignment in the MOP model.



(a) No Conflict Constraint in the Conceptual Model



(b) No Conflict Predicate in IRL Model

Figure 26: Constraints for the Simple Mapping Problem

Figure 26 shows the *NoConflict* constraint in the conceptual model of the mapping problem and the IRL predicate generated by the translator. In the conceptual model, the $NoConflict$ constraint is a concrete binary constraint with the *Task* component as both contexts, where the constraint definition is given by the CSL expression shown. This concrete constraint is translated to $iNoConflict$, a predicate with arguments $ctx1$ and $ctx2$, such that $ctx1, ctx2 \in iTask$. The constraint definition is rewritten to reflect functional notation. Therefore '`MapsTo(ctx)`' refers to an element $p \in iProcessor$, such that $\langle ctx, p \rangle \in iMapsTo$

III.6 Constraint Satisfaction and Mono-Objective Optimization

The generic framework provides support to solve constraint satisfaction and mono-objective DSE problems modeled in the framework. We in this section we discuss this solver support in detail.

The goal of solving a constraint satisfaction DSE problem is to retrieve all valid design solutions that satisfy design constraints. A number of constraint satisfaction techniques, such as boolean satisfiability [37], constraint programming [10], symbolic constraint satisfaction [24] have been used to solve DSE problems (refer to Section II.1). A constraint satisfaction DSE problems can be solved using any of the techniques. However, constraint programming was found to be the least restrictive in terms of formulation (both integer and binary variables allowed) of the problem, which suited the requirements of the generic DSE framework. All other approaches (boolean satisfiability, symbolic constraint satisfaction) were found to be too restrictive as the problems had to formulated using boolean variables and propositional logic formulas. Encoding these constraints as propositional formulas can easily lead to combinatorial explosion in the size of the formula [26].

The goal of solving a mono-objective DSE problem is to retrieve a single valid design solution that is optimal with respect to a single objective function. A set of techniques, such as linear programming [109], branch and bound [75], local search [5] have been used in the past to solve mono-objective DSE problems (refer to Section II.2). Branch and bound is an enumerative algorithm used to retrieve global optima in a design space. Most constraint programming based solvers support mono-objective optimization using branch and bound algorithm. Given that we require constraint programming solver for solving constraint satisfaction problems, we select the branch and bound algorithm for solving mono-objective optimization problems, so that a single solver can solve both kinds of problems.

In this section we illustrate a constraint programming based model, referred to as the CP Model, for formulating mono-objective and constraint satisfaction DSE problems. The CP Model is essentially a logic program in Minizinc [91], a *Constraint Programming Language* for specifying CSPs. Minizinc was chosen as the CPL in the framework for two main reasons. Firstly, Minizinc is an expressive CPL that supports a variety of constraints expressions (finite domain, integer,set and linear arithmetic). This makes it a good candidate for a generic framework like ours, that supports modeling of wide range of DSE problems, where the constraints can vary from simple numerical relations to complex structural constraints. Also Minizinc supports a simple mathematical notation-like syntax that simplified the translation of the IRL model including constraint definitions to a Minizinc program.

Secondly, Minizinc is a medium-level constraint programming language. A model in

Minizinc can be used by an array of solvers (Gecode [122], Eclipse [11]). This enables the modeler to solve the same Minizinc model using different solvers. Moreover, Minizinc distribution [3] comes with a set of parsers that translate the model to SAT, ILP and hybrid solvers. This support is limited because not every construct in Minizinc can be used with every solver. For example, a SAT solver will not support the decision array variables in Minizinc. Additional advantage of using Minizinc is that it supports the separation of model with the data. This enables reusing the same model for a number of problem instances.

III.6.1 The Constraint Programming (CP) Model

The CP model is formally a 4-tuple:

$$M_{CPL} = \langle D_{CPL}, C_{CPL}, S_{CPL}, Obj_{CPL} \rangle \tag{III.2}$$

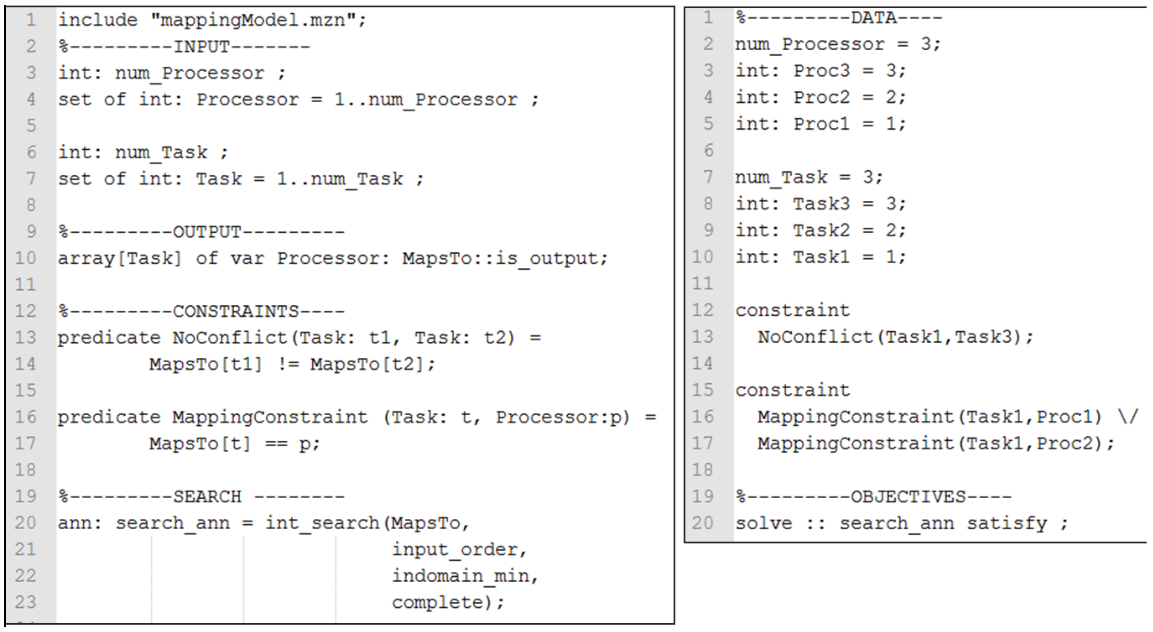
where

- D_{CPL}, is a set of declarations of decision and parameter variables. The value of a parameter variable is read from the data model and the values of decision variables are calculated by the CP solver.
- C_{CPL}, is a set of constraints. There are two kinds of constraints supported in CPL : predicates and constraints. A predicate is a function that is called in the data model.
- Search options are used to specify the search variables that are used for propagation.
- Objectives of the problem are obtained from the DSE problem model, discussed in Section III.6.3.

CP Model of the problem is automatically generated from its conceptual model MM_{CM} using the *IRL2CP* translator (Figure 3). Appendix D presents the rules for translating each concept in the IRL model, including the pCSL constraints to corresponding concepts in the CP model.

III.6.2 CP Model of the Simple Mapping Problem

Left side of Figure 27 shows an excerpt of the CP model for the simple mapping problem and is automatically generated from the IRL model of the problem (Section III.5.3) using the *IRL2CP* translator. The CP model consists of declaration of input parameters Lines (3-7) generated from *iTask* and *iProcessor* input parameters in the IRL model of the problem (Figure 25). Line (10) in the model shows the declaration of the decision variable generated corresponding the *iMapsTo* decision variable in IRL model.



```
 1  include "mappingModel.mzn";
 2  %---------INPUT-------
 3  int: num_Processor ;
 4  set of int: Processor = 1..num_Processor ;
 5
 6  int: num_Task ;
 7  set of int: Task = 1..num_Task ;
 8
 9  %---------OUTPUT---------
10  array[Task] of var Processor: MapsTo::is_output;
11
12  %---------CONSTRAINTS----
13  predicate NoConflict(Task: t1, Task: t2) =
14          MapsTo[t1] != MapsTo[t2];
15
16  predicate MappingConstraint (Task: t, Processor:p) =
17          MapsTo[t] == p;
18
19  %---------SEARCH --------
20  ann: search_ann = int_search(MapsTo,
21                               input_order,
22                               indomain_min,
23                               complete);
```

```
 1  %---------DATA----
 2  num_Processor = 3;
 3  int: Proc3 = 3;
 4  int: Proc2 = 2;
 5  int: Proc1 = 1;
 6
 7  num_Task = 3;
 8  int: Task3 = 3;
 9  int: Task2 = 2;
10  int: Task1 = 1;
11
12  constraint
13    NoConflict(Task1,Task3);
14
15  constraint
16    MappingConstraint(Task1,Proc1) \/
17    MappingConstraint(Task1,Proc2);
18
19  %---------OBJECTIVES----
20  solve :: search_ann satisfy ;
```

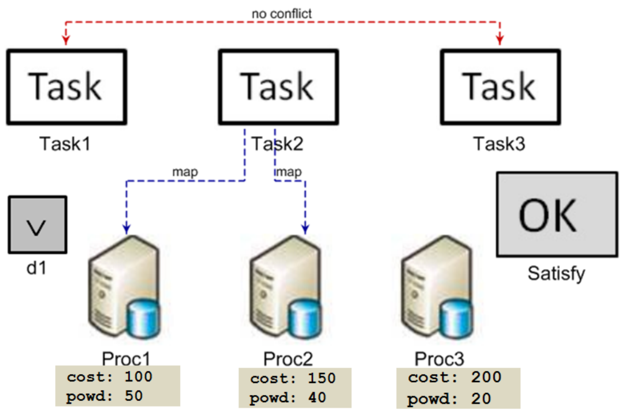
Figure 27: An excerpt of the CP Model (Minizinc) of the simple mapping problem

The search options in the CP model (Line(20)) include the search variables as well as value and variable ordering that is used to configure the search procedure of the constraint solver. All independent decision variables in the IRL model are considered as search variables and are combined to form a single array of search variables. In the simple mapping problem only independent decision variable is '`MapsTo`'. The generic ordering used to order the variables is to add functions first, followed by set variables and finally predefined variables. The value ordering of the search is to sequentially label the variable with each value from the variable's domain in an ascending order. This is represented by '`indomain_min`'. The variable ordering of the search is given by '`input_order`', which represents all variables in the search string are ordered in the order of their appearance. In this case there is one decision variable in the search string. Finally the last parameter '`complete`' is used for exhaustive search of the design space.

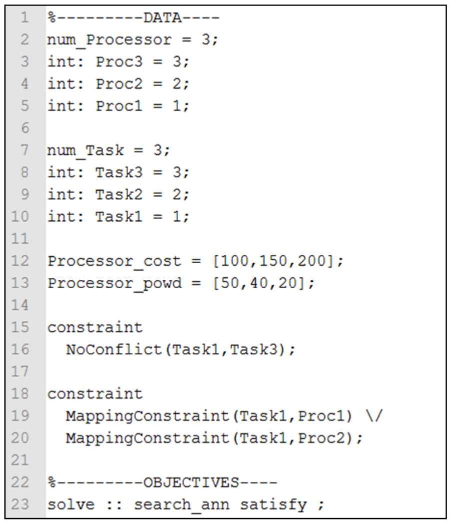
The right side of Figure 27 shows a sample data file that includes instances of the *Task* and *Processor* concepts as well as instance of the *NoConflict* constraint. This data is used by the solver along the CP Model to retrieve solutions of the problem, if any exist. The data that is used by the solver along with the CP Model is generated from a design space model M_D that conforms to the conceptual model. This data is generated using the $M2D$ translator. The details of the translation rules to generate CP model from an IRL model are included in Appendix D.

III.6.3 Model and Search using CP Model

The conceptual model of the simple mapping problem (Appendix A) is used to configure the GME to create a domain specific DSE environment, which can be used by the domain-engineers to create problem models. Figure 28(a) shows a problem model, which contains a set of 3 task objects; $\{Task1, Task2, Task3\}$, that is to mapped to 2 processor objects $Proc1$ and $Proc2$. $Task1$ and $Task3$ have a *NoConflict* constraint specifying that they cannot be mapped to the same processor. $Task2$ has two mapping constraints to $Proc1$ and $Proc2$ that are contained in the *Disjunctive Set d1*, specifying the constraint that $Task2$ can be mapped only to $\{Proc1, Proc2\}$. The objective of the exploration is to find a solution that satisfies the constraint. The data generated from this instance model is shown is Figure 28(b). This data is used in conjunction with the CP model and fed to the solver. We use Minizinc finite domain solver to retrieve possible solutions. The design space of

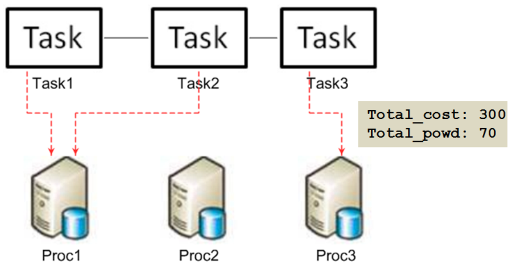


(a) Problem Model M_{CM}



```
%---------DATA----
num_Processor = 3;
int: Proc3 = 3;
int: Proc2 = 2;
int: Proc1 = 1;

num_Task = 3;
int: Task3 = 3;
int: Task2 = 2;
int: Task1 = 1;

Processor_cost = [100,150,200];
Processor_powd = [50,40,20];

constraint
  NoConflict(Task1,Task3);

constraint
  MappingConstraint(Task1,Proc1) \/
  MappingConstraint(Task1,Proc2);

%---------OBJECTIVES----
solve :: search_ann satisfy ;
```

(b) Data generated from M_{CM}



(c) Solution Model M_S

Figure 28: Mapping Problem Solution

this small example consists of 27 design points out of which 12 are valid design configurations. Figure 28(c) shows one of the valid suboptimal design configurations returned by the solver.

III.7 Multi-objective Optimization

A wide variety of DSE problems found in literature [13, 38, 45] require simultaneous optimization of more than one objectives. In these problems, a single optimal solution that is best in terms of all objectives most probably does not exist. Thus we need to find a set of solutions that trade off between the different objectives. These problems are known as multi-objective optimization problems and consist of at least two objectives. The goal of solving multi-objective optimization problems is retrieve a set of Pareto-optimal solutions (see Section II.3). A solution is Pareto-optimal if improvement of one objective will make it worse with respect to other objectives.

Mono-objective optimization techniques solve multi-objective DSE problems by using an aggregation function to combine multiple objectives into a single function. However, these techniques cannot find all Pareto-optimal solutions (refer to Section II.3). In order to overcome these limitations, in this section we present the multi-objective optimization model, referred to as the MOP model, that uses a population based Evolutionary Algorithm (EA) [12] to solve multi-objective DSE problems. The basic principle of a population-based EA is iterative improvement of a set of solutions, performed over a series of generations to reach a final set that contains Pareto optimal solutions. Contrary to the CP approach, where the multiple objective are scalarized to solve optimization problem, an EA approach uses the dominance relation (refer to Definition 1) to rank the solutions and retrieve a set of Pareto-optimal solutions in a single run. However, EA is an approximate algorithm, so it does not guarantee finding all Pareto optimal solutions but only an approximation of the Pareto front. Commonly used EA's for solving multi-objective optimization include the Non-dominated Sorting Genetic Algorithm (NSGA-II) [33] and Strength Pareto Evolutionary Algorithm (SPEA-2) [135]. In this section, we discuss how multi-objective optimization using evolutionary algorithm is supported in our framework.

III.7.1 Background

In this section, we briefly discuss the different components of an EA followed by a discussion on how we have used it in the generic framework.

III.7.1.1 Components of Population-based Evolutionary Algorithms

An evolutionary algorithm consists of the following main components [123]:

- **Solution Representation** is referred to as the *chromosome* in evolutionary algorithms and is the most important component of the algorithm. The chromosome is essentially a vector of decision variables referred to as the *genes*. Each gene can take a set of possible values called the *alleles*. The solution representation plays an important role in the effectiveness of the search algorithm. A linear encoding is a string of symbols of a given alphabet. Most common linear encodings are binary and discrete encodings. A binary encoding is usually used when the decision variables denote boolean decisions. A solution in this case will be encoded by a vector of binary variables. A discrete encoding scheme is a generalization of the binary encoding that uses an n-ary alphabet instead of binary alphabet. A decoder function $d : R \rightarrow S$ is used to retrieve the solution represented by an encoding.

- **Population Initialization**: As mentioned, evolutionary algorithm are based on iterative improvement of a set of candidate solutions (individuals). The set of individuals is referred to as the population. The initialization of the population affects the quality of solutions retrieved after a finite number of generation. The initial population should consist of diverse candidate solutions so that the search does not converge prematurely. However the most common method of population initialization is to use a pseudo-random number generator.

- **Objective Function** formulates the goal of the optimization problem. The objective function essentially provides a complete ordering to all solutions in the search space by associating a fitness value to each solution. The function thereby guides the search towards good solutions in the search space.

- **Constraint Handling** is an important component of the algorithm while solving constraint optimization problems. One of the common approaches for handling constraints is to use a penalize invalid candidate solutions by ranking them lower than the valid solutions. Another strategy of handling constraints is to repair invalid solutions to generate a valid solution.

- **Selection Strategy** is used to select candidate solutions that can be used for reproduction. A selection strategy uses the fitness value of the individuals to select the parents.

- **Reproduction Strategy** involves use of suitable mutation and recombination operators to produce a population of offspring. Mutation is a unary operator that acts on a single individual and generally involves changing the value of a gene with another value in the alphabet. The parameter p_m gives the probability to mutate each gene of the chromosome. Usually, the value is p_m is very small. Application of mutation is followed by recombination, which is a binary (nary) operator and acts on 2 (or more) individuals. The recombination operator is used to inherit characteristics of both parents while producing offspring. Most commonly used operator is a one-point crossover operator, where a crossover point k is randomly chosen and offspring are produced by interchanging segments of the parent strings. A parameter $p_c \in [0...1]$ gives the proportion of the parents on which the recombination is performed.

- **Replacement Strategy** is used to modify the old population by taking the new offspring into consideration. One possible replacement strategy is to replace the entire parent population by the offspring population. Another possible strategy is to replace the weakest parent individual with a better offspring. Usually a mixture of these replacement strategies are used.

- **Stopping Criteria** represents the end condition to be satisfied to stop the search. Usually a *static* condition like number of generations , or maximum number of objective function evaluations can be used to quit the search algorithm. It can also be *adaptive* condition, where the end of the search is not known a priori. For example, a fixed number of generations without any change in the fitness value. A more advanced criteria calculate the diversity of the population and the algorithm stops when the diversity of the population falls below a given threshold.

Our focus is on the use of evolutionary algorithm for finding Pareto-optimal solutions of multi-objective optimization problems.

III.7.1.2 Platform and Programming Language Independent Interface for Search Algorithms (PISA)

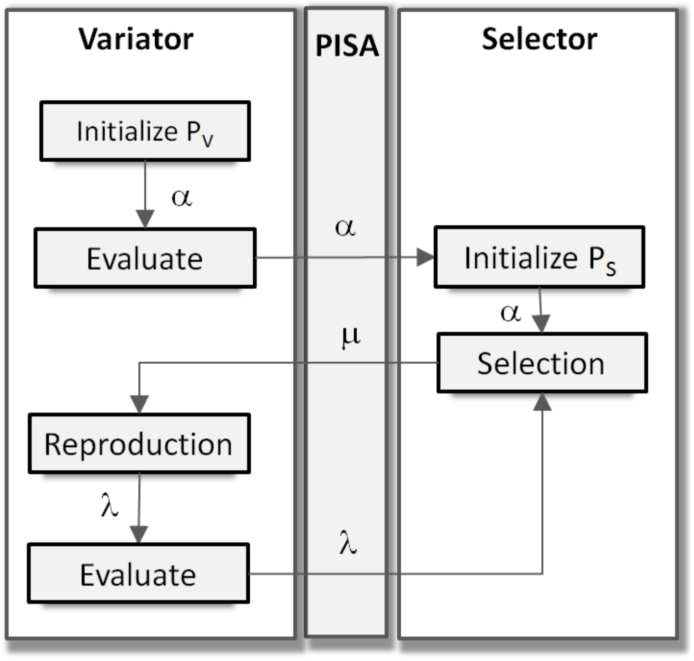


Figure 29: Platform and Programming Language Independent Interface for Search Algorithms (PISA)

Figure 29 shows the control flow in an EA. In an Evolutionary Algorithm, the search starts with initialization of a population of randomly generated solutions of size α. Every solution in the population is an encoded version of a solution of the problem, usually a vector. A evaluation function is used to associate a fitness value to each solution. This fitness value is used to select a set of μ solutions with better fitness value to act as parents, to produce λ offspring (using mutation and cross-over). The newly generated offspring are evaluated and finally, a replacement strategy is used to decide which individuals out of the parents and offspring will survive. This represents one *generation*. This process is repeated till a termination condition (for example, a constant number of iterations) is satisfied.

The solution representation, the evaluation function and the reproduction strategies are problem-dependent components of EA and have to be customized for each new problem. For example evaluation function for a Knapsack problem will be different from the evaluation function of a network processor problem. On the other hand the selection and replacement strategies are dependent only on the fitness values of the candidate solutions and not on the characteristics of the problems. Consequently, the selection and replacement strategy can be easily separated from the rest of the program and implemented as separate module that interacts with the problem specific part through an interface. The *Platform and Programming Language Independent Interface for Search Algorithms* (PISA) [21] is a text based interface that is based on this principle of separation of problem-specific and independent parts. The PISA interface is designed for interaction and synchronization between two separate modules :

Variator, which consists of all problem specific aspects of the algorithm. This includes the solution representation, the objectives and evaluation functions, mutation and recombination operators.

Selector, which consists of the selection strategy that is used to identify promising candidate solutions that can act as parents. A library of various selection algorithms, for example SPEA, NSGA-II can be found at [134]. Same Variator can use different Selector algorithms to solve the problem, thus giving the flexibility two choose an algorithm best suited for solving a given problem. The two modules can be implemented separately and interact with each other through text files. This interface allows the same problem to be solved using with different selector modules. For more details on the interface, refer to [134].

III.7.2 The Multi-Objective Optimization (MOP) Model

In our framework we use PISA interface to divide the EA into Variator and Selector modules. The Variator module is synthesized from the model M_{IRL} of the problem and the ready to use Selector modules, such as NSGA-II are used to solve the problem. Although the Variator module in our framework is implemented as a Java program, of the Variator module, it can be easily replaced by another imperative language as the interface is language-independent and interaction is done using text-files. In order to reduce the complexity of using an EA to solve a problem in the generic framework some parts of the Variator are fixed. This includes:

(1) Representation: The framework supports discrete representation of the solution, where the encoded solution (chromosome) is formed by a set of decision variables. For example, if there are two integer decision variables x and y in the problem, then the chromosome is of length 2, where each gene represents one decision variable.

(2) Population initialization: The initial population is by default done randomly. The size of the initial population is fixed by the domain engineer using a parameter, `alpha`.

(3) Reproduction Strategy: The reproduction strategy includes application of mutation and recombination operators. A mutation operator makes slight changes in the chromosome. The probability `mutation_probability` specifies the probability to mutate each gene in the chromosome. In a discrete representation, mutation of a gene would replace value associated with a decision variable by another value from

its domain. The recombination operator is a binary operator that inherits the characteristics of the two parent chromosomes to generate the offspring. The generic framework supports three strategies for recombination (cross-over) operators: one-point crossover, where a crossover point in the chromosome is selected randomly and two offspring are created by interchanging the segments of the parents. Similarly in two-point crossover, two sites are selected randomly and two offspring are generated by keeping the first and the last segments of one parent and mid segment of another parent. In uniform crossover, each gene is inherited randomly from either parents.

(4) Constraint Handling strategy: We use the penalizing strategy for handling the constraints in the constraint optimization problems.

(5) Stopping Criterion: The evolution process continues for `max_generations` and stops.

Details of the strategies are hidden from the user. The domain-engineers selects these strategies (replacement, selection, etc) using the parameters mentioned when he solves a multi-objective problem. Besides these fixed parts, the Variator also contains problem-specific parts: the decision variables and the evaluation function that are synthesized from the IRL model of the problem. This part is referred to as the *MOP model* in this thesis. In the generic framework, the Variator module and hence the MOP model are implemented as Java classes.

Formally M_{MOP} is a 5-tuple:

$$M_{MOP} = \langle P, \vec{D}, O, c(\vec{D}), e(\vec{D}, o) \rangle \tag{III.3}$$

where

- P is a set of parameters that characterize the search space of the problem rather than a solution point. The parameters remain constant for a given problem and are used in evaluation of the fitness of solution. All input parameters in the IRL model are common to all solutions in the search space and are refined to parameters in the MOP model. For example, the set of tasks in the simple mapping problem is a parameter. This is discussed in detail in Section III.7.2.1.

- $\vec{D}$ is a candidate solution of the problem described as a vector, also referred to as the *Chromosome*. Each position in the vector is derived from the decision variables of the problem. This is discussed in detail in Section III.7.2.2.

- O is a set of objective variables that are used to guide the search towards the good solutions in the search space. The MOP model is used for multi-objective optimization and therefore consists of 2 or more objective variables

- $c(\vec{D})$ is a function that returns 0 if the individual does not satisfy a certain constraint. There can be a set of functions, each corresponding to a constraint in the IRL model. This is discussed further in Section III.7.2.3.

- $e(\vec{D},o)$, is an objective function that calculates the an objective value $o \in O$ for a specific candidate solution. There is a set of objective functions, each corresponding to one objective. Each function also takes the number of constraint violations into consideration while calculating the fitness value.

The MOP model is organized into two Java classes, namely

(i) *Param Class*, a singleton that contains the problem specific parameters that are read from the data file.

(ii) *Individual Class*, that encapsulates the decision variables, chromosome, initialization and evaluation using constraint and objective functions.

III.7.2.1 MOP Parameters

In this section we discuss the synthesis of parameters from the input parameters in the IRL model. The values of the generated parameters are obtained from the problem model. The type of the parameter depends on the variable type.

Parameter Basic Types. Most imperative languages support basic variable types- `int`, `bool`, `string` and `float` for *inputs*. Therefore the basic types of IRL have a direct correspondence with the basic types in the imperative language.

Parameter Sets. Java already supports sets, so all set and compound set parameters are refined to sets in Java. Unlike the CP model where all the sets were refined to set of integers, the sets in MOP model can be of any type. For example in the simple mapping problem, the set *iProcessor* in the IRL model is refined to a string set shown in Figure 30 (line 2). This set contains the names of *Processor* objects in the problem model.

```
1  class Param{
2          HashSet<String> Task;
3          HashSet<String> Processor;
4          Function<String,Integer> Processor_cost = null;
5          Function<String,Integer> Processor_powd_rate = null;
6  }
```

Figure 30: Parameters: Mapping Problem MOP

Parameter Functions. We have built a template Java Class for *Function* data type. This enables simple translation of input variables in the IRL model to objects of corresponding Java Classes. The *Function* Java class consists of three data members: (1) a set *D*, representing the domain, (2) a set *R* representing the range of the function set, and (3) *f*, a map with keys from the domain and values from the range set. Methods of the Function class correspond to the operators included in pCSL constraint expressions. For example, '`irl_F(d,r)`' in pCSL expression is rewritten as '`F.isMember(d,r)`', where `isMember` is a method used to check for the membership of a pair $\langle d, r \rangle$ in the function F. Figure 30 (line 3) shows the function variable synthesized from *Processor_cost* parameter function in the IRL model of the simple mapping problem.

Parameter Relation. Like functions, we created a template Java Class for *Binary Relations*, consisting of data members and methods. It consists of three data members, two sets S_1 and S_2, and r, a map associating elements of S_1 and S_2. Methods of the class are operations that can be performed on a relation and are similar to the function operations.

All the input variables in the IRL model are synthesized to parameters in the MOP model. The value of these parameters, that is the elements of the sets, functions and relations are obtained from the design space instance.

III.7.2.2 MOP Chromosome

Each candidate solution in the population is modeled as a vector (chromosome) $\vec{D}$, where vector components (genes) might represent a decision variable. We used a discrete vector chromosome to represent the solution to a problem, where each gene i in the chromosome is a value in the range (allele) $[min_i \dots max_i]$. The min_i and max_i depend on the domain of the decision variable that it represents . In this section we discuss the synthesis of the chromosome from the independent decision variables in the IRL.

Decision Basic Types. We restrict decision variables to integers and booleans. They are translated to a single gene in the chromosome. The allele in case of a boolean variable is `[0..1]`.

Decision Sets. The set decision variables in the IRL are integer interval sets within the range `[1...max_cardinality]`. This set is translated to a single gene that can take values in the range `[min...max]`.

Decision Functions.A function $F : A \rightarrow B$ is represented by a set of genes $[x \dots (x + |A|)]$

```
class Individual{
//independent decision variables
   DecisionFunction<String,String> MapsTo = new DecisionFunction<String,
      String>(Node, Processor);
   int MapsTo_offset = 0;
   int MapsTo_length = this.MapsTo.card();
   int MapsTo_allele = this.MapsTo.getRange().size();
//candidate solution
   int[] Chromosome;
   int Chromosome_length = mopMapsTo_length;

public Individual(){
  this.Chromosome = new int[this.Chromosome_length];
  for (int i = 0; i < this.Chromosome_length; ++i) {
     if(i >= MapsTo_offset && i < MapsTo_offset + this.MapsTo.card()){
         this.Chromosome[i] = randomGenerator.nextInt(MapsTo_allele);
     }
 }
 this.objectiveSpace = new double[dim];
 this.eval();
}
}
```

Figure 31: Chromosome: Mapping Problem MOP

in the chromosome, where x is the offset of the decision variable F, such that

$$\forall i \in [x \dots (x + |A|)] \cdot \exists j \in [1..|B|] \cdot D[i] = j$$

where D is the chromosome. For this condition to hold, both A and B require an index map to associate map each element to a unique index number. For example set $iTask = \{t1, t2\}$ is indexed, such that $t1 = 1$ and $t2 = 2$. This indexing is implemented in the *DecisionFunction* class that extends the *Function* class and initializes the index maps for domain and range sets. Figure 31 shows the declaration of *MapsTo* decision function in the MOP model. The variable *MapsTo_allele* takes the maximum value that each gene representing the function can take. The mapping problem has a single independent decision variable *iMapsTo* (see Figure 25(b)). Therefore, the length of the chromosome is given by the cardinality of the function, which is equal to $|iNode|$, since all functions in IRL are total functions. The offset of the decision variable is `0`. Similarly, a function $F_2 : A \to \mathscr{P}(B)$ is refined to a 1-D array of length $|A| \times |B|$ where

$$\forall i \in [1..|A| \times |B|],\ \ D[i] \in \{0,1\}$$

Decision Relations. A relation $R \subseteq A \times B$ is refined to a 1-D boolean array of size $|A| \times |B|$. The allele of each gene is `[0..1]`.

The chromosome is intialized by randomly choosing a value for each gene from *Processor*, range set. The chromosome is formed by the combination of all the independent output variables. The position of each decision variable in the chromosome is captured using an index offset. The initialization of the chromosome is done by choosing a random value for each gene in the chromosome from the domain.

III.7.2.3 MOP Constraints

The generic framework uses penalizing strategy to handle constraints. This means that the solutions that violate the constraints have a lower fitness value that the valid ones. Also the invalid solutions are ranked in proportion to the constraint violation, which implies that solutions with less number of constraints violations is ranked better than solutions with more number of constraint violations. Each objective function $f(x)$ is extended to include the number of constraint violation into account, such that .

$$f_p(x) = f(x) + \sum_{i \in C} w_i \alpha_i \qquad \text{(III.4)}$$

where $\alpha_i = 1$ if the constraint i is violated and $\alpha_i = 0$ if the constraint is satisfied by the solution x. A constraints in the MOP model is a function with boolean return value. This is the most generic way of handling constraint optimization problems. A better approach of handling constraints is repairing an invalid candidate solution. A repair operation can take an arbitrary invalid solution and modify it to yield a valid solution. In a generic framework like ours, the problems can have constraints that are too complex and determining how to modify a solution to satisfy these constraints can be challenge.

```
public class NoConflict implements Predicate{
   String _t1;
   String _t2;
   public mop_NoConflict(String a, String b){
        this._t1 = a;
        this._t2 = b;
   }
   public boolean call(Individual ind){
     String s1 = ind.MapsTo.getImage(_t1);
     String s1 = ind.MapsTo.getImage(_t2);
     if( ! s1.equals(s2)){
       return true;
     }
     return false;
   }
}
```

Figure 32: Predicate: Mapping Problem MOP model

A constraint class in the IRL model models an invariant condition that must be satisfied by all valid design solutions. Each constraint class is refined to a method with a boolean return value. A predicate class in the IRL model models a template condition that can be instantiated in the design space instance. Each predicate class in refined to a separate Java class. Figure 32 shows the class corresponding to the *iNoConflict* predicate (Figure 26) in the IRL model. The class *NoConflict* extends the base class *mop_Predicate*. The base class contains a *call* method that takes the chromosome as an argument and returns a boolean value. This method is overwritten to include the definition of the predicate. The function returns false if the constraint is violated and true if the constraint is satisfied by the current solution. The arguments of the predicate in IRL model are translated to data members of the derived class. The objects of this predicate class are obtained from the design space instance. Although it is possible to implement the predicates as methods, the class approach maintains a clear distinction between the constraints and predicates. This is required while retrieving relevant data from the design space instance. A set of simple rules are used to translation the constraint definition in pCSL to imperative functions. The methods supported by the *Function* and *Relation* Java classes are similar to the operations on the same supported by pCSL making the translation straight forward. Figure 32 shows the imperative function corresponding the pCSL constraint definition in IRL, where the boolean expression with '`iMapsTo(ctx)`' is translated to '`MapsTo.getImage(ctx)`'. A subset of the rules are included in Appendix E for reference.

III.7.2.4 MOP Functions

All output variables are data members of the *Individual* class. The independent output variables are refined to a Chromosome representation, while the dependent output variables are calculated using the assignment statements or function definitions that specify how the values of these variables depend on other variables. Assignment statements and function definitions are refined to methods of the *Individual* class. For example, the function definition of the '`Processor_isUsed`' function in the simple mapping problem is parsed to a set of functions shown in Figure 33.

III.7.3 Model and Search using the MOP Model

The problem model presented in Figure 28 is run with an initial population of size 10, a mutation probability of 0.1, a recombination probability of 0.1 and maximum generation of 40. We use the simple evolutionary multi-objective optimization algorithm for selection. The solutions obtained are shown in Figure 34. The invalid solutions obtained have high cost and power dissipation due to the constraint violation penalty function used to penalize the objective values. We obtain two non-dominated valid solutions.

Solution A with `total cost= 350` and `total powd = 60` corresponds to the mapping

```
public void processor_used_assign_stmt(){
  //Processor_used(ctx) <-> (card(MapsTo(~ctx) >= 1)
  for(String ctx: Variator.population.nfParams.Processor){
    boolean bexpr = processor_used_assign_stmt_bexpr_1(ctx);
    this.Processor_used.add(ctx, bexpr);
  }
}
public boolean processor_used_assign_stmt_bexpr_1(String ctx){
  int aexpr1 = processor_used_assign_stmt_aexpr_1(ctx);
  int aexpr2 = 1;
  if(aexpr1 >= aexpr2){
    return true;
  }else{
    return false;
  }
}
public int processor_used_assign_stmt_aexpr_1(String ctx){
  HashSet<String> D1 = MapsTo.getCoImage(ctx);
  return D1.size();
}
```

Figure 33: Assignment Statement: Mapping Problem MOP model

`T1->p2, T2->p1, T3->p1`

Solution B with `total cost= 250` and `total powd = 90` corresponds to the mapping

`T1->p2, T2->p3, T3->p3`

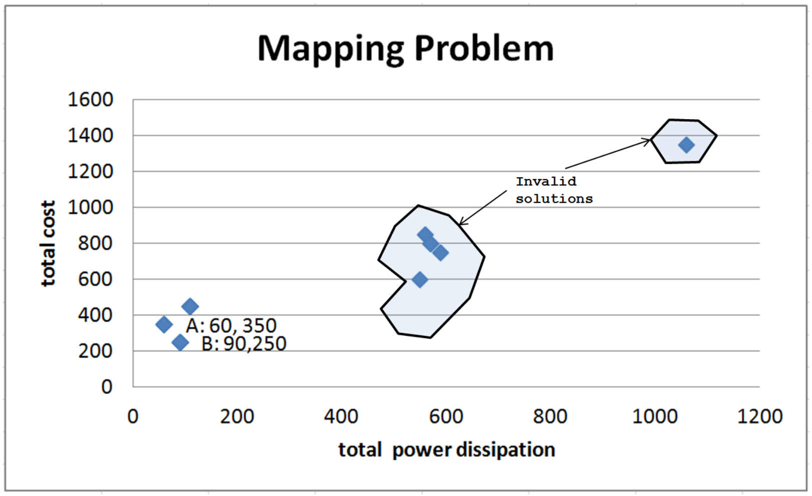


Figure 34: Mapping Problem with multi-objectives solution

CHAPTER IV

EVALUATION OF THE FRAMEWORK

Our aim in this section is to evaluate the generic framework based on the following hypothesis

- The reusable classes (ADSEL) provide an abstraction level that allows us to easily model DSE problems that cover a wide range of problems from different domains. Moreover, once created, the models of the problem are easy to modify.

- An IRL model for a class of DSE problems can be translated to alternative solver-specific models, thus enabling flexibility to solve both constraint satisfaction and optimization problems. The generated models (CP and MOP) are syntactically and semantically correct and provide correct results.

The first challenge in evaluation of our framework was to identify a set of problems that are diverse, not only in terms of the application areas but also in terms of the objectives. Even though a large number of DSE case studies exist in literature, these case studies have mainly been used for evaluation of domain-specific DSE approaches. Similarly, a number of established benchmarks exist but they focus on a given application domain [48]. Literature survey reveals that similar problems can exist in different application areas, for example resource allocation problems exist in network design, embedded systems etc. Therefore, we attempt to categorize the commonly found DSE problems based on structure and intent of exploration rather than the application area. Towards this goal, we present 6 categories of DSE problems: resource allocation, selection, placement, routing, scheduling and configuration. Existing DSE case studies found in literature are variants of these classes, or a combination of them. This classification is along the lines of the CSP classification proposed by Miguel et. al.[1]. Here we briefly discuss each of the 6 problem categories.

1. **Resource allocation problems**: Given a set of T objects and R resources, the goal of a resource allocation problem is to allocate resources to the objects, such that all constraints (memory, cpu, etc) are satisfied. A number of variations of this problem are found in literature [38, 53, 112].

2. **Selection problems**: Given a set of objects S, the goal of a selection problem is to a find a subset $K \subseteq S$, such that a set of constraints (for example $memory \leq 100$) are satisfied and a cost function (for example cost, memory consumption etc.) is optimized. Selection problems have been commonly found in software-product line engineering. [19, 130]

3. **Placement problems**: Given a set of two-dimensional objects and a plane, the placement problem is to assign a position to each object, such that all objects lie within the given boundary of the plane and the objects do not overlap. In a variant of the placement problem, the plane is not given and the objective is to minimize the area of the enclosing box containing all objects. Placement problems have been found most often in VLSI design [105].

4. **Routing problems**: Given a graph $G = (V, E)$, where V is a set of nodes, E is a set of non-negative edges, and a set of terminals $T \subseteq V$, the routing problem is to find the a tree connecting the nodes in T, such that the tree is optimal with respect to an objective. These problems are commonly found in wire routing in VLSI [104], network design [29] etc.

5. **Scheduling problems**: Scheduling problems are characterized by assigning start times to a series of tasks that have to be performed by some deadline with the possibility of precedence constraints between them [73, 99].

6. **Configuration problems**: A configuration problem involves creation of a relationship between decision variables and their domain values subject to additional constraints. Besides this, we consider all unconstrained optimization problems to be configuration problems [119]. This class also includes optimization problems that do not have an analytical function to calculate the cost of a design solution.

We use the DSE classification to evaluate the scope of the framework. We evaluate the framework based on the following criteria:

Scope: This is the most important criteria for evaluating the framework. We select a set of 5 problems, covering the following categories: resource allocation, construction, placements, routing, and scheduling. Then, we attempt to model these problems using the framework. Section IV.1-Section IV.5 present the modeling and solution of each of the problems. Each section consists of two steps: (1) *Configuration*, which is to be performed by the domain expert. This consists of creating a conceptual model of the problem and writing CSL constraints, and (2) *Model and search*, which consists of creating a DSE problem model and performing search to retrieve solutions.

The configuration category includes class of problems that do not have an analytical optimization model to calculate the cost of a design point in the design space. An example class of problem in this category is the cache configuration problems [133], which has been solved using the simulation-based methods. In order to solve these problems using our framework, a standard interface is required to integrate the simulation tools with the framework, so that the cost estimation using the simulation tools can be synchronized with the search engine. We consider this task in our future work.

Solver Support: Can the same problem model M_{CM} be used for constraint satisfaction, mono and multi-objective optimization? In Section IV.6, we evaluate the solver flexibility of our framework by generating a CP model as well as an MOP model from a complex resource allocation problem and then use this model to retrieve Pareto-optimal solutions.

IV.1 Resource Allocation Problems

A resource allocation problem is a common DSE problem found in most often the embedded systems domain. Formally, an allocation problem is a three tuple

$$\langle N, R, A, C \rangle$$

where N is a set of components with resource requirements , R is a set of resources which are allocated to the nodes, $A \subseteq N \times R$ is the allocation relation, and C is a set of constraints that A should satisfy. Different resource allocation problems found in the literature can be classified into two subcategories based on the allocation relation:

- A **Relational Allocation Problem** has a *many-to-many* relation between components and resources, such that a resource can be allocated to many components and a component requires more than one resource. The SONET (Synchronous Optical Network) [7] design is an example of a relational resource allocation problem. The goal of the SONET problem is to allocate a set of nodes to one or more optical network rings.

- A **Functional Allocation Problem** is an allocation problem where *every* component requires exactly one resource, such that for every $c \in C$ there exists an $r \in R$ such that $\langle c, r \rangle \in A$. Each node can requires only one resource, but more than one nodes can require the same resource. The mapping problem in Erbas et al.[38], FPGA design problem in Sarawat [104] are both cases of functional resource allocation problems. The simple mapping problem used as a running example in Chapter III is also an example of a functional allocation problem. Functional allocation problems can differ depending on the properties of the allocation function (surjective, injective etc.).

In this section, we use a simplified version of the SONET design problem in [7] to evaluate the framework. A SONET communication network consists of a set of rings, each

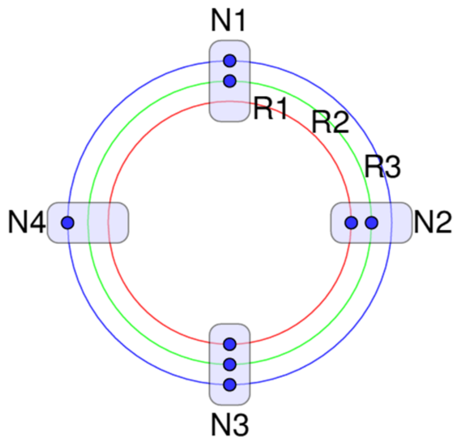


Figure 35: SONET Problem [115]

ring joining a number of nodes, shown in Figure 35. The placement of a node on a ring requires an expensive equipment called an add-drop multiplexer (ADM). A node can be installed on a number of rings. However, there is an upper bound on the number of nodes that can be placed on a ring. Two nodes can communication with each other only if they are installed on the same ring. The objective is to find a placement of nodes on rings, such that the communication demands are met and the number of ADMs used is minimum.

Problem Specification

Inputs of a SONET problem include an undirected demand graph $G = \langle N, Edge \rangle$, where N is a set of nodes, and $Edge$ is a set of communication links between the pairs of nodes. Each communication link $(u, v) \in Edge$ corresponds to traffic demand of a pair u and v nodes.

Output of a problem includes the results required from the search. In the SONET problem, the search should return a solution model M_S, with a set of rings R and an assignment of rings to nodes, $Ring\text{-}Node{:}R \times N$, where R. This reflects the relational nature of the problem.

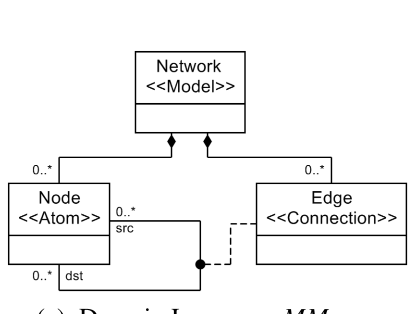
The ***constraints*** included in our version of the SONET are as follows :

C1: at most r rings are used;

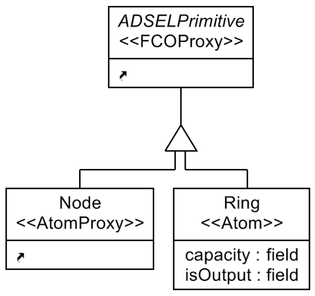
C2: every pair of nodes $n_i, n_j \in N$, such that $(n_i, n_j) \in Edge$ is mapped to at least one common ring;

C3: at most a ADMs on each ring;

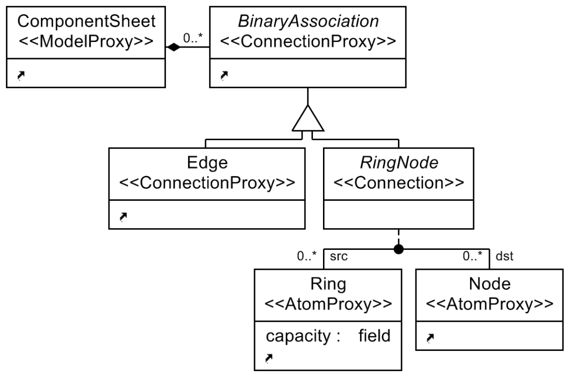
The ***objective*** is to minimize the total number of ADMs used is minimized.



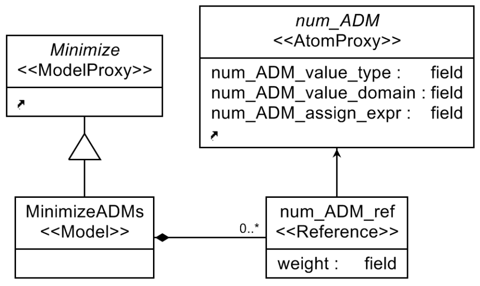
(a) Domain Language MM_{DSML}



(b) eSONET Components



(c) eSONET Association



(d) eSONET Objective

Figure 36: eSONET Conceptual Model

IV.1.1 Configuration

Based on the informal problem specification the SONET problem, the domain expert creates a conceptual model MM_{CM} of the problem by associating DSE aspects with domain-specific modeling concepts. The domain-specific modeling language (DSML) for networks, captured as a metamodel MM_{DSML} (shown in Figure 36a) consists of *Node*s with communication *Edge*s. The classes in MM_{DSML} are composed with the ADSEL classes to capture the different aspects of the DSE problem. The resulting conceptual model consists of components, associations, constraints, objectives and global variables for modeling a class of SONET problems.

IV.1.1.1 Conceptual Model

Components: The conceptual model MM_{CM}, includes the *Node* concept of the domain language, that plays the role of a *Primitive* component, as shown in Figure 36(b). A new *Primitive* component, *Ring* is introduced in the conceptual model to model the rings in the network topology. The *Ring* component is an output component (‘`isOutput = true`’), where the number of instances of *Ring* is determined by the search. The `capacity` property of the *Ring* component models the maximum number of ADMs that can be installed on a *Ring* object.

Associations: The conceptual model MM_{CM} includes a binary association *RingNode*, which models the allocation relation between *Ring* and *Node* components (shown in Figure 36c). The cardinalities of the association reflect the fact that more than one *Node* object can be associated to one *Ring* object using the *RingNode* association, and vice-versa. As the *RingNode* association models an output relation, it is modeled as an *abstract* binary association. The *Edge* communication link in the domain language M_{DSML} is included in the conceptual model MM_{CM} and plays the role of a binary association.

Global Variables: The objective of the problem is to minimize the number of ADMs in the topology, shown in Figure 36d. The total number of ADMs is modeled using a global variable *num_ADMS*, which has an integer value type and an assignment statement

```
context: Num_ADM
assignment statement:   $self = card(RingNode)
```

The objective of the exploration is modeled as a minimize class *MinimizeADM* with the *num_ADM* as an argument with `weight = 1`.

Constraints:

(C1) Constraint *C1* specifies that at most *r* rings are used. This constraint is captured using the '`cardinality`' property of the *Ring* component.

(C2) Constraint *C2* specifies that every pair of node connected by an edge should be mapped to at least one common ring. We use an *abstract* binary constraint *edgeCnstr* to capture this constraint (Figure 37) as the constraint is applicable to all instances of the edge.

The constraint definition is formally specified as:

$$\forall\{n_1,n_2\} \in Edge:\ RingNode(\_,n_1) \cap RingNode(\_,n_2) \neq \phi \qquad \text{(IV.1)}$$

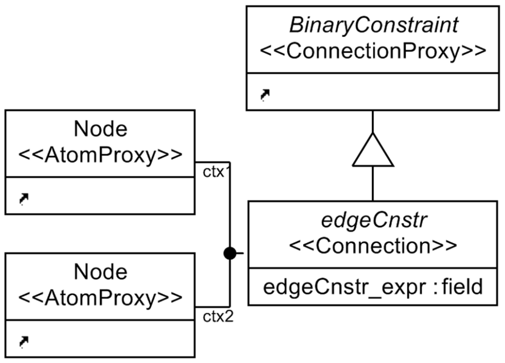


Figure 37: Edge Constraint in the SONET Problem

```
context:$self= edgeCnstr, $ctx1=Node, $ctx2=Node
CSL: forall( <$ctx1, $ctx2> in Edge )
     (
        RingNode(_, $ctx1) intersect RingNode(_, $ctx2)
         sneq {}
     )
```

Figure 38: Edge Constraint CSL Expression

The CSL constraint expression corresponding to Eq. IV.1 is shown in Figure 40. The contexts '$ctx1', '$ctx2' are combined in a tuple to iterate over the 'Edge' association domain. The constraint applies to all pairs of *Node* objects that are associated to each other by the *Edge* association. The CSL constraint expression has backward navigation along the *RingNode* association to retrieve all the *Ring* objects associated to a given *Node* object by *RingNode* association. The constraint specifies that two *Node* objects should be associated to at least one common *Ring* object using the *RingNode* association, if they are also associated using the *Edge* association.

(C3) Constraint *C3* specifies each ring can have at most a ADMS. In this simplified model number of ADMs is equal to the number of nodes on the ring. This constraint can be modeled as an abstract unary constraint *capacityCnstr*. Formally, the constraint definition states:

$$\forall r \in Ring : |(RingNode(r,\_)| \leq capacity(r) \qquad \text{(IV.2)}$$

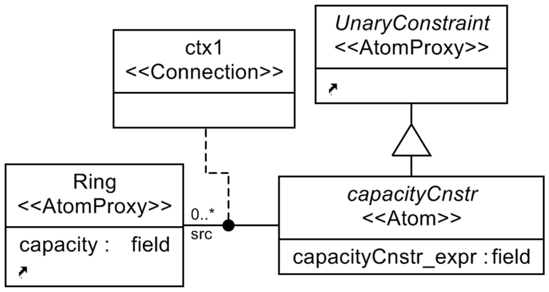


Figure 39: Capacity Constraint in SONET Problem

```
context:$self= capacityCnstr, $ctx1=Ring
CSL: forall($ctx1)
     (
        card (RingNode(_, $ctx1) ) <= $ctx1.capacity
     )
```

Figure 40: Capacity Constraint CSL Expression

IV.1.2 Intermediate Model

The conceptual model of the SONET problem MM_{CM} is translated to a model M_{IRL} in IRL. This translation is done automatically using the *CM2IRL* translator. Figure 41 shows parts of the IRL model. The Inputs comprise of three ***input parameters***:

(iV1) A set of integers, *iNode* that is created by refinement of the *Node* component in the conceptual model

(iV2) A relation, $iEdge \subseteq iNode \times iNode$ that is is a created by refinement of the *Edge* binary association

(iV3) An integer variable *iRing_capacity* that is created by refining the `capacity` property of the *Ring* component. As detailed in Section C.1, the properties of an output component are refined to variables of predefined types.

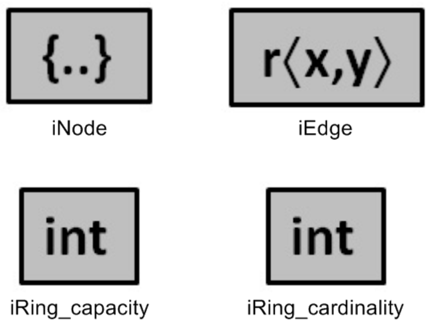
(iV4) An integer variable *iRing_cardinality* that is created because the *Ring* is an output component and the maximum number of instances of an output component is captured by '`cardinality`' property common to ADSEL components.

The outputs consists of three ***decision variables***:

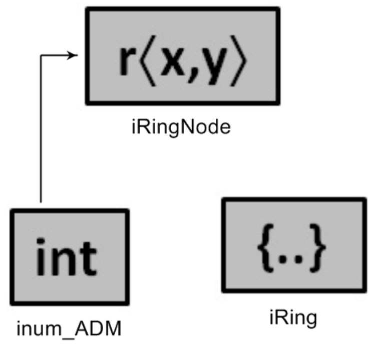
(iV5) a relation *iRingNode*, which is a refinement of the *RingNode* binary association

(iV6) an interval set *iRing*, which is a refinement of the *Ring* output component

(iV7) an integer variable *inum_ADM*, refinement of the global variable *num_ADM*. We use the assignment statement of *inum_ADM* to infer its dependence on the *iRingNode* relation. There are two independent output variables. The default variable ordering (Section II.1.1) that is followed while searching for a solution is to first assign values to compound variables (functions, relations), followed by sets before assigning values to primitive variables.



(a) eSONET IRL Parameters



(b) eSONET IRL Decision variables

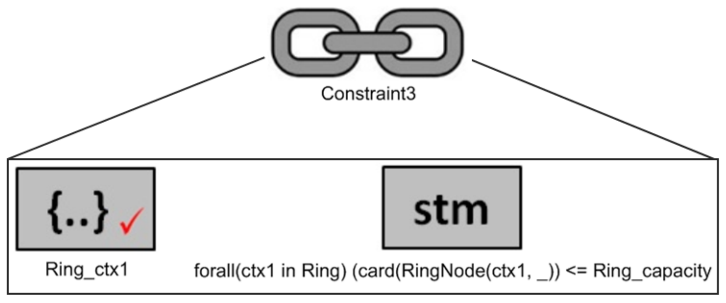

Constraint3
{..} ✓
Ring_ctx1
stm
forall(ctx1 in Ring) (card(RingNode(ctx1, _)) <= Ring_capacity
(c) eSONET IRL Constraints

Figure 41: eSONET IRL elements

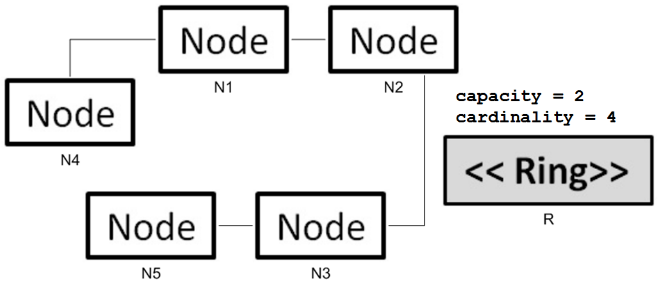
The ***constraints*** in the IRL model generated from the constraints in the conceptual model using the *CM2IRL* translator. This translator also includes a constraint rewriting engine that processes the CSL constraint expressions and rewrites them in terms of the parameter inputs and decision variables in the IRL, referred to as the processed CSL (pCSL) constraints. This constraint is in addition to the constraints generated by the assignment statement of `num_ADM`. For example, constraint $C3$ in the conceptual model is rewritten as a pCSL constraint shown in Figure 42

```
pCSL: forall(c in iRing)
      (
         card (iRingNode(_, c) ) <= c.capacity
      )
```

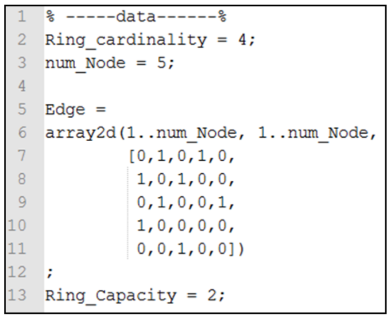
Figure 42: Capacity Constraint pCSL Expression

IV.1.3 Model and Search

The CP Model of the SONET problem, given in Appendix F is auto-generated from the IRL model of the problem. Figure 43(a) shows a sample design space model M_{CM} that conforms to the conceptual model MM_{CM} of the problem. M_D consists of 5 nodes with connections. A singleton *Ring* object is used to set the properties of Ring, for example here `cardinality = 4` and `capacity = 2`. The data generated from this model is shown in Figure 43(b). The data and CP model are processed by the solver (Flatzinc FD) to generate a solution, shown in Figure 43(c), showing a pair of nodes mapped to one ring satisfying the capacity constraint of the rings. For the given example, a hand written code takes 42 milliseconds to come up with the answer whereas the auto-generated code takes 80 milliseconds for the same. The main reason for a slower time comes from set based encoding of the *Constraint 2*.

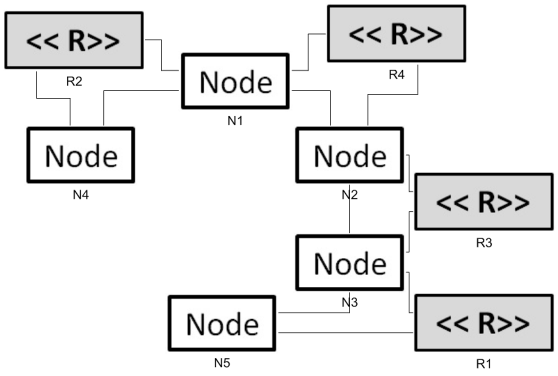


(a) SONET design space instance



```
% -----data------%
Ring_cardinality = 4;
num_Node = 5;

Edge =
array2d(1..num_Node, 1..num_Node,
          [0,1,0,1,0,
           1,0,1,0,0,
           0,1,0,0,1,
           1,0,0,0,0,
           0,0,1,0,0])
;
Ring_Capacity = 2;
```

(b) Model Data



(c) SONET Solution

Figure 43: eSONET Instance Model

IV.2 Selection Problems

Given a set of objects S, the selection problem is to a find a subset $K \subseteq S$, such that a set of constraints is satisfied and an objective function is optimized. These problems are often found in software-product line engineering, where the set of features are organized in a tree-like structure called the *feature model* [62]. Figure 44 shows the feature model of a face recognition sytem adapted from [131]. A feature model consists of primitive features, modeled as the leaves of the feature model, and composite features that are modeled as internal nodes (including root) of the feature model.

Formally, a feature selection problem in software-product line engineering can be defined as a five-tuple $CP = \langle F, C, D, L \rangle$, where

- F is a set of features, such that $F = F_l \cup F_c$, where F_l is a set of primitive features, and

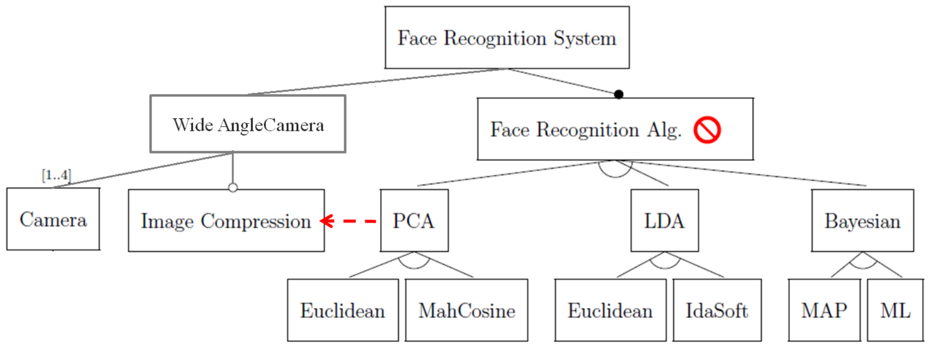


Figure 44: Feature Model(Adapted from [131])

F_c is a set of composite features. Each feature $i \in F$ has a requirement req_{ij} which represents the amount of resource i required by feature j.

- C is a set of composition constraints, which represent the relationship between a composite feature $f_c \in F_c$ and any other feature $f \in F$.
- D is a set of dependency constraints between features. For example A *requires* B, which means the selection of A implies the selection of B.
- L is a set of bound constraints on the resource requirements of a feature. For example, $A.memory \leq 100$.

The solution of a feature selection problem is a subset of the features (configuration) that satisfy all the constraints and optimize the total resources required by the product. In this section, we model and solve the feature selection problems to evaluate the scope of our framework in handling selection problems.

IV.2.1 Configuration

Based on the informal problem specification a feature selection problem, the domain expert creates a conceptual model MM_{CM} representing the class of problem by associating DSE aspects with domain-specific modeling concepts. The domain-specific modeling

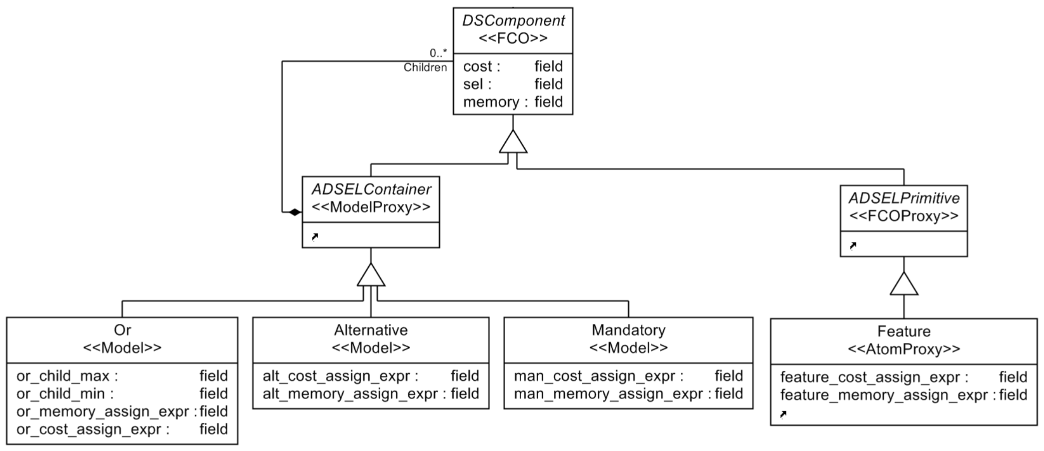

DSComponent
<<FCO>>
cost : field
sel : field
memory : field
0..*
Children
ADSELContainer
<<ModelProxy>>
ADSELPrimitive
<<FCOProxy>>
Or
<<Model>>
or_child_max : field
or_child_min : field
or_memory_assign_expr : field
or_cost_assign_expr : field
Alternative
<<Model>>
alt_cost_assign_expr : field
alt_memory_assign_expr : field
Mandatory
<<Model>>
man_cost_assign_expr : field
man_memory_assign_expr : field
Feature
<<AtomProxy>>
feature_cost_assign_expr : field
feature_memory_assign_expr : field

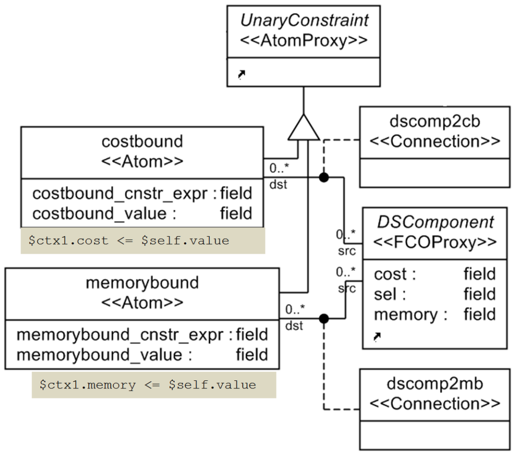
Figure 45: Conceptual Model Components

language (DSML), captured as a metamodel MM_{DSML} consists of a *System* containing *Features*. The classes in MM_{DSML} are composed with the ADSEL classes to capture the different aspects of the feature selection problem. The resulting conceptual model consists of components, associations, constraints, objectives and global variables for modeling a class of feature selection problems.

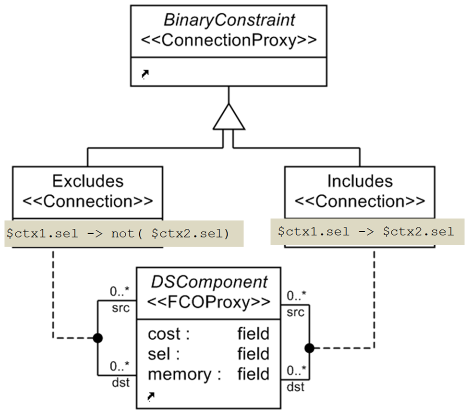
Components. The *Feature* class in the DSML is included in the conceptual model as a *Primitive* Component. A set of new components are introduced in the conceptual model to enable modeling of the feature model. We model them as a set of *Container* components in the conceptual model, where each component represents a unique composition relationship. Figure 45 shows the new components: (1) *Mandatory*; (2) *Alternative*; and (3) *Or*, that are inherited from the *ADSEL Container* component. Instead of creating new components, we can also use the specialized container classes mentioned in Section III.3.

All features (both composite and primitive) are *DSComponent*s, which has three properties: (1) `select`, a boolean valued *decision property* that models the inclusion or exclusion of the feature in a configuration, (2) `cost`, an integer valued metric property that models the cost requirements of the feature, and finally (3) `memory`, an integer valued metric property that models the memory requirements of the feature. As mentioned in Section III.3.2.3, all properties of the base are inherited by all derived classes. In this case, the `sel`, `cost` and `memory` properties of *DSComponent* is inherited by the primitive and container classes.

Associations. The conceptual model includes a single association, a composition relation *Children* modeling the fact that a *Container* component can contain other *Container* and *Primitive* features. A forward navigation of the *Children* association denotes by '`Children(`**`$self,_`**` )`' returns a set of child features of the component.



(a) eFeature Resource bound Constraints



(b) eFeature Cross Tree Constraints

Figure 46: eFeature Conceptual Model

Assignment Statements The `cost` and `memory` metric properties of a *Container A* are calculated based on the property values of the contained features *Children(A)* . For example, the assignment statements for cost and memory properties of an *Alternative* component is given by CSL assignment statements shown in Figure 47.

```
context: $self = Alternative
CSL:
$self.cost = sum ( s in Children( $self,_) )(s.select*s.cost)

$self.memory = sum ( s in Children( $self,_) )(s.select*s.memory)
```

Figure 47: Assignment Statement for metric properties of Alternative Component

The assignment statement for the *Or* component is the same as that of the *Alternative* component. The assignment statements for cost and memory properties of a *Mandatory* component is given by assignment statements shown in Figure 48

```
context: $self = Mandatory
CSL:
$self.cost = sum (  s  in  Children( $self,_)  )(s.cost)

$self.memory = sum (  s  in  Children( $self,_)  )(s.memory)
```

Figure 48: Assignment Statement for metric properties of Mandatory Component

The assignment statement for feature components is given by a value input by the domain engineer. In this case the inherited property is constrained to a single value using the following assignment statement shown in Figure 49. The **param** operator is used to constrain a decision property to a single value input by the domain-engineer.

```
context: $self = Feature
CSL:
$self.cost = param ( $self.cost)

$self.memory = param ( $self.memory)
```

Figure 49: Assignment Statement for metric properties of Feature Component

Constraints There are three kind of constraints in the conceptual model: composition constraints, dependency constraints and bound constraints.

Composition constraints, specify the constraint on selection of a child features based on the inclusion/exclusion of the parent component. The composite features *Mandatory*, *Alternative* and *Or* have different composition relations with the contained features. For example, if an *Or Component o* is selected in a configuration, then the number of child features selected is within [*Child_min(o)...Child_max(o)*]. Figure 50 shows the CSL expression for the composition constraint for *Or* component.

```
context: $self = OrCnstr, $ctx1 = Or
CSL:

forall( $ctx1)
(
       $ctx1.child_min <= sum ( s in Children($self,_) )
                              ( bool2int(s.select) )
  /\
      $ctx1.child_max >= sum ( s in Children($self,_) )
                              ( bool2int(s.select) )
)
```

Figure 50: Compostion constraint for Or Component

Similar composition constraints exist for *Alternative* and *Mandatory* components as well. For example, if an *Alternative a* is included in a configuration then exactly one of its child features is included in the configuration. Similarly, if a *Mandatory m* feature is included in a configuration, then all its children are selected.

Dependency constraints: The conceptual model includes two types of dependency constraints: *Excludes* and *Includes* that are used to model constraint between two feature in the feature diagram. Both are modeled using binary constraint class where the source and destination are *DSComponents*. The *Includes* constraint specifies that the target component is selected if the source component is selected. On the other hand the *Excludes* constraint specifies that the target component is excluded if the source component is selected. Figure 51 shows the CSL expression for an *Includes* dependency constraint, where '`$ctx1`' refers to the source and '`$ctx2`' refers to the target components.

Resource Bound constraints specify upper bounds on the resource requirements of features.

```
context: $self = $ctx1 = DSComponent, $ctx2= DSComponent
CSL:

$ctx1.select -> $ctx2.select
```

Figure 51: Includes constraint

These constraints are modeled using concrete *Unary Constraint* class with the CSL expression, '`$ctx1. <resource> <= $self.value`', where the `<resource>` is either `cost` or `memory`.

Root constraint specifies that the root component of the feature model is always selected. This constraint is modeled using a concrete *Unary Constraint* with CSL expression,

'`$ctx1.select <-> true`'

.

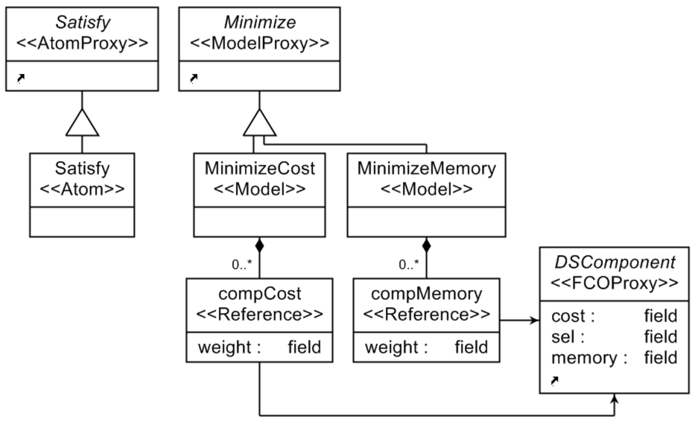


Figure 52: eFeature Objectives

Objective There are three kinds of objectives for solving feature selection problems, shown in Figure 52. The *Satisfy* objective is used to retrieve all valid feature model configurations that satisfy the different constraints. On the other hand, the domain engineer may want a configuration that minimizes cost or memory requirement of the a particular composite feature or the entire system. This is done by the *Minimize* objectives. In this problem we use the objective variables *compCost* and *compMemory* to refer to the `cost` and `memory` properties of a particular component instance, typically the root instance. The domain engineer can change the exploration objective during design time.

IV.2.2 Intermediate Model

The conceptual model is translated to a model in the IRL, which is essentially a tuple $IM_c = \langle I_c, O_c, C_c, Obj_c \rangle$, where I_c is a set of inputs, O_c is a set of outputs and C_c is a set of constraints and Obj_c is a set of objective variables. We discuss each in detail:

Inputs: All the components in the class hierarchy, shown in Figure 45 are refined to sets. By default all the leaf components in the component hierarchy are translated to string sets and the abstract internal classes are translated to compound sets. For example, *Alternative* component is translated to *iAlternative*, a string set. Whereas *DSComponent* is translated to translated to a compound set, such that $iDSComponent = iPrimitive \cup iContainer$, where *iContainer* in turn is also a compound set. The properties of *DSComponent* are translated to functions (decision variables or input parameters) depending on their type. The containment relation *Children* is translated to a function

$$iChildren : iContainer \rightarrow \mathscr{P}(iDSComponent)$$

Outputs: The metric and decision properties of the features: `memory`, `sel` and `cost` are refined to corresponding decision variables. For example, the `select` property of *DSComponent* is translated to a decision variable of function type. The assignment statements that are used to assign value to `select` property in the derived classes of *DSComponent*, for example *Alternative*, *Mandatory* are combined to form a conditional definition of the following form:

$DSComponent\_memory : DSComponent \rightarrow \mathbb{N}$, with definition

$$DSComponent\_memory(x) = \begin{cases} \text{sum}(m \text{ in } Children(x)) & (DSComponent\_memory(m)) \\ & \text{if } x \in iMandatory \\ \text{sum}(a \text{ in } Children(x)) & (DSComponent\_memory(a) \\ & *DSComponent\_select(a)) \\ & \text{if } x \in iAlternative \\ \text{sum}(o \text{ in } Children(x)) & (DSComponent\_memory(o) \\ & *DSComponent\_select(o)) \\ & \text{if } x \in iOr \\ Feature\_memory(x) & \text{if } x \in iFeature \end{cases}$$

The memory property depends on the selection property of the *DSComponent*, which is translated to :

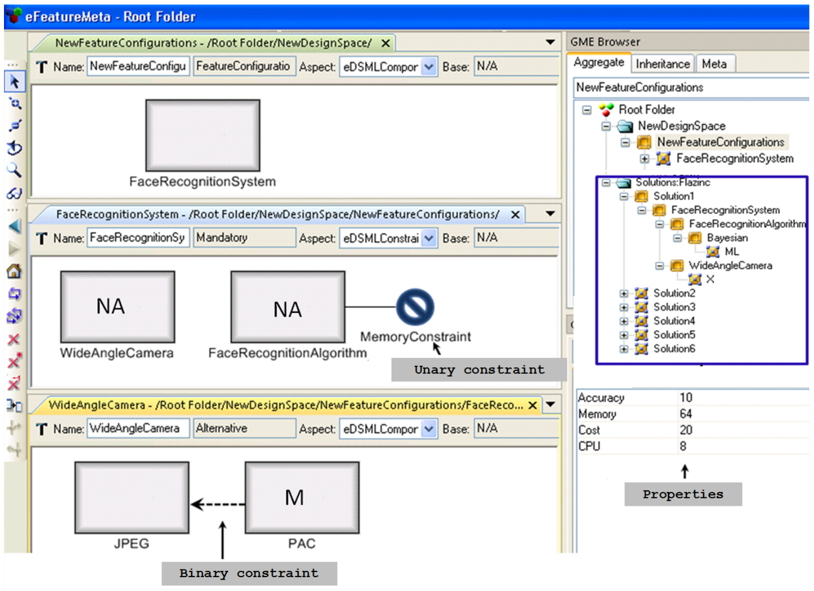
$$DSComponent\_select : DSComponent \rightarrow B$$

Constraints in the conceptual model are refined to predicates and constraints.

- composition and cost constraints are applicable to all instances of the context component and are thus refined to constraint objects in IRL model.
- bounds and dependency constraints are explicitly instantiated in the design space instance model and are thus refined to predicate objects in the IRL model.

IV.2.3 Model and search

In order to evaluate our framework we use a simplified version of the face recognition system problem adapted from[131] as a case study. The face recognition system problem has a feature diagram, shown in Figure 44, with three constraints: (1) a dependency constraint where *PCA* variant of the algorithm requires the image in *JPEG* compression format, and (2) the face recognition algorithm must require, $Memory \leq 2048$. The objective of the exploration is to find a subset of features that satisfy the composition and bounding resource constraints, while optimizing the cost consumption.



Accuracy	10
Memory	64
Cost	20
CPU	8

Figure 53: eFeature Design Space Instance

The CP model generated from the IRL model and the data generated from the problem model M_{CM} is processed by the solver to retrieve solutions. The model corresponds to the feature model in Figure 44. The data and CSP specification are given to the Minizinc finite domain solver. One of the solutions returned consists of one instance of the camera and ML implementation of the algorithm.

IV.3 Placement Problems

A typical placement problem involves arranging objects according geometric constraints, where all objects must lie within the given boundary and the objects do not overlap. Placement problems have been found most often in VLSI design. Formally, the VLSI placement problem deals with the nets and modules. The nets connect the modules through pins. A circuit is a tuple $C = \langle M, P, N, A \rangle$, where M is a set of modules, P is a set of pins, N is a set of nets and A is the area with IO-pins. Each module $m \in M$ has a width (w_m) and height (h_m) of the module. Modules are connected to each other through pins. Each pin $p \in P$ is mapped to a module. Pins are placed with respect to the lower left corner of the module. Nets are subsets of P such that each $n \in N$ is $n \subseteq P$. All pins should be a part of exactly one net. A placement of a circuit C describes the position of each module.

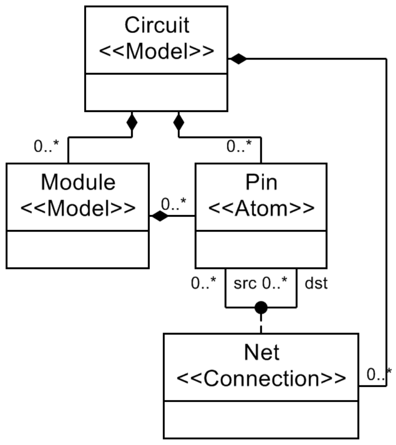


Figure 54: DSML for Circuits

The goal of the placement problem is to find a legal placement. Formally, a placement problem is a tuple

$$\langle R, C, M, M_x, M_y \rangle \tag{IV.3}$$

where R is a set of rows in the layout grid, C is a set of columns, M is a set of modules, every single module has a height and width. $M_x : M \rightarrow C$ represents the column positions of modules, and $M_y : M \to R$ represents the row position of modules. The goal is to find M_x and M_y, such that the modules do not overlap, that is for any

$$\begin{aligned}\forall m_1, m_2 \in M,\ M_x(m_1) \geq M_x(m_2) + width(m_2)\\ \vee M_x(m_2) \geq M_x(m_1) + width(m_1))\\ \vee M_y(m_1) \geq M_y(m_2) + height(m_2)\\ \vee M_y(m_2) \geq M_y(m_1) + height(m_1))\end{aligned} \tag{IV.4}$$

The objective is to find an optimal placement of the modules, such that length of the wires required to connect the pins is minimum. Two pins are connected only if they belong to the same net. Distance between pins is calculated by the Manhattan distance between the pins.

$$total\ wire\ length = \sum_{(p_1,p_2)\in Net} \mid P_x(p1) - P_y(p2) \mid + \mid P_x(p1) - P_x(p2) \mid \tag{IV.5}$$

where P_x and P_y give the position of the pins on the grid.

IV.3.1 Configuration

Based on the problem specification, the domain expert creates a conceptual model MM_{CM} representing the class of placement problems by associating DSE aspects with domain-specific modeling concepts. The domain-specific modeling language (DSML), captured as a metamodel MM_{DSML} is shown in Figure 54. The classes in MM_{DSML} that are relevant to DSE are composed with the ADSEL classes to capture the different aspects of the placement problem. The resulting conceptual model consists of components, associations, constraints, objectives and global variables for modeling a class of feature selection problems.

Components: In this problem, the *Module* and *Pin* concepts in the DSML metamodel are included in the conceptual model, as shown in Figure 55. The *Module* component is

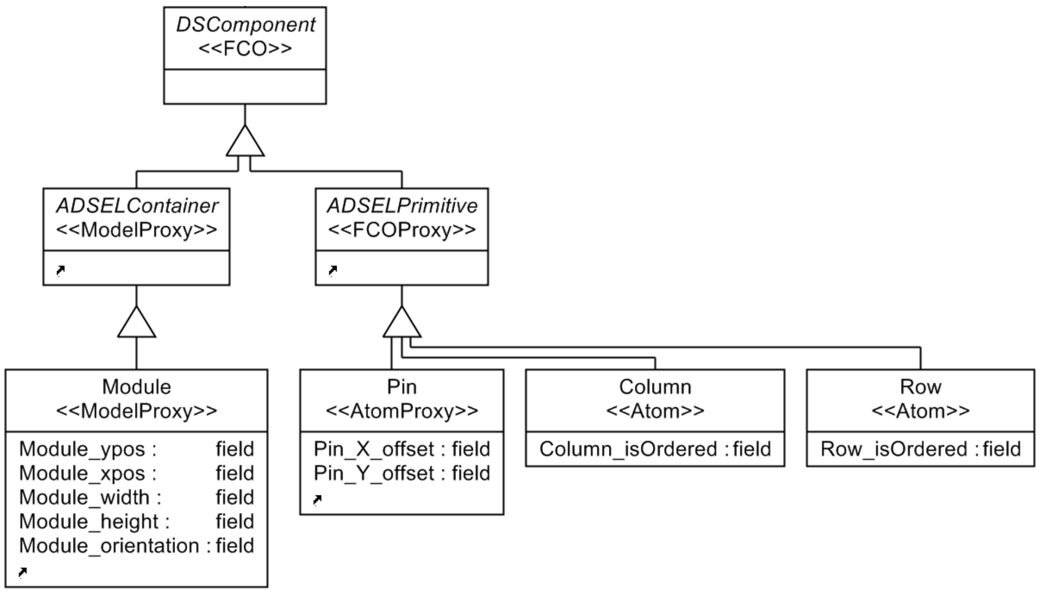


Figure 55: Placement Conceptual Model Components

a *Container* component, and *Pin*, a *Primitive* component. The *Module* has `width` and `height` parametric properties. The position of a *Module* on the grid is modeled by two decision properties: `xpos`, `ypos`, which gives the lower left corner of the module. The pins in a module are located at a certain offset with respect to location of the parent module, which is modeled using `xoffset` and `yoffset` properties of the *Pin* class. Two new primitives, namely *Row* and *Column*, are created to model the rows and columns in a grid. These primitives are *ordered* and therefore each object of the primitive can be uniquely identified with a natural number $n \in [1...max\_cardinality]$, where $max\_cardinality$ is the maximum number of objects that can be included in a design space model M_D

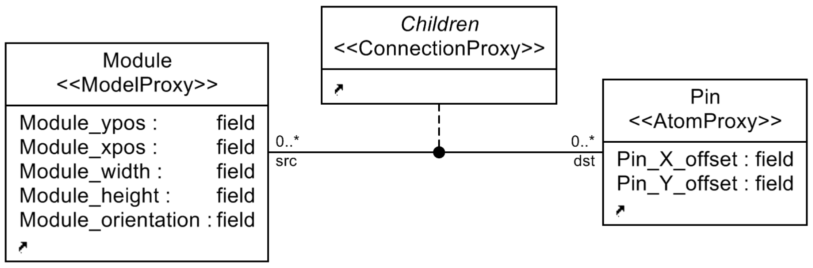


Figure 56: *Children* association reference

Associations: There are two associations in this conceptual model: the *Net* association, and the *Children* association. The *Net* association in the *DSML* metamodel (see Figure 54) is included in the conceptual model as a *Binary Association*. A reference to containment relationship between *Module*s and *Pin*s is also included in the conceptual model, as shown in Figure 56. This is required by the search procedure to calculate the positions of the *Pins* based on the position of the *Module*s. Formally,

$$Children : Module \rightarrow \mathscr{P}(Pin) \tag{IV.6}$$

Global Variables: The goal of the search is to find a legal placement which satisfies all the constraints and minimizes the length of the nets. The global variable *total_wire_length* is used to calculate the total length of the nets for a given placement, where length of each net is given by the Manhattan distance between the position of the source and target pins. The assignment expression of the variable is given in Figure 57 .

```
context: total_wire_length
CSL assignment statement:
$self = sum (m1  in Module )
(
 sum ( p1  in  Children(m1, _), p2  in  Children(m2, _) )
 ( bool2int( Net(p1,p2) )
   * abs( p1.xoffset+ m1.xpos -   p1.xoffset+ m1.xpos )
   + abs( p1.yoffset+ m1.ypos -   p1.yoffset+ m1.ypos )
  )
)
```

Figure 57: Total Length CSL Assignment Statement

Constraints:

C1. *NoOverlap* The modules should be placed such that they do not overlap each other and all the modules should lie with in the area of the circuit. This constraint is

```
context: noOverlap, $ctx1 = Module, $ctx2 = Module
CSL constraint:
forall($ctx1,  $ctx2 )
(
    ($ctx2.xpos  + $ctx2.width  <= $ctx1.xpos )
    \/  ($ctx1.xpos  + $ctx1 .width  <= $ctx2.xpos )
    \/  ($ctx2.xpos  + $ctx2 .height  <= $ctx1.ypos )
    \/  ($ctx1.xpos  + $ctx1 .height  <= $ctx2.ypos )
)
```

Figure 58: No Overlap CSL Constraint

formally specified in Eq. IV.4. The CSL definition of the constraint is shown in Figure 58.

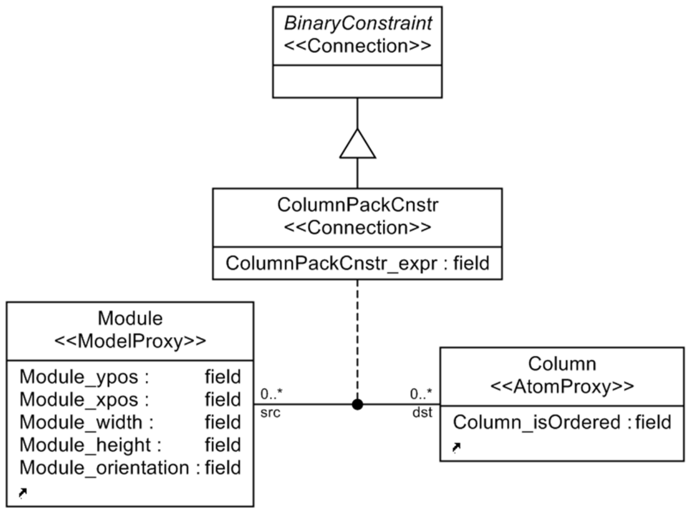


Figure 59: Packing Constraint

C2. There are two binary packing constraints *RowPackCnstr* and *ColumnPackCnstr* to ensure the placement of the modules is within the height and width of the grid. The *ColumnPackCnstr* (shown in Figure 59) specifies that the total height of all modules placed in a column should not exceed the height of the grid. The corresponding CSL constraint is given in Figure 60. The *RowPackCnstr* is similarly used to specify the

```
context: columnPack, $ctx1 = Module, $ctx2 = Column
CSL constraint:
forall($ctx2 )
( sum($ctx1 )
    ( bool2int(  ($ctx2.index >= $ctx1.xpos)
                 /\  ($ctx2.index <=$ctx1.xpos +$ctx1.width)  )
    * $ctx1.height
   ) <= card($ctx2)
)
```

Figure 60: Column Packing CSL Constraint

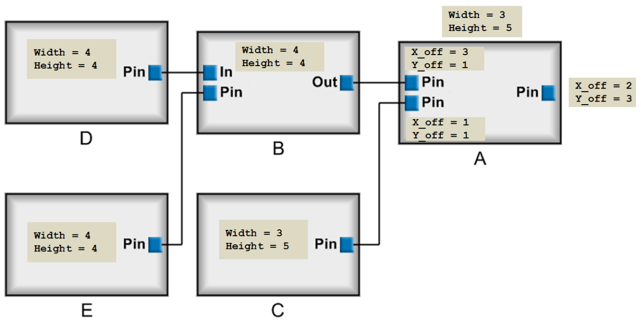
constraint the total width of all elements in a *Row* is not more than the width of the grid.

Objective: The overall objective is to minimize the total wire length.

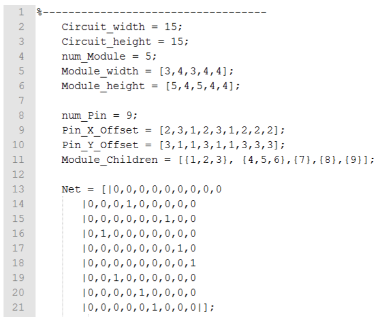
IV.3.2 Model and search

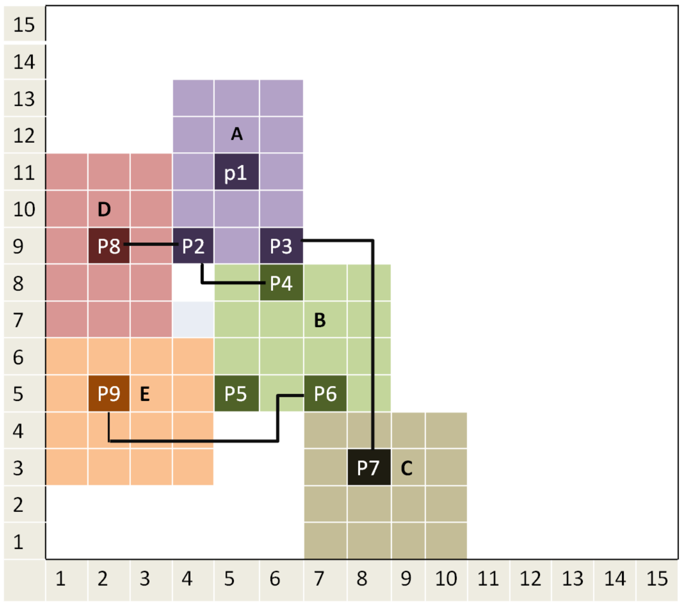
Figure 61(a) shows a simple example with five modules. The *Row* and *Column* both have cardinality `15` respectively. The CP Model auto-generated from the the IRL model of the problem is used in conjunction with the data generated from the design space model M_D. The resulting placement minimizes the total wire length of the nets. The solution in Figure 61(c) minimizes the total wire length of the nets to 18. The time taken to solve the problem averaged out to approximately 60 seconds.

The packing constraints can also be modeled using `cumulative` global constraints. We manually replaced the generated packing constraints with the global constraints and observed a speed up of more than 100% as a result of it. Thus, in order to handle larger instances, global constraints have to be manually introduced.



(a) Design Space Model M_D


```
%-----------------------------------
    Circuit_width = 15;
    Circuit_height = 15;
    num_Module = 5;
    Module_width = [3,4,3,4,4];
    Module_height = [5,4,5,4,4];

    num_Pin = 9;
    Pin_X_Offset = [2,3,1,2,3,1,2,2,2];
    Pin_Y_Offset = [3,1,1,3,1,1,3,3,3];
    Module_Children = [{1,2,3}, {4,5,6},{7},{8},{9}];

    Net = [|0,0,0,0,0,0,0,0,0
        |0,0,0,1,0,0,0,0,0
        |0,0,0,0,0,0,1,0,0
        |0,1,0,0,0,0,0,0,0
        |0,0,0,0,0,0,0,1,0
        |0,0,0,0,0,0,0,0,1
        |0,0,1,0,0,0,0,0,0
        |0,0,0,0,1,0,0,0,0
        |0,0,0,0,0,1,0,0,0|];
```
(b) Data

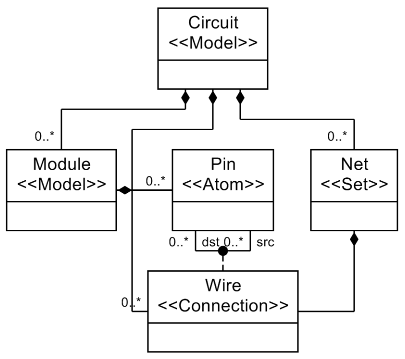


(c) Solution Model M_S

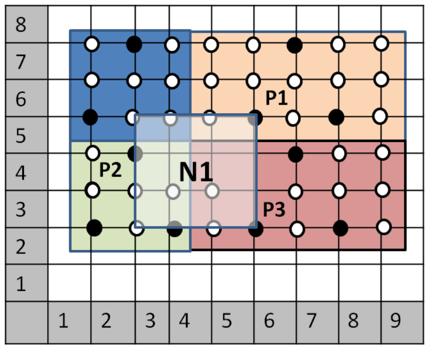
Figure 61: Placement Problem Model and and solution

IV.4 Routing Problems

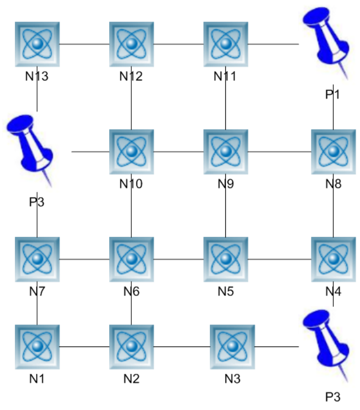
Routing problems are also found in VLSI design in combination with placement problems presented in Section IV.3. Figure 62(a) shows a DSML for VLSI, where a *Circuit* consists of *Module*s and each *Module* in turn consists of *Pin*s. A *Wire* connects a pair of *Pins*. A *Net* is a set of *Wire*s connecting the pins . Figure 62(b) shows a diagrammatic representation of a circuit with 4 modules placed on the grid. Each module (box) consists of pins (black dots). The circuit is divided into a routing grid. The intersection points are shown as white dots. A pin can be placed in an intersection point (black node). A wire connecting two pins is routed by selection of edges in the routing grid. The goal of the routing problem is to find a steiner tree connecting the pins in each net, such that the total length of the wires in the *Net* is minimized.



(a) Original Metamodel



(b) Route Problem Model



(c) Route Problem Grid

Figure 62: Routing Problem

Figure 62(c) shows a routing grid where three pins $p1$, $p2$ and $p3$ belong to a net $N1$ are to be connected. The goal is to find a steiner tree connecting the pins using (n-1) wires. One way of solving this problem is to assume one of the pins, say $p1$ as the start point of the path and the other pins $p2$ and $p3$ as the end points and for each wire, we try to find the shortest path.

Formally, a routing grid is a tuple $R = \langle N, E \rangle$, where N is a set of nodes and E is a set of edge, where $numWire(e)$ gives the number of wires routed through the edge $e \in E$. Then the following properties hold for all the nodes:

1. The wire should be routed, such that for each node, the number of incoming wires is equal to the number of outgoing wires except for the start and end nodes of the route.

$$\forall n \in Node \cdot \sum_{e \in Edge \wedge Src(e)=n} numWire(e) == \sum_{e \in Edge \wedge Dst(e)=n} numWire(e) \tag{IV.7}$$

2. For the start node the number of outgoing wires should be equal to the number of end points in the net.

$$\forall n \in Node \cdot isStart(n) \rightarrow \sum_{e \in Edge \wedge Src(e)=n} numWire(e) == \sum_{n \in Node} isEnd(n) \tag{IV.8}$$

3. For the end node there should be no outgoing wires and only one incoming wire.

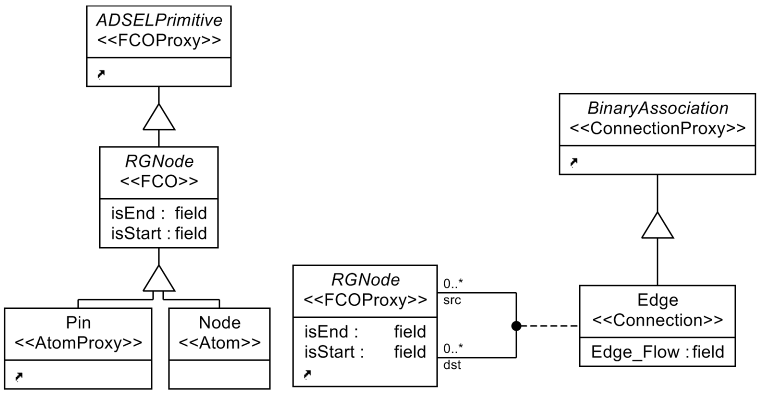
$$\forall n \in Node \cdot isEnd(n) \rightarrow \sum_{e \in Edge \wedge Dst(e)=n} numWire(e) == 1 \tag{IV.9}$$

The goal of the routing problem is to minimize the total length of the wire, that is calculated by adding the number of edges that have more than one wire.

IV.4.1 Configuration

The conceptual model representing the class of routing problems found in VLSI design consists a new component *RGNode* that is used to represent the intersection points in the routing grid. An intersection point can be a *Pin* or an empty intersection point. An association *Edge* is used to connect each pair of *RGNode*.

In order to retrieve a routing path, we need to assign a start node and one or more end nodes. In each net we assume one pin is the start and all the other pins are the end pins.



(a) Components (b) Associations

Figure 63: Routing Problem Conceptual Model

The is denoted by the `isStart` and `isEnd` parametric variables of the node. The actual number of wires that are routed through an edge are modeled as `Edge_flow` decision variable which is calculated as a result of the search. An edge is considered as a part of the routing path if at least on wire *flows* through it.

Besides these, a global variable is used to capture the total wire length of the route. The goal of the exploration is find a route that satisfies all the constraints and also has minimum total wire length. The global variable *total_wire_length* is used to calculate the total length of a single net, where the wire length is equal to the lengths of the edges selected to form the route. The assignment statement of the variable is shown in Figure 64.

```
context: total_wire_length
CSL constraint:
$self= sum(<p1,p2> in Edge)( bool2int(  Edge(p1,p2).edge_flow >0 ) )
```

Figure 64: Total wire length CSL Constraint

Constraints : The CSL constraints corresponding to the constraint in Eq. IV.7 and Eq. IV.8 are given in Figure 65 and Figure 66 respectively.

```
context: Node
CSL constraint:
forall($ctx1)
 ( not($ctx1.isStart) /\ not ($ctx1.isEnd)
   ->  (sum( n in Node)( Edge($ctx1, n).edge_flow)
            ==   sum( n in Node)( Edge(n,$ctx1).edge_flow)
        )
)
```

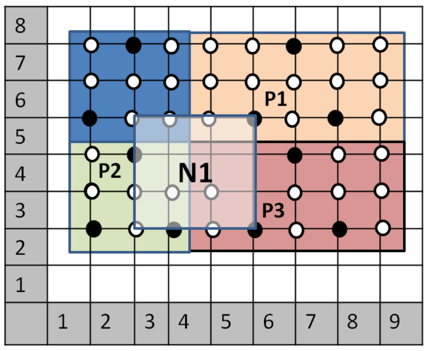
Figure 65: Node constraint in Routing Problem

```
context: Node
CSL constraint:
forall($ctx1)
 ( ($ctx1.isStart)
   ->  (sum( n in Node)( Edge($ctx1, n).edge_flow )
            ==   sum( n in Node)( bool2int)(n.isEnd) )
        )
)
```

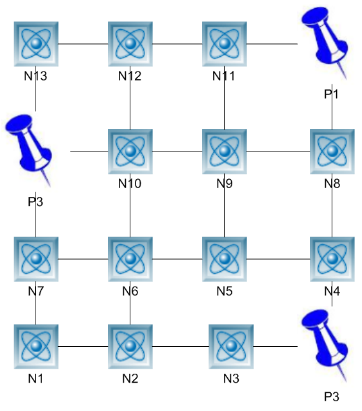
Figure 66: Start constraint in Routing Problem

IV.4.2 Model and Search

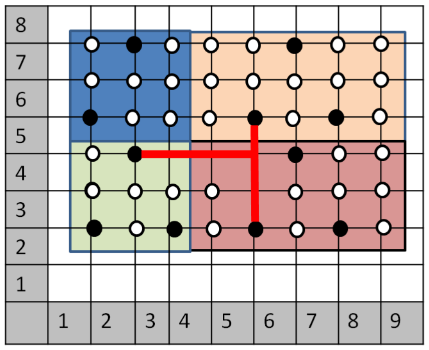
Figure 67(a) shows a simple example with 4 modules and a net N_1 connecting the pins P_1, P_2 and P_3 of three of these modules. The routing of the net N_1 is done by modeling only a subset of the grid as shown in Figure 67(b). The IRL model is refined to a CP model and used in conjunction with the that is used in conjunction with the data generated from the model in Figure 67(b). The solution of the routing problem returns the layout of the wires connecting the pins in the net such that the route minimizes the total wire length of the nets, as shown in Figure 67(c).



(a) Route Problem Model



(b) Route Problem Grid



(c) Route Problem Solution

Figure 67: Routing Problem and Solution

IV.5 Scheduling Problems

Scheduling problems is a common category of DSE problems often found in embedded systems [72], [80]. Several variants of th scheduling problem exist, depending on the constraints imposed due to the properties of the tasks and resources. We use the scheduling problem as a case study to evaluate the adaptability of the framework. We first model and solve a class of generic scheduling problems to evaluate the scope of the framework in handling scheduling problems. Then we introduce additional requirements to model special classes of scheduling problem (example tasks are non-preemptive, resources are cumulative) and evaluate if the conceptual model of the generic scheduling can be reused to model the new class of problems.

A scheduling problem is characterized by assignment of start times to each task, such that both temporal constraints and resource constraints are satisfied. Broadly, there are two kinds of resource and tasks [98]:

- A ***disjunctive resource*** can execute at most one task at each point in time. The tasks which require this resource can execute only when no other task is executing on the resource.
- A ***cumulative resource*** can execute several activities in parallel, provided the resource requirement of the executing tasks does not exceed the resource capacity at any point of time.
- A ***non-preemptive task*** must execute without interruption from start to end.
- A ***preemptive task*** can be interrupted by other task depending on some priority etc.

Most real scheduling problems consist of a combination of cumulative and disjunctive resources as well as both interruptible and non-interruptible activities. In this section we discuss three classes of scheduling problems: (1) Generic Scheduling; (2) Non-preemptive disjunctive; and (3) Non-preemptive cumulative.

IV.5.1 Generic Schedule Problem

The generic scheduling problem corresponds to constraint-free scheduling, where any number of tasks can be mapped onto a single resource and there are no constraints on how many tasks can execute on a resource and a given instance of time.

Problem Specification: Formally, a scheduling problem is a tuple

$$SP = \langle Task, Rsrc, Slot, Interval, Dur, Sched, C, makespan \rangle \tag{IV.10}$$

- *Task* is a set of activities with given processing times and resource requirements, Each task t_j has an attribute $\{duration_j, start_j, end_j\}$ where dur_j gives execution duration of the task, $start_j$ gives the start time of the task, end_j gives the end time of the task.

- $Rsrc$ is a set of resources with given capacities,
- $Slot$ is an ordered set of integer time slots,
- $Interval : Task \rightarrow \mathscr{P}(Slot)$
- $Sched : Task \rightarrow Rsrc$, is a function which assigns a task to a resource.
- C is a set of temporal constraints between tasks.
- $makespan \in Slot$, which is the end time of the last task that executes in the schedule.

Objective is to find $Interval$ and $Sched$ such that the schedule has minimum makespan.

Task Constraints specify generic constraints on task properties. The start time of the task is the minimum time slot in the interval and the end time is the maximum time slot in the interval. The size of the interval set is equal to the duration of the task.

$$\forall t \in Task \cdot Start(t) = min(Interval(t)) \tag{IV.11}$$

where $Start(t)$ returns the start property of task t.

$$\forall tsk \in Task \cdot End(tsk) = min(Interval(tsk)) \tag{IV.12}$$

where $End(t)$ returns the end property of task t.

$$\forall tsk \in Task \cdot Dur(tsk) = |Interval(tsk)| \tag{IV.13}$$

where $Dur(tsk)$ gives the duration of tsk. Any task execution must end before the *makespan*,

$$\forall tsk \in Task \cdot End(tsk) \leq makespan \leq max(Slot) \tag{IV.14}$$

The *makespan* is always less than the maximum time slot.

The template includes a set of temporal constraints that can be used for relative or absolution ordering of the task execution. The Eqs. IV.15-IV.18 are used to order two tasks relative to each other. For example, Eq IV.15 can be used to place a temporal constraint between two instances of a periodic task. Similarly, Eq IV.16 specifies that task tsk_i should end before tsk_j starts. This constraint can be used to model precedence constraint.

$$Start(tsk_i) + value <= Start(tsk_j) \tag{IV.15}$$

$$End(tsk_i) <= Start(tsk_j) \tag{IV.16}$$

$$Start(tsk_i) <= End(tsk_j) \tag{IV.17}$$

$$End(tsk_i) <= End(tsk_j) \tag{IV.18}$$

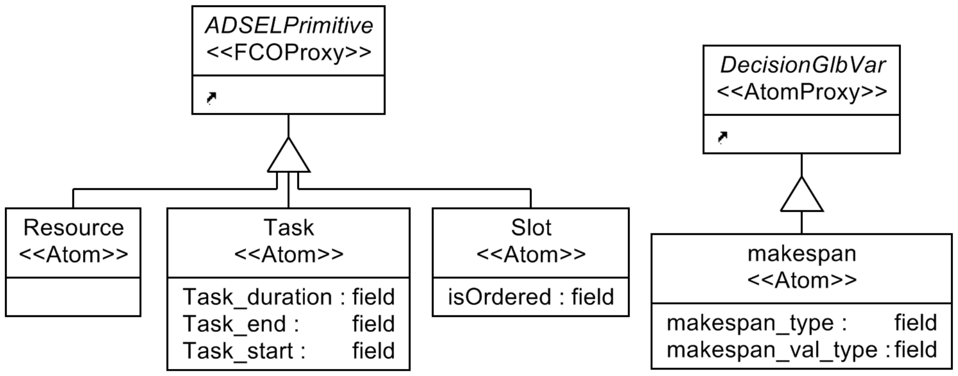
$$End(tsk) <= N \tag{IV.19}$$

$$N <= Start(tsk) \tag{IV.20}$$

IV.5.1.1 Configuration

The conceptual model of the generic scheduling problem captures the different aspects of the problem specification.

Components: The two basic concepts in a scheduling problem are resources and tasks. Figure 68(a) shows the components used to model these concepts. The *Resource* primitive component models a set of resources in the problem. The *Task* component models a set of schedulable activities in the problem where each $tsk \in Task$ has a set of properties. The



(a) Generic Scheduling Components (b) Generic Scheduling Global Variables

Figure 68: Schedule Template Components and Global Variable

`duration` parameter property of a task *tsk* represents the actual processing time of *tsk* on a resource *r*. The class of problems considered in this section are assumed to have identical resources, such that the processing time of task *tsk* remains the same irrespective of the resource on which it is scheduled. The timeline of the schedule is modeled using the *Slot*, an ordered component class that represents a set of objects that can be accessed using unique index number in the range `[1...max_cardinality]`, where `max_cardinality` is the maximum number of objects of *Slot* that can be included in the problem model.

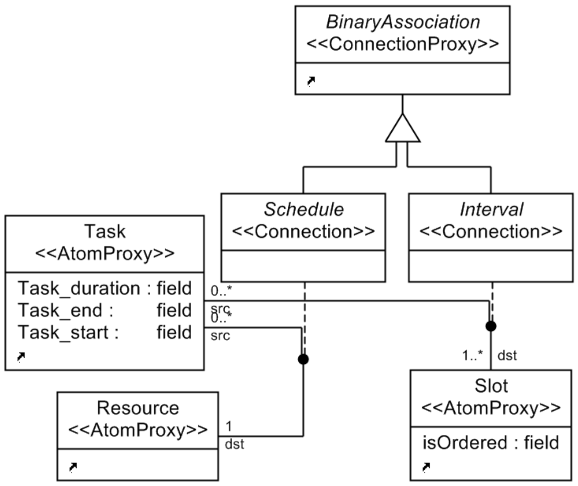


Figure 69: Generic Scheduling Associations

Associations: The conceptual model has two associations, shown in Figure 69. The *Schedule* association models the mapping of *Task* to a *Resource*, where each task is associated with exactly one resource and a number of tasks can be associated with the same resource.

Each task, irrespective of the resource on which it is mapped, executes in an interval of time modeled as a set of $Slot$. This is modeled using the *Interval* association from the *Task* to *Slot*. Each task is associated with a set of time slots reflected by the cardinalities. For example, a non-preemptive task is associated with a sequence `[A.start...A.end]` of contiguous time slots. Whereas a preemptive task is associated with a set of (discontinuous) time slots where the minimum value of the set is the start time and the maximum value is end time of the task.

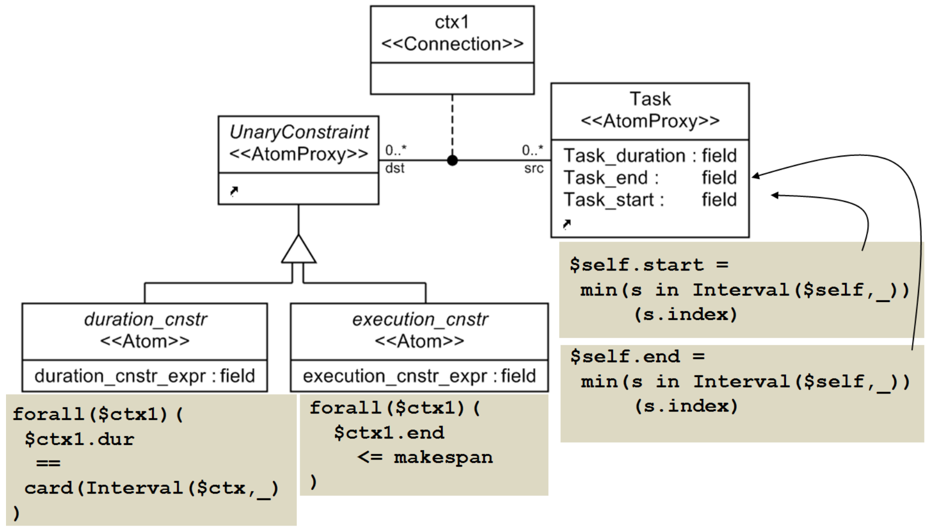
Metric Properties: The `start` and `end` are *metric* properties of the *Task* component (shown in Figure 68(a)) that represent the actual start and end time of execution of the task on a resource. The task constraints specified in Eq.IV.11 and IV.12 are modeled as assignment statements of the metric properties. Figure 70 (a) shows the assignment statements in the CSL.

Constraints: Figure 70(a) shows four temporal constraints corresponding to Eq. IV.15 - IV.15 that can be used to order tasks. For example, the end_before_start constraint is used to order two tasks T_i and T_j such that T_i finishes before T_j.

Objective:One of the common goals in the scheduling problems is to reduce the overall execution time of the set of tasks. Since makespan represents the overall execution time of the schedule, the template includes a $Minimize$ objective to minimize makespan.

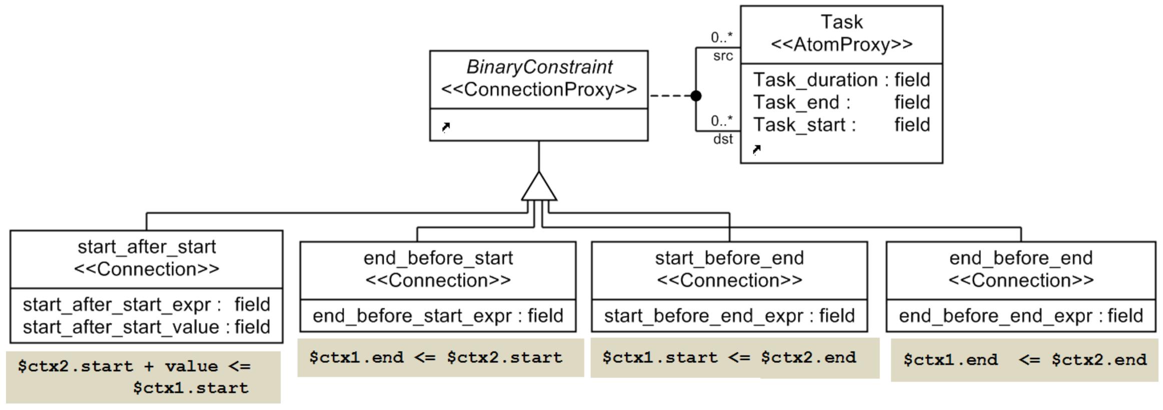
IV.5.1.2 Model and Search

Figure 71 presents the task graph and resources of an example scheduling problem. The task graph consists of 8 tasks with fixed duration. The lines in the graph model precedence constraints between pairs of tasks. When the release time is not specified all the tasks are



ctx1
<<Connection>>
UnaryConstraint
<<AtomProxy>>
0..*
dst
0..*
src
Task
<<AtomProxy>>
Task_duration : field
Task_end : field
Task_start : field
$self.start = min(s in Interval($self,_)) (s.index)
$self.end = min(s in Interval($self,_)) (s.index)
duration_cnstr
<<Atom>>
duration_cnstr_expr : field
execution_cnstr
<<Atom>>
execution_cnstr_expr : field
forall($ctx1) ($ctx1.dur == card(Interval($ctx,_))
forall($ctx1) ($ctx1.end <= makespan)

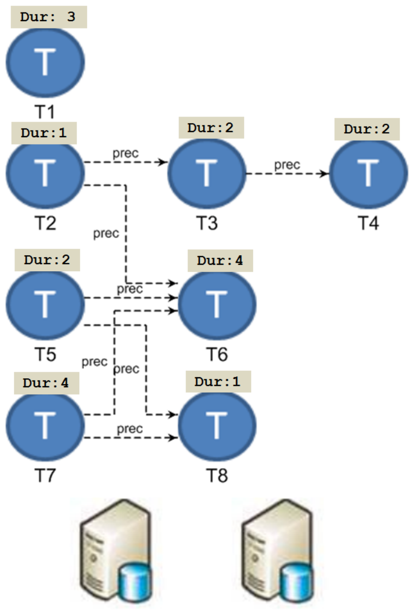
(a) Task Constraints



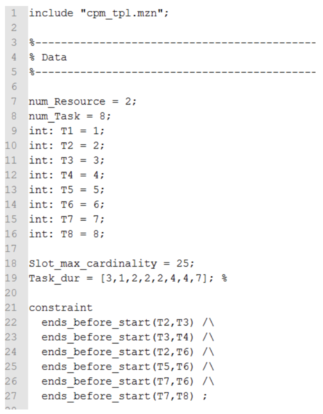
(b) Timing Constraints

Figure 70: Generic Scheduling Constaints

assumed to be released at time 0. The objective is to schedule and execute the tasks on the two resources. The resources are assumed to be identical. This example is used to test each of the conceptual models for correctness. Figure 71(c) presents the schedule generated by CPM_{tpl}, where the *Slot* ordered component has a cardinality of 25. In a generic schedule the ordering is done to satisfy the precedence constraints. As the resources do not have any capacity constraints, all tasks are mapped to one resource.

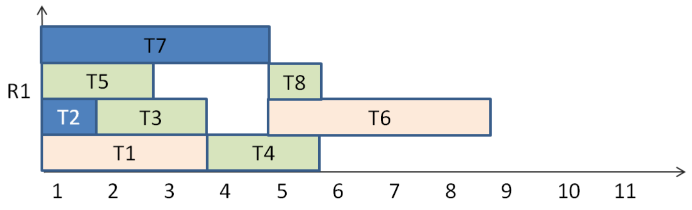


(a) Scheduling Example



```
include "cpm_tpl.mzn";

%-------------------------------------------
% Data
%-------------------------------------------

num_Resource = 2;
num_Task = 8;
int: T1 = 1;
int: T2 = 2;
int: T3 = 3;
int: T4 = 4;
int: T5 = 5;
int: T6 = 6;
int: T7 = 7;
int: T8 = 8;

Slot_max_cardinality = 25;
Task_dur = [3,1,2,2,2,4,4,7]; %

constraint
  ends_before_start(T2,T3) /\
  ends_before_start(T3,T4) /\
  ends_before_start(T2,T6) /\
  ends_before_start(T5,T6) /\
  ends_before_start(T7,T6) /\
  ends_before_start(T7,T8) ;
```

(b) Example Data



(c) Generated schedule

Figure 71: Generic Schedule Example

IV.5.2 Non-preemptive Disjunctive (NPD) Scheduling

In this section we illustrate how a class of non-preemptive disjunctive scheduling problems can be solved in the framework by reusing the conceptual model for generic scheduling problems created in the previous section. The conceptual model of the non-preemptive disjunctive scheduling CM_{npd} consists of all the components and associations in the generic problem conceptual model. Additionally the CM_{npd} consists of a task constraint to model the non preemptive property of tasks and a resource constraint to model the disjunctive nature of resources in this class of problems.

IV.5.2.1 Configuration

Task Constraint:The duration of the task tsk, given by $Dur(tsk)$ is equal to the size of the time interval *Interval(tsk)* in which the task executes. Execution of a non-preemptive task can not be interrupted by another task. A set of contiguous time slots satisfies the condition specified by Eq. IV.21, where the difference between the maximum and minimum time slots is equal to the cardinality of the time interval.

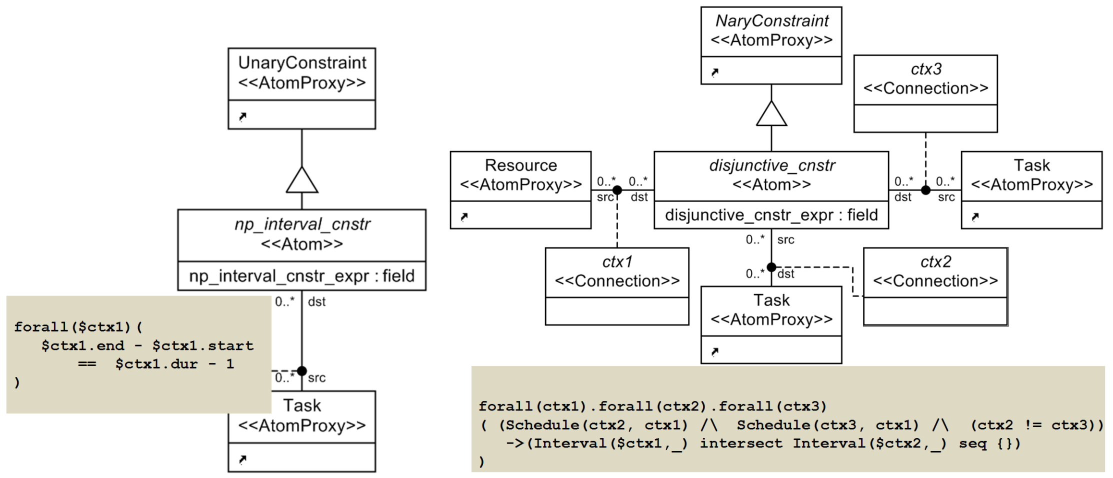
$$\forall tsk \in Task \cdot \max(Interval(tsk)) - \min(Interval(tsk)) = |Interval(tsk)| - 1 \qquad \text{(IV.21)}$$

Figure 72(a) models this non-preemptive task constraint.

Resource Constraint: A disjunctive resource $rsrc_i$ can execute only one task at a time. The $Schedule(rsrc_i)$ functions returns all the tasks that execute on the resource $rsrc_i$. Any pair of tasks tsk_i and tsk_j that require the same disjunctive resource $rsrc_i$ execute in disjoint intervals of time.

$$\forall tsk_i, tsk_j \in Schedule(\sim rsrc_i) \cdot tsk_i \neq tsk_j \rightarrow Interval(tsk_i) \cap Interval(tsk_j) = \emptyset \quad \text{(IV.22)}$$

The disjunctive resource constraint is modeled using a ternary constraint and is applicable to all instances of the resource. Figure 72(b) shows the CSL expression of the disjunctive constraint.



```
forall($ctx1) (
   $ctx1.end - $ctx1.start
       == $ctx1.dur - 1
)
```

```
forall(ctx1).forall(ctx2).forall(ctx3)
( (Schedule(ctx2, ctx1) /\  Schedule(ctx3, ctx1) /\  (ctx2 != ctx3))
   ->(Interval($ctx1,_) intersect Interval($ctx2,_) seq {})
)
```

(a) Task Constraints (b) Resource Constraint

Figure 72: Sched Template Constaints

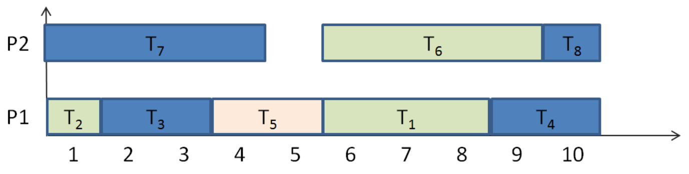


Figure 73: Non-preemptive disjunctive schedule for example in Fig. 71

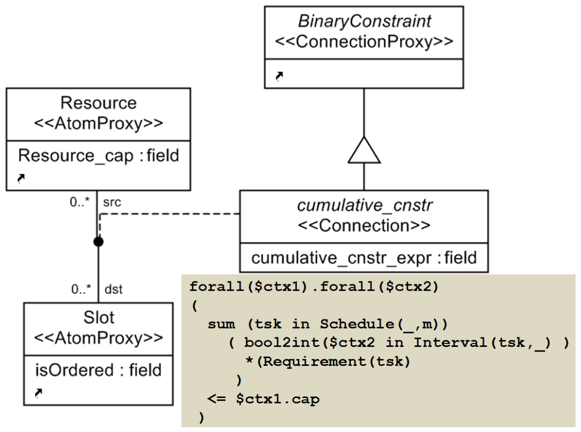
IV.5.2.2 Model and Search

Figure 73 shows the non-preemptive disjunctive schedule generated for the example problem in Figure 71(a). The schedule satisfies the precedence constraints and therefore cannot resource *P*2 cannot execute any task in time slot 5. The total makespan of the generated schedule is 10.

IV.5.3 Non-Preemptive Cumulative (NPC) Scheduling

Non-preemptive cumulative scheduling is a special class of scheduling problems in which a resource can execute more than one task at a given instance of time provided the combined resource requirement of the tasks does not exceed the capacity of the resource. The conceptual model CM_{npc} is an extension of CM_{tpl} such that $CM_{npc} = \langle CM_{tpl}, P_{npc}, C_{npc} \rangle$, where P_{npc} is a set of new properties added to existing components and C_{npc} is a set of additional constraints that specify a constraint imposed by the new properties. The P_{npc} set consists of two properties: (1) `Task_req`, which models the resource requirement of the task, and (2) `Resource_cap`, which models the capacity of the resource component.

IV.5.3.1 Configuration



```
forall($ctx1).forall($ctx2)
(
  sum (tsk in Schedule(_,m))
    ( bool2int($ctx2 in Interval(tsk,_) )
      *(Requirement(tsk)
     )
  <= $ctx1.cap
 )
```

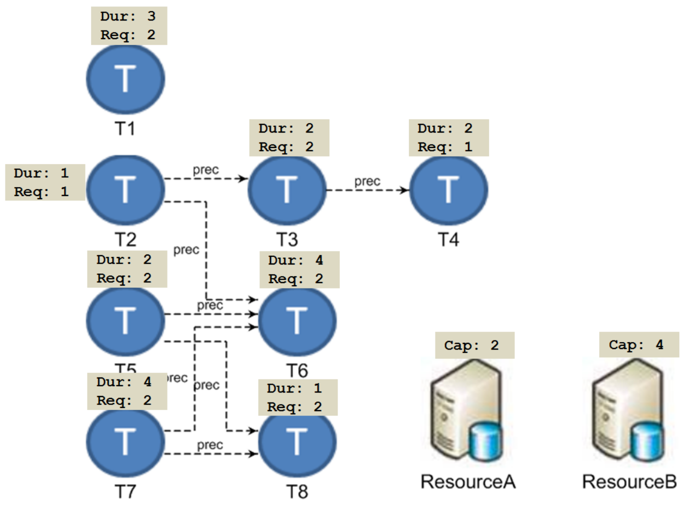
Figure 74: Cumulative Constraint

A cumulative resources can execute any number of tasks, unlike the disjunctive resource. However, the requirements of the tasks executing on the resource at a particular instance of time, say $t \in Timeline$ should not exceed the capacity of the resource. Formally, the constraint can be specified as follows:

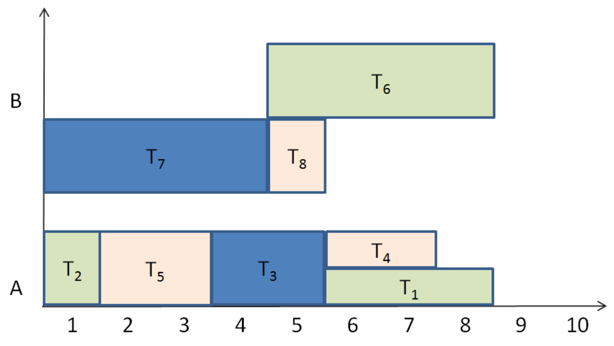
$$\forall rsrc \in Resource \cdot \forall t \in Slot \cdot$$

$$\sum_{tsk \in Schedule(\sim rsrc) \wedge t \in Interval(tsk} Task\_req(tsk) \leq Capacity(rsrc) \qquad \text{(IV.23)}$$

This constraint is modeled using the binary constraint *CumulativeCnstr*, shown in Figure 74. The constraint class is abstract since the same constraint is applicable to all the instances of the context classes. The task constraint remains the same as the one shown in the non-preemptive disjunctive scheduling (Eq. IV.21), where it models the non-preemptive property of the task.



(a) Scheduling Example



(b) Non-preemptive cumulative schedule

Figure 75: Non-preemptive cumulative scheduling example

IV.5.3.2 Model and Search

Figure 75 shows the scheduling example from Figure 71(a) with task requirements and the resource capacities. Figure 75(b) shows the generated cumulative schedule with the minimum makespan. The task $T6$ cannot execute on resource B even though the processor can fulfill the requirements of the task because of the precedence constraints which require that task $T6$ can start only after $T2$, $T5$ and $T7$ have completed.

IV.6 Functional Allocation Problem with multiple objectives

In this section, we use a functional allocation problem [38] to evaluate the use of different solvers to solve the same problem. We discuss how the same problem can be solved with constraint programming and evolutionary algorithms based on the number of objectives in a problem model belonging to the class. The same problem model is used to retieve valid solutions, mono-objective optimal solutions, while the MOP model is used to retrieve Pareto-optimal solutions when more than one objective is included in the problem model.

Problem Specification: The problem consists of application and architecture models, and an explicit mapping step to relate the two models.

Application model consists of a directed graph $KPN = (V_K, E_K)$, where V_K is a set of Kahn nodes and E_K is a set of directed FIFO channels between the nodes. Each node $a \in V_K$ is connected to a set of channels and each channel is connected to exactly two nodes.

Architecture model consists of a directed graph $ARC = (V_A, E_A)$, where V_A represent the architectural nodes and the E_A represents the connections between those nodes. An architectural node can be a memory or processor, such that $V_A = P \cup M$. Each processor $p \in P$ is connected to a set of memory nodes.

Mapping is associating the application model of the problem with the architecture model. x_{ap} denotes whether node a is mapped to processor p. Similarly x_{bm} and x_{bp} denotes whether channel b is mapped to memory m and p. The mapping must satisfy the following constraints:

- *C1:* each node $a \in V_K$ is mapped on to a single processor;

$$\forall a \in V_K \sum_{p \in Processor} x_{ap} = 1 \tag{IV.24}$$

- *C2:* every channel $c \in E_K$ is mapped onto a processor or a memory.

$$\forall b \in E_K \sum_{p \in Processor} x_{bp} + \sum_{m \in Memory} x_{bm} = 1 \quad \text{(IV.25)}$$

- *C3:* if two communicating nodes are mapped to the same processor then the channel(s) between them is also mapped onto the same processor.

$$x_{ip} = 1 \,\&\, x_{jp} = 1 \rightarrow \exists b = (i, j) \in E_K, x_{bp} = 1 \quad \text{(IV.26)}$$

- *C4:* if two communicating nodes are mapped to the different processors then the channel(s) between the two is mapped to one of the memories reachable by both the nodes.

$$p_k \neq p_l \in P, a_i, a_j \in V_K \,\&\, b = (a_i, a_j) \in E_K \cdot x_{a_i p_k} = 1 \wedge x_{a_j p_l} = 1 \rightarrow x_{bm} = 1 \quad \text{(IV.27)}$$

- *C5:* a processor is in use if at least one of the nodes is mapped onto it.

$$\forall p \in Processor \cdot y_p \longleftrightarrow \sum_{a \in Node} x_{ap} \geq 1 \quad \text{(IV.28)}$$

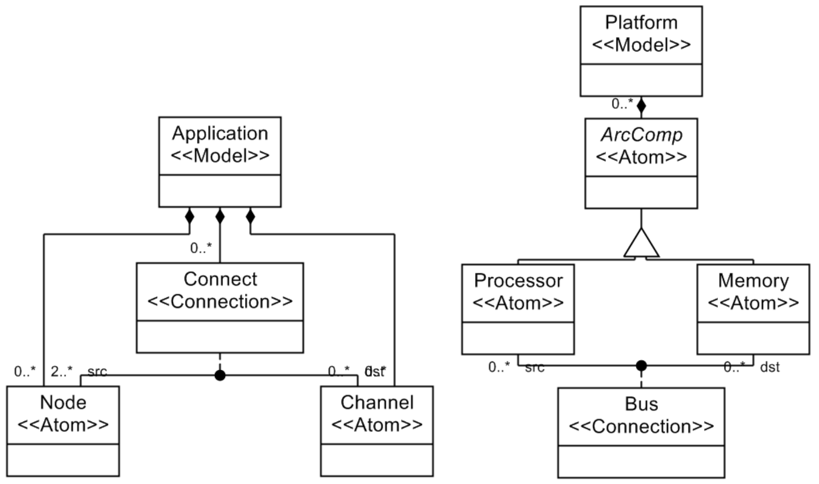
  Similarly, a memory is in use if at least one channel is mapped to it.

$$\forall m \in Memory \cdot y_m \longleftrightarrow \sum_{b \in Channel} x_{bm} \geq 1 \quad \text{(IV.29)}$$

The ***objective*** is to minimize find a mapping such that constraints are satisfied with respect to three objectives:

- minimize processing time of the application
- minimize power consumption of the system

- minimize power the total cost of the architecture



(a) Application original metamodel (b) Architecture original meta-model

Figure 76: Original Metamodel

IV.6.1 Conceptual Model

Components: The conceptual model extends the original metamodel to capture the inputs, outputs, constraints and objective of the problem. The original metamodel is shown in Figure 76. In the conceptual model the *Node*, *Channel* and *ArchComp* are *Primitive* components. Both the *Node* and *Channel* components have a fixed computation requirement modeled as `comp_req` parameter property. In addition to a computation requirement, the *Channel* component also has a fixed communication requirement, modeled as `comm_req` parameter property. In this problem, an architecture component can be a processor or a memory, where all architecture components have a fixed `cost` and processing `capacity`. All architecture component dissipate power while execution and communication at a fixed rate. This rate is given by `powd_rate`. The actual power dissipation depends on the processing time of the architecture component, which depends on the computation requirements of the nodes mapped onto it.

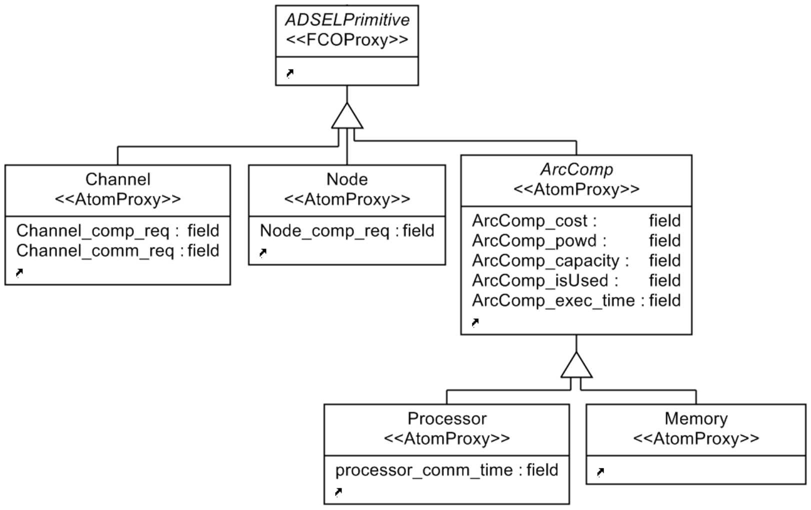


Figure 77: Components in the eSONET conceptual model

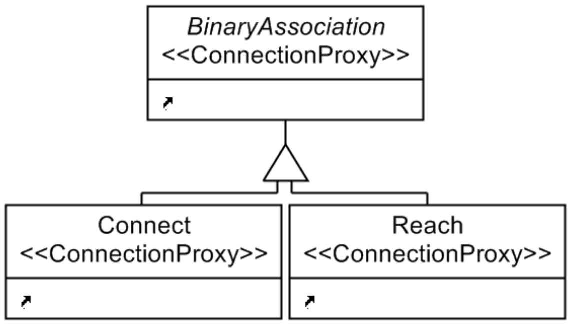
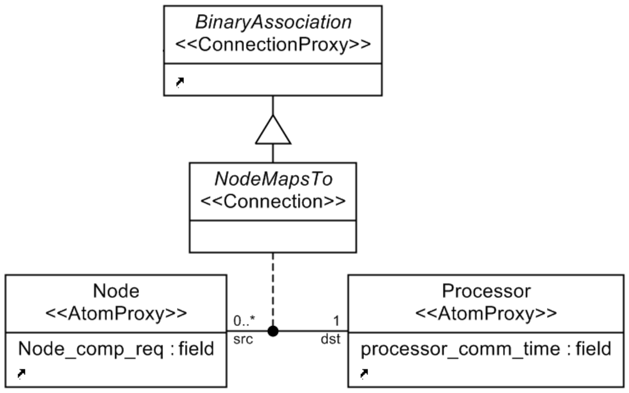
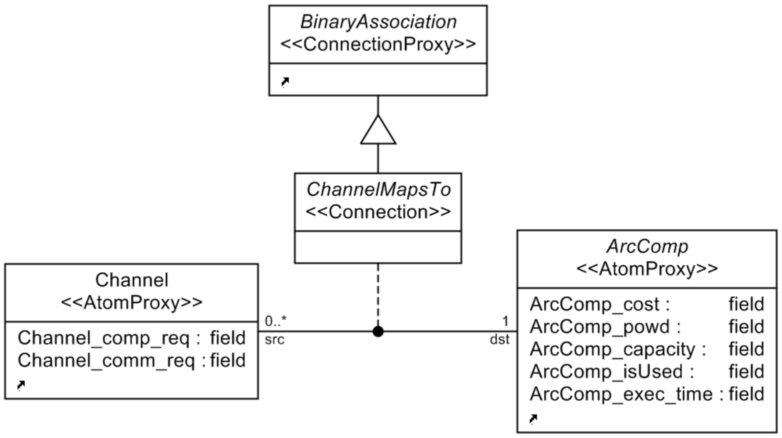


Figure 78: Concrete Associations

Associations: The *Connect* association (Figure 78) models a relationship between the *Node* and *Channel* components. Each node is connected to a set of channels and each channel is connected to exactly two nodes. The *Reach* association refers to the *Bus* connection from the processor to the memory components in the original metamodel that models the memory components that can be reached by the node. The *NodeMapsTo* (Figure 79) abstract association models the mapping of nodes to processors. Each node can be mapped to exactly one processor and a processor can have zero or more nodes mapped onto it. The *ChannelMapsTo* abstract association models the mapping of channels to an architecture component (memory or a processor). Each channel is mapped to one architecture component. The cardinalities of these associations enfore constraint *C1* and *C2*.



(a) Node to Processor Mapping



(b) Channel to Processor/Memory Mapping

Figure 79: Abstract Associations in the eSONET conceptual model

Metric Properties: An architecture component is in use if a node or a channel is mapped onto it. This constraint is given in equations IV.29 and IV.28. In the conceptual model, y_p and y_m are modeled as `isUsed` metric property of the *ArchComp*. This property is inherited by the *Processor* and the *Memory*. The CSL assignment statements corresponding to IV.29 and IV.28 are:

```
context: Processor
assignment statement:
   $self.isUsed <-> (  card ( NodeMapsTo( _, $self) ) >= 1 )

context: Memory
assignment statement:
   $self.isUsed <-> (  card ( ChannelMapsTo( _, $self) ) >= 1 )
```

The execution time of an architecture component is modeled as the `exec_time` metric property. The execution time of the processor includes the time spent in executing the nodes and the time spent in communication, which depends on the channels connected to the nodes. The value of the `exec_time` and `comm_time` properties for the processor are calculated using the following assignment statements:

```
context: Processor
assignment statement:
 $self.exec_time = (  sum ( n in NodeMapsTo( _, $self)   ) (n.comp_req)
                + $self. comm_time  )
```

```
context: Processor
assignment statement:
$self.comm_time = (  sum ( n in NodeMapsTo( _, $self))
                   (  sum ( c in Connect( n, _ ))
                      ( bool2int( not (ChannelMapsTo(c, $self ) )
                       *c.comm_req)
                   )
                )
```

where `$self` refers to the *Processor* component. The total processing time of a processor is given by the sum of the execution time of the nodes mapped on it and the communication required of the channels which communicate with the mapped nodes but are *not* mapped on same processor. Similarly the total processing time of a memory is given by:

```
context: Memory
assignment statement:
 $self.exec_time = (  sum ( c in ChannelMapsTo( _, $self)  ) (c.comp_req
    ) )
```

Decision Global Variables:Three metric global variables are included in the conceptual model, namely `sys_cost`, `sys_powd` and `sys_exec_time` representing the total cost, power dissipation and execution time of the system. The statements used for assigning values to these variables are given by:

(i) `sys_exec_time` models the processing time of the system. The value of this variable is assigned using the following statement:

```
context: sys_exec_time
assignment statement:
 $self = sum ( a in ArchComp )  ( bool2int(a.exec_time) )
```

(ii) `sys_cost` models the total cost of the system. The assignment statement is given by:

```
context: sys_cost
assignment statement:
 $self = sum ( a in ArchComp )  ( bool2int(a.isUsed) *a.cost) )
```

(i) `sys_powd` gives the total power dissipation of the system. The assignment statement is:

```
context: sys_powd
assignment statement:
 $self = sum ( a in ArchComp ) (a.powd_rate *a.exec_time )
```

Constraints: There are two constraints in the conceptual model corresponding to the constraint equations *C3* and *C4*. The CSL expression for the constraint given in Eq. IV.26 is given by Figure 81. The CSL expression corresponding of the constraint in Eq. IV.27 is given by Figure 82.

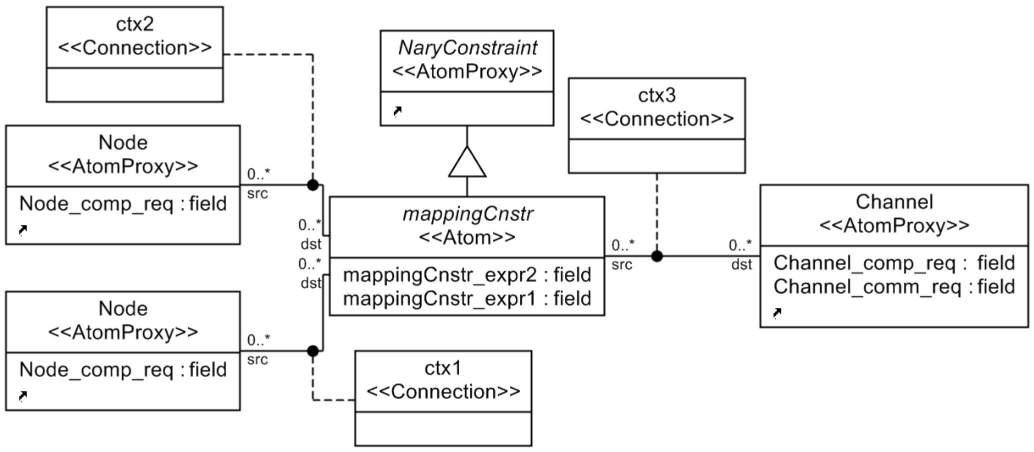


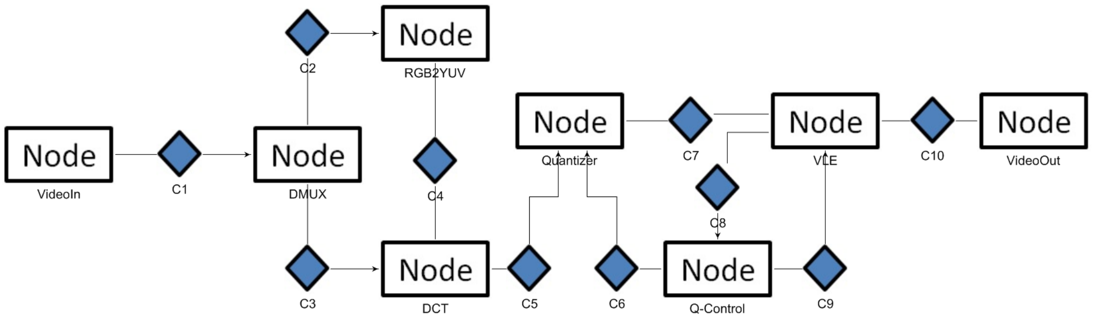
Figure 80: Ternary Constraint

```
context: $ctx1 = Node, $ctx2 = Node, $ctx3 = Channel
CSL expression:
forall$ctx1).forall$ctx2).forall($ctx3)
    ( (NodeMapsTo($ctx1,_) == NodeMapsTo($ctx2,_)
       /\ Connect($ctx1,$ctx3)
       /\ Connect($ctx2,$ctx3)
      ) -> ChannelMapsTo($ctx3,_) == NodeMapsTo($ctx1,_)
    )
```

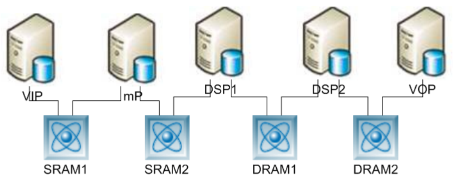
Figure 81: The CSL constraint definition for the mapping constraint

```
context: $ctx1 = Node, $ctx2 = Node, $ctx3 = Channel,
            $ctx4 = Processor, $ctx5 = Processor
forall$ctx1).forall$ctx2).forall($ctx3).forall$ctx4).forall$ctx5)
    ( (NodeMapsTo($ctx1, $ctx4)
       /\ NodeMapsTo($ctx2,$ctx5)
       /\ Connect($ctx1,$ctx3)
       /\ Connect($ctx2,$ctx3)
       /\  $ctx4 != $ctx5
      ) -> ChannelMapsTo($ctx3,_) in ( Reach($ctx1,_) intersect Reach(
         $ctx5,_))
    )
```

Figure 82: CSL Expression corresponding to Eq. IV.27



(a) MPEG application model



(b) Model Data

Figure 83: MPEG Model

IV.6.2 Model and Search

The conceptual model is translated to two solver specific models: (1) the CP Model that is processed using the Minizinc solver to retrieve valid solutions, and (2) MOP model that is processed by the evolutionary algorithm to retrieve Pareto-optimal solutions. We consider an MPEG encoder mapping problem model, where the application consists of an MPEG encoder application and architecture model consists of 5 processors and 4 memory components, as shown in Figure 83. The goal is to map the application model to the architecture model such that the objectives: cost, execution time and power consumption are minimized. We consider all objectives while solving the problem using MOP model whereas use one objective at a time for obtaining optimized mapping with respect to one objective.

We used a Intel Quad Core, 1.60 Ghz, 8GB RAM and 64 bit process to perform the experiments. We used the Flatzinc solver to the CP model of the problem and NSGA-2 (Non-dominated Sorting Genetic Algorithm) to solve the MOP model of the problem.

The CP model of the problem was used to retrieve valid design solutions. 593 valid design solutions were returned in around 1.5 seconds. Then the CP model was used to retrieve optimal results with respects to each of the three objectives. Figure 84 shows the solutions obtained by executing the MOP model and CP models. The initial results obtained by the execution of the MOP model did not return valid solutions. We manually initialized the MOP population used 6 extreme points in the CP model obtained by minimizing (maximizing) each of the objective variables. These solutions and a set of other random solutions are used to initialize the MOP model. The Pareto-optimal solutions obtained by executing the MOP model with with the initial population size of 10, the cross over probability of 0.1 and the mutation probability of 1. The population in the MOP model is initialized using the solutions from the CP model. Three Pareto optimal solutions are obtained after 150 generations in a single run. Point *A* in the figure has the minimum cost as well as power consumption but has the longest execution time for the application and point *C* has the least

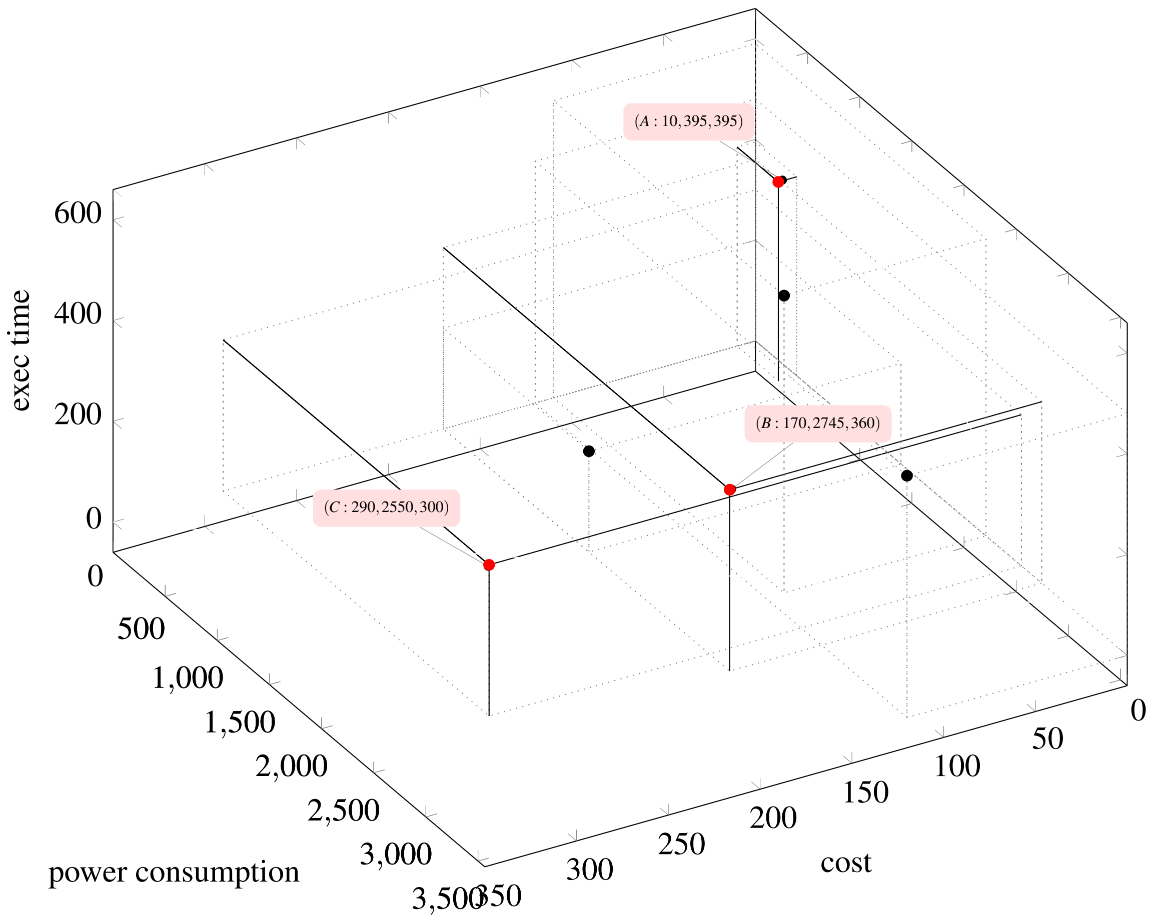


Figure 84: MOP output

execution time and the maximum cost. An additional B is another pareto optimal solution during the run.

IV.7 Summary

In this chapter we evaluated the scope of modeling and solving a wide range of problem using the generic framework. We first classified the commonly found classes of DSE problems into 6 categories, namely: resource allocation problems, selection problems, placement problems, routing problems, scheduling problems and configuration problems. We selected a case study from each of the 6 categories. The level of abstraction provided by ADSEL classes and the expressiveness of CSL enabled us to model problems from 5 out of 6 classes of problems (except configuration). Through the case studies presented in this chapter, we could highlight the following features of the framework:

- The framework can be used to model and solve problems from a wide *range of domains*. For example, embedded systems (Section IV.6), product-line engineering (Section IV.2), network-design (Section IV.1). This highlights the generality of the ADSEL classes, that can associate DSE aspects to any DSML, to create a model of the problem.
- The framework can model and solve a *range of problems*. Each of the classes presented in the classification has a unique set of requirements. The framework was used to model and solve problems from each of the classes, provided an analytical model for cost estimation was available. This highlights that the ADSEL concepts are rich enough to capture all the aspects of a DSE problem. Almost all aspects of ADSEL (except N-ary associations) were used in modeling the different problems. The notations supported by CSL was sufficient to capture the validity constraints in each of the problems. The support for multi-context constraints was especially useful in simplifying the specification of complex constraints, for example the CSL constraint presented in Figure 82, is comparatively simpler than an OCL expression for the same constraint that would require multiple navigations.
- The framework can be used to solve *different DSE problems in same domain*, for

example placement (Section IV.3) and routing problems (Section IV.4) from VLSI system design. The same DSML can be used as a basis to model both the problems.

Besides this, the framework can be also be used to solve a new variant of a DSE problem as the problem evolves over time. The scheduling problem (Section IV.5) highlighted this feature of the framework by modeling a generic scheduling problem. The conceptual model of the generic scheduling problem was reused to build a conceptual model for non-preemptive disjunctive scheduling by adding two simple constraints: *disjunctive* resource constraints and a *non-preemptive* task constraint. Further is shows modifying the conceptual model for the non-preemptive disjunctive problem to create a conceptual model for non-preemptive conjunctive scheduling problem. These modifications (additional, deletion) would be difficult in the existing DSE approaches, where a minor change in the requirements of the DSE problem, would trigger changes to the infrastructure (model compiler, classes, etc). This feature of the framework supports the domain experts in developing conceptual models for complex DSE problems. The domain-expert can start with a subset of the problem requirements and develop a conceptual model based on these requirements. Then at each stage, he can modify the conceptual model by addition of new elements (constraints, components, associations) until the entire problem is modeled. This simplifies the development of a conceptual model for a class of problems and in turn reduces the development time. Moreover, the domain-expert can incrementally test the correctness of the model at each stage using simple examples (Section IV.5.1.2 and Section IV.5.2.2).

The framework supports solutions of all problems modeled in the framework. Section IV.6 presents a complex case study where both CP and MOP models are used. The CP model is used to retrieve satisfactory as well as three solutions, corresponding to each of the goals, which the NSGA retrieves three Pareto-optimal solutions. The considered example had numerous constraints and therefore,the CP model performed better as it pruned out the infeasible solutions. Figure 84 shows 3 non-dominated solutions returned by NSGA algorithm by solving the MOP model of the problem. We used a generic penalizing technique for handling the constraints of the problem, where the value of a solution is indirectly proportional to the number of constraint violations. This help prune out invalid solutions and return valid Pareto-Optimal solutions.

IV.8 Observations

The evaluation resulted in the following observations:

Conceptual modeling: Given the variety of features supported by ADSEL, the domain expert needs to carefully choose the formulation. Although the IRL model corresponding to a conceptual model is auto-generated in the framework, the domain expert needs to aware of the the formulation choices he makes at the conceptual model level. For example, in case of the Placement problem for example, two formulations were tested. The *set formulation*, where the area of the circuit is represented as a set of slots $XSlots = 1..circuit\_length$ and the slots occupied by the modules is modeled as an association, $X\_Interval : Module \rightarrow \mathscr{P}(Slot)$, to represent the slots occupied by the module. The CP model generated from this conceptual model did not return a solution in a reasonable amount of time. The second model (presented in Section IV.3) replaced the associations with properties of the modules $Module\_X$ and modified the constraints. This generated a CP model that returned an optimal solution in 58 seconds for the example shown. Thus, a minor change in the modeling made a significant difference in the time to obtain a solution.

Constraint Specification: In order to maintain the generality of the CSL, global constraints supported by the constraint solvers (`alldifferent`, `cumulative`) are not included in the language. One possible approach of using global constraints is to incorporate them while translating the CSL constraint to solver specific model, in our case Minizinc. We consider this task in our future work.

Model Redundancy: The conceptual model is built by associating DSE aspect to the constructs in the DSML. These constructs (for example *Node*) can have properties (for example `name`). Most of these properties are not relevant for DSE, but a few can be relevant (`memory_req`). The current framework ignores the original properties, so in order to be considered for exploration, a reference to the original property has to be created. Similarly, the containment relationships in the original models are ignored and in order to be considered a reference is created. An example of this is the *Children* containment relationship in the Placement Problem. This is again redundant and should be avoided. This introduces a small extent of concept redundancy while creating the conceptual model.

Solver Support: The solutions obtained using the evolutionary algorithm are highly dependent on the initial population. In a constrained optimization problem, the initial population has to be carefully initialized, so that valid Pareto-optimal results can be retrieved at the end of a finite number of iterations of the algorithm. By default, the initial population of the population in the MOP model is randomly generated. Although both models are generated for a given problem, there is no exchange of information between the two. In the multi-objective resource allocation problem presented in Section IV.6, we initialized the population of the MOP model with a set of diverse solutions obtained from the CP model. This lead to better results in terms of validity of the solutions returned.

CHAPTER V

CONCLUSION

The core contribution of this dissertation is a generic environment for design space exploration that supports *modeling* and *solving* exploration problems from different domains. In order to use the framework, the domain-expert *meta-programs* (configures) the generic framework to work for a class of DSE problems in a given domain to create a domain-specific DSE environment. The resulting environment can then be used by domain-engineers to model and solve several instances of the DSE problem. Each problem model (instance) created using the framework is automatically translated to a solver-specific format. The solver is then executed on the input and solutions (if any exist) are returned to the domain-engineer. This allows domain-engineers to model and solve DSE problems without any particular background knowledge of the respective solver and its search technique.

There are two key features that make it possible to use the framework to *model* a wide range of exploration problems. First key feature is a set of abstract modeling concepts (ADSEL) that are generic enough to be applicable to DSE problems in a wide range of domains. These concepts include components, associations, constraints and the objectives. While configuring the framework selected concepts are specialized/inherited, possibly multiple times, to model domain-specific DSE concepts. The second key feature is an expressive specification language (CSL) used to annotate the model concepts with constraint definitions. The CSL is based on first order logic and set theoretic concepts that enable specification of a wide range of constraints from simple mathematical constraints to complex structural constraints. Further, the addition of syntactic sugar makes the expressions easier to read and write for the domain-experts. The framework supports constraints templates that can be created by the domain-experts as a part of the conceptual model. These constraints templates can be instantiated and used by the domain-engineers by plugging in values of the constraint parameters in the template.

The framework simplifies the use of search methods (constraint programming and evolutionary algorithm) to *solve* DSE problems. This is achieved by fully automating the process chain required for solving a high-level model of a DSE problem. The process chain includes solver-independent abstractions and a set of interpreters to automatically generate low-level mathematical models from the class diagram of the problem . The solver-independent abstraction (IRL) is used to decouple the structure of the problem from the optimization method used to solve it. Consequently, a single model can be solved using both mono and multi-objective optimization methods in a single environment. The choice of the solver is made by the domain-engineer depending on the number of objectives in the problem instance. This is more flexible than the existing approaches where the choice of the solver is made by the domain-expert, and is hard-coded for all the problem instances.

We presented a detailed evaluation of the framework in Chapter IV. The main criteria of evaluation was to verify the scope of our approach. Towards that goal, we first classified frequently found DSE problems into six categories, namely allocation problems, construction problems, placement problems, routing problems, scheduling problems and configuration problems. We selected 6 problems, one from each class and then attempted to model and solve these problems using the framework. We discussed in detail how one benchmark application from each class can be modeled and solved using the generic framework. This evaluation showed that the framework can be used to model and solve DSE problems from different domains, for example, embedded systems (Section IV.6), product-line engineering (Section IV.2), network-design (Section IV.1). The framework can also model and solve different DSE problems that belong to the same domain, for example placement (Section IV.3) and routing problems (Section IV.4) from VLSI system design.

Finally, the framework can be used to model and solve different variants of the same problem (IV.5.1). We used a multi-objective resource allocation problem to verify the solver support in the framework. The same problem is solved to retrieve valid solutions, optimal solutions with respect to a single solution, and Pareto-optimal solutions with respect to multiple objectives. is used to highlight the solver flexibility in the generic framework in Section IV.6. Unlike most existing DSE approaches where the framework comes with a preselected search method that is used for all instances of the problem, the generic framework supports selection of search method at the instance level. This is advantageous as different methods can be selected for different objectives.

V.1 Future Work

The generic framework can be extended in several ways.

1. *Improving the scope*: The configuration class represents a set of unconstraint optimization problems, which may or may not have an analytical function to calculate the cost. For example, cache configuration problems [133] have often been solved using simulation-based tools to estimate cost of design solutions. Moreover, the analytical models [60] found were too complicated to be implemented using CSL. In order to handle such problems, the framework has to be extended to provide an interface, so that the simulators can be used in conjunction with the CP and MOP models.

2. *Improving the solver support*: One of the most promising extensions will be to combine the MOP and CP models. A set of strategies to combine meta-heuristics with constraint programming are described in [123]. One of the hybrid strategies is to use MOP and CP models in a pipeline manner, where the CP model is used to produce a set of valid design points. This set is used to initialize the population in the MOP model, which then tries to improve the solutions in the set. Another strategy is to use the CP model in the recombination function, where the idea is to keep the common

elements of the parents. Using this incomplete solution, the CP model can be used to find the best offspring.

3. *Domain-expert assistance*: In the generic framework, a conceptual model is created for a class of DSE problem. This step is assumed to be performed by the domain-experts. In order to increase the usability of the framework, a minimum level of formulation support should be provided to the domain expert to create a new conceptual model. This will reduce the time required to configure the framework as we move from one domain to the other. One way of achieving this is to create conceptual model templates corresponding to each class of problems. These conceptual model templates can then be used by the domain-expert to create the conceptual model of his problem.

4. *Improving the Constraint Programming Model:* The generic framework has been tested with small problems. In order to make this approach more scalable, bottlenecks need to be identified. For example, one of the reasons for the lack of efficiency is the absence of global constraints. A possible approach to include global constraints is to include them in *IRL2CP* translation. At present the translation of the conceptual model of the problem to the is performed without any user intervention. This will One future direction of development will be to allow the domain expert to choose the formulation of constraints, such that the domain-expert can introduce global constraints, as well as a more efficient strategy for labeling the search variables. Currently a standard labeling strategy is to label the variables corresponding to the IRL functions, then sets and finally the variables of the basic type.

APPENDIX A

MAPPING PROBLEM CONCEPTUAL MODEL

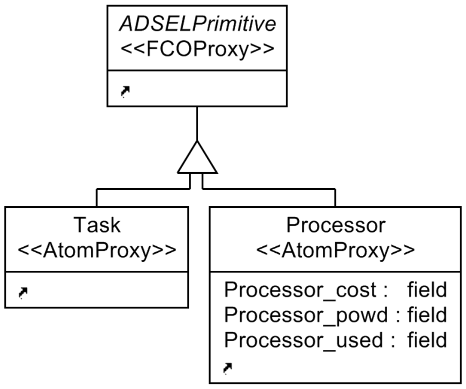


Figure 85: Components in the Conceptual Model of the Mapping Problem

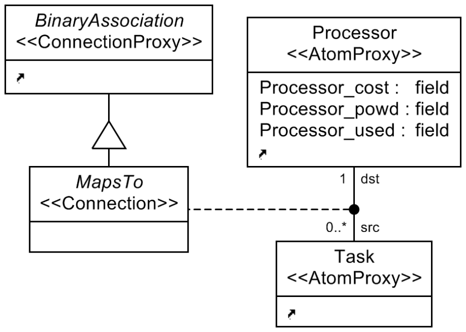


Figure 86: Association in the Conceptual Model of the Mapping Problem

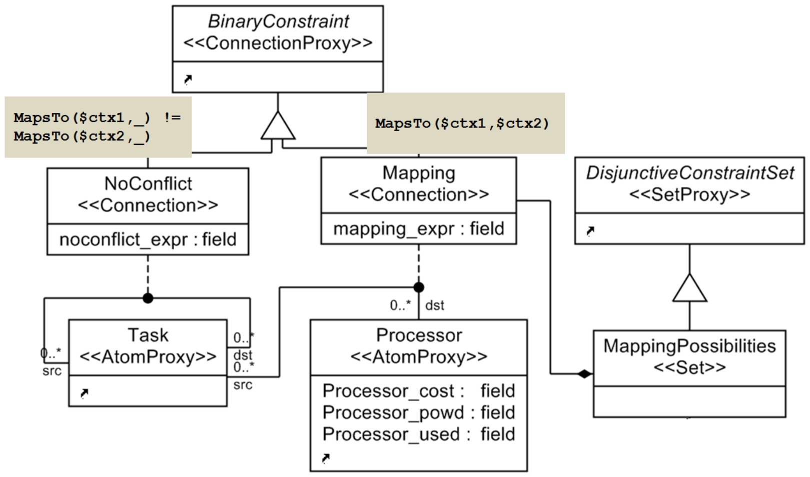


Figure 87: Constraints in the Conceptual Model of the Mapping Problem

APPENDIX B

CONSTRAINT SPECIFICATION LANGUAGE

B.1 Notation

We use the following syntactic convention to present the grammar:

- non-terminals are written in $\langle italic \rangle$ font
- terminals are written in `'type-writer'` font or <u>underlined</u> are terminals.
- keywords are written in **bold**. A letter is an alphabetic character.
- An *ID* is a string whose first character is a letter and the rest of its characters are alphanumeric. Identifier recognition is case sensitive.
- A *number* is any string whose elements are the numeric characters.
- $\{a\}$ denotes one, or several times the grammar segment a;
- $[a]$ stands for one or zero occurences of a;
- a^+ denotes the expression $a\{,a\}$;
- a^* denotes the expression $a\{.a\}$;

B.2 Grammar: CSL

⟨*Constraint*⟩	::=	⟨*ConstraintExpression*⟩ `;`
	|	⟨*Assignment*⟩ `;`
⟨*Assignment*⟩	::=	`$self =` ⟨*ArithmeticExpression*⟩
	|	`$self subset` ⟨*DomainExpression*⟩
	|	`$self <->` ⟨*BooleanExpression*⟩
	|	`$self.`⟨*property*⟩ `=` ⟨*ArithmeticExpression*⟩
	|	`$self.`⟨*property*⟩ `<->` ⟨*BooleanExpression*⟩
	|	`$self.`⟨*property*⟩ `= param(` ⟨*IntegerProperty*⟩`)`
	|	`$self.`⟨*property*⟩ `<-> param(` ⟨*BooleanProperty*⟩`)`
⟨*ConstraintExpression*⟩	::=	(`forall (` ⟨*VarDeclList*⟩$^+$`)`) *⟨*ConditionalExpression*⟩
	|	⟨*ConditionalExpression*⟩
⟨*ConditionalExpression*⟩	::=	⟨*BooleanExpression*⟩ ⟨*BinOp*⟩ ⟨*BooleanExpression*⟩
	|	`not(` ⟨*ConditionalExpression*⟩ `)`
	|	`(` ⟨*ConditionalExpression*⟩ `)`
⟨*BooleanExpression*⟩	::=	⟨*ArithmeticExpression*⟩ ⟨*RelOp*⟩ ⟨*ArithmeticExpression*⟩
	|	⟨*DomainExpression*⟩ ⟨*SetOp*⟩ ⟨*DomainExpression*⟩
	|	⟨*DomainElement*⟩ (`==` | `!=`) ⟨*DomainElement*⟩
	|	⟨*BooleanLiteral*⟩
	|	⟨*BooleanPropertyAccess*⟩
	|	⟨*DomainElementAccess*⟩

⟨*ArithmeticExpression*⟩	::=	*number*
	|	⟨*QuantifiedExpression*⟩
	|	⟨*ArithmeticExpression*⟩ ⟨*Mulop*⟩ ⟨*ArithmeticExpression*⟩
	|	**abs(** ⟨*ArithmeticExpression*⟩ **)**
	|	**bool2int (** ⟨*ConditionalExpression*⟩ **)**
	|	**card (** ⟨*DomainExpression*⟩ **)**
	|	⟨*IntegerPropertyAccess*⟩
⟨*QuantifiedExpression*⟩	::=	**sum (** ⟨*VarDecl*⟩$^+$**)** ⟨*ArithmeticExpression*⟩
	::=	**min (** ⟨*VarDecl*⟩$^+$**)** ⟨*ArithmeticExpression*⟩
	::=	**max (** ⟨*VarDecl*⟩$^+$**)** ⟨*ArithmeticExpression*⟩
⟨*DomainElementAccess*⟩	::=	⟨*DomainElement*⟩ **in** ⟨*DomainExpression*⟩
	|	⟨*AssociationID*⟩ **(** ⟨*Arg*⟩ **,** ⟨*Arg*⟩ **)**
⟨*DomainExpression*⟩	::=	{ }
	|	⟨*CompDomExpr*⟩ {⟨*BinOp*⟩ ⟨*CompDomExpr*⟩}
	|	⟨*AssocDomExpr*⟩
	|	⟨*ComponentID*⟩
	|	**sources (** ⟨*AssociationID*⟩ **)**
	|	**targets (** ⟨*AssociationID*⟩ **)**
	|	⟨*AssociationID*⟩ **(** ⟨*Arg*⟩ **, _)**
	|	⟨*AssociationID*⟩ **(_ ,** ⟨*Arg*⟩ **)**
VarDecl	::=	⟨*TempVar*⟩ **in** ⟨*CompDomExpr*⟩
	|	**≤ $ctx**[*number*]$^+$ **≥ in** ⟨*AssociationID*⟩
	|	**$ctx**[*number*]

$$\begin{array}{rcl}
\mathit{Arg} & ::= & \texttt{\$ctx}[\mathit{number}] \\
 & | & \texttt{\$self} \\
 & | & \langle \mathit{TempVar} \rangle \\
\\
\langle \mathit{IntegerPropertyAccess} \rangle & ::= & \texttt{\$ctx}[\mathit{number}]\texttt{.}\langle \mathit{IntegerProperty} \rangle \\
 & | & \texttt{\$self.}\langle \mathit{IntegerProperty} \rangle \\
 & | & \langle \mathit{TempVar} \rangle\texttt{.}\langle \mathit{IntegerProperty} \rangle \\
\\
\langle \mathit{BooleanPropertyAccess} \rangle & ::= & \texttt{\$ctx}[\mathit{number}]\texttt{.}\langle \mathit{BooleanProperty} \rangle \\
 & & \texttt{\$self.}\langle \mathit{BooleanProperty} \rangle \\
 & & \langle \mathit{TempVar} \rangle\texttt{.}\langle \mathit{BooleanProperty} \rangle \\
\\
\langle \mathit{SetOp} \rangle & ::= & \texttt{seq} \mid \texttt{sneq} \\
\langle \mathit{BinOp} \rangle & ::= & \texttt{union} \mid \texttt{intersect} \\
\langle \mathit{MulOpt} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \star \\
\langle \mathit{BoolOp} \rangle & ::= & \texttt{->} \mid \texttt{<->} \mid \backslash/ \mid /\backslash \\
\langle \mathit{RelOp} \rangle & ::= & \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \\
\langle \mathit{BooleanLiteral} \rangle & ::= & \texttt{true} \mid \texttt{false}
\end{array}$$

APPENDIX C

CM2IRL TRANSFORMATION

The *CM2IRL* translator applies a sequence of transformation rules to MM_{CM}, the conceptual model representing a class of DSE problems to transform it to M_{IRL}, an equivalent model in the IRL. There transformation rules can be divided into 5 set of transformation rules that are applied in sequence. Each set handles a concept in the conceptual model and transforms it a set of concepts in IRL. The components, associations and the global variables in the model form the data declaration part in the conceptual model. Therefore, these are transformed before the constraints and assignment expressions can processed. Finally, the objectives are processed at last. We briefly discuss each set of transformation rules in this section.

C.1 Component Translation Rules

Table 6: Excerpt of the translation rules for Component Class

	Component	**Contained Property**	**IRL Concept**
1	Primitive Component A		Parameter String Set iA
		Parameter Property p	Parameter Function $iA\_p : iA \to Z$
		Decision Property p	Decision Function $iA\_p : iA \to Z$
2	Output Primitive Component A		Decision Integer Set iA
		Parameter Property p	Parameter Integer iA_p
3	Ordered Primitive Component A		Integer Set iA
		Parameter Property p	Parameter Integer iA_p
4	Compound Component A		String Set iA
		Parameter Property p	Parameter Function $iA\_p : iA \to Z$
5	Abstract Component A		Compound Set $iA = iB \cup iC$
	with derived class B and C		Parameter Function $iA\_p : iA \to Z$
		Parameter Property p	Parameter Function $iA\_p : iA \to Z$

Table 6 summarizes the transformation rules used for generating IRL concepts from the conceptual model concepts. A component A ($Primitive$ or $Container$) by default is translated to a set of strings. This set basically consists of the names of the instances of the components. Any property of the component is translated to a binary function, where the domain is the string set generated corresponding to the parent component and the range depends on the type of the property. Table 6 shows translation rules for a property of integer type. A property of an output component is analogous to a static attribute of a UML Class. This property is not attached to any instance of the class but the class as whole and is therefore transformed to an input parameter of primitive type. An $Abstract$ component class in the conceptual model is refined to a compound set in the IRL model using Rule 5 in Table 6.

C.2 Association Translation

Table 7: Excerpt of the translation rules for Association Class

	Association Class	**Contained Property**	**IRL Concept**
1	Concrete Binary Association A from B to C		Parameter Function $iA : iB \rightarrow iC$
		Parameter Property p	Parameter Function $iA : iB \times iC \rightarrow Z$
		Decision Property p	Decision Function $iA : iB \times iC \rightarrow Z$
2	Abstract Binary Association A from B to C		Decision Function $iA : iB \rightarrow iC$

Table 7 shows the default transformation rules used for generating IRL concepts from the *Association Class* in the conceptual model. By default a *Binary Association* is translated to a function, where the domain and range are sets generated corresponding to the source and destination components. A property of the association class is refined to a function (Rule 1 in Table 7). All functions in the IRL model are total functions. The type of the

Table 8: Association to IRL Refinement Rules

Association multiplicity	IRL Concept	Informal Description
$\{0...*\} \rightarrow \{1\}$	$iA2B : iA \rightarrow iB$	total function
$\{1...*\} \rightarrow \{1\}$	$iA2B : iA \rightarrow iB$	surjective function
$\{0...1\} \rightarrow \{1\}$	$iA2B : iA \rightarrow iB$	injective function
$\{1\} \rightarrow \{1\}$	$iA2B : iA \rightarrow iB$	bijection function
$\{0...*\} \rightarrow \{1...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$	total function to non-empty subsets of *iB*
$\{0...*\} \rightarrow \{0...*\}$	$iA2B \subseteq iA \times iB$	relation
$\{1...*\} \rightarrow \{1...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$ $\wedge\, img(iA2B) = iB$	total function to non empty subsets which cover *iB*
$\{1...*\} \rightarrow \{0...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$ $\wedge\, img(iA2B) = iB$	total function to subsets of *iB* which cover *iB*
$\{0...1\} \rightarrow \{1...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$ $\wedge\, disjoint(iA2B)$	total function to non non-empty subsets of *iB* which do not intersect
$\{0...1\} \rightarrow \{0..*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$ $\wedge\, disjoint(iA2B)$	total function to subsets of *iB* which do not intersect
$\{1\} \rightarrow \{1...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB)$ $\wedge\, disjoint(iA2B)$ $\wedge\, img(iBA) = iB$	total function to non -empty subsets of *iB* which cover *iB* without intersecting
$\{1\} \rightarrow \{0...*\}$	$iA2B : iA \rightarrow \mathscr{P}(iB) \wedge$ $disjoint(iA2B)$ $\wedge\, img(iA2B) = iB$	total function to subsets of *iB* which cover *iB* without intersecting

generated function is decided by the source and destination cardinalities of the association in the conceptual model. For example, source cardinality of $\{1\}$ and destination cardinality of $\{1\}$ generates a bijective function. Table 8 summarizes the special cases of association class to function transformation rule based on the end point cardinalities of the binary association, where *iA* and *iB* are the sets generated corresponding to *ComponentA* and *ComponentB* respectively and *iA2B* is a function representing the association between *A* and *B*.

C.3 Global Variable Translation

Table 9: Excerpt of the translation rules for Global Variables

	Global Variable	IRL Concept
1	Parameter Global Variable *A*	Parameter Integer *iA*
2	Parameter Global Variable *A* is Singleton = false	Parameter String Set *iA* Parameter Function $iA\_value : iA \rightarrow Z$
3	Decision Global Variable *A*	Decision Integer *iA*
4	Decision Global Variable *A* is Singleton = false	Parameter String Set *iA* Decision Function $iA\_value : iA \rightarrow Z$

A *Global Variable* in the conceptual model is used to specify a property with global scope. Table 9 summarizes the rules for translation of global variables in the conceptual model to concepts in IRL. By default a global variable is translated to an input parameter or decision variable of predefined type. It is possible to have more than one instances of the same global variable, reflected by (`isSingleton=false`) property. In this case, a global variable basically represents an array and is refined to a set representing the instances of the global variable and a function to index to the value of each instance of the variable.

C.4 Assignment Statement Translation

An assignment statement in the conceptual model is used to assign values to the output components, component properties and decision global variables. It is also used to constrain the value of an inherited property. Table 10 summarizes the rules for translating the assignment statements to IRL concepts. The translation of an assignment statement depends on the context. For example, assigment of a parametric property p of Component D is translated to the definition of function $D\_p : iA \rightarrow Z$, if D owns the property (Rule 1). If p is an inherited property and belongs to the base Component B instead then it is translated to a conditional definition of function $B\_p : iA \rightarrow Z$, as described in (Rule 2). In case the

Table 10: Excerpt of the translation rules for Assignment Statement

	Context	IRL Concept
1	**def**: A_{CSL}($self) **context**: Component B, Property p	**def**: $B\_p(b) = A_{pCSL}(b)$, `if` $b \in iB$ **context**: Function $iB\_p \rightarrow Z$
2	**def**: A_{CSL}($self) **context**: Derived Component D Base Component B, Property p	**def**: $B\_p(b) = A_{pCSL}(b)$, `if` $b \in iD$ **context**: Function $B\_p : iB \rightarrow Z$
3	**def**: param($self.p) **context**: Derived Component D Base Component B, Property p	**def**: $B\_p(b) = D\_p(b)$, `if` $b \in iD$ Parameter Function $D\_p : iD \rightarrow Z$ **context**: Function $B\_p : iB \rightarrow Z$
4	**def**: A_{CSL}($self) **context**: Output Component B	**def**: $iB = A_{pCSL}$ **context**: iB
5	**def**: $self=$A_{CSL}$ **context**:Global Variable V	**def**: $iV = A_{pCSL}$ **context**: iV

6	**def**: A_{CSL}($self)
	context: Global Variable V
	isSingleton=false
		def: $iB = A_{pCSL}$
		context: iB

inherited property is constraint to a single value using **param**, then a new parameter function is generated. This is used to indicate that the value of the inherited property is to be read from the design space model.

C.5 Constraints

Table 11 summarizes the rules for translating a constraint class in the conceptual model. If the constraint class is abstract, then it is considered a global constraint which is applicable to all instances (or at least one unknown instance) of the context. This constraint is translated to a constraint in the IRL model otherwise it is translated to a predicate. The orginal constraint definition is a boolean expression, is rewritten in terms of the variables in the IRL model.

C.5.1 CSL to pCSL Rewrite Rules

CSL provides syntactic sugar to write constraint definitions in terms of components, associations and their properties. The syntactic sugar is removed to restate the CSL definitions

Table 11: Excerpt of the translation rules for Constraints

	Constraint	**IRL Concept**
1	Concrete Constraint *C* **context:** Component *A*	Predicate *iC* Argument Set *iA*
2	Abstract Constraint *A* **context:** Component *A*	Constraint *iC* Argument Set *iA*

in terms of sets, functions, relations and basic integer and boolean variables in the IRL model. We will refer to this concise language as ***processed*** CSL (**pCSL**). The processing step involves introduction of variables to replace context keywords. We describe a set of processing rules perform additional processing, written in the format:

$$LHS \Rightarrow RHS \mid C$$

Each processing rule consists of an LHS pattern from the CSL that is refined to a simplified from in pCSL, captured by the RHS if the conditions are satisfied. The general form of a CSL definition is given by:

$$[\{\langle Quantifier\rangle\ \underline{(}\langle Quantifying-Variable\rangle\underline{)}\}]\ (\langle BooleanExpression\rangle); \tag{C.1}$$

Rule 1 *Quantifying Variable Processing Rule* :

```
forall($ctx[Number])  ⇒  forall(<cntxt-var> in <set-name>)
```

where a unique `Number` is used to refer to each context component and '`set-name`' is the name of the *irlSet* in IRL model generated from the the context component of the constraint in the conceptual model, and '`cntxt-var`' is a variable of the basic type introduced to access the elements of the *irlSet*. The type of '`cntxt-var`' depends on the type of the set.

Rule 2 Component Property Access

```
$ctx[N].<propID>  ⇒  <propFuncID>(<cntxt-var>)
```

where `N` is a natural number to index the context component, ‘`propFuncID`’ is the name of the *irlFunction* in IRL model created by transformation of the component property, and ‘`cntxt-var`’ is the integer variable introduced by processing of the formal parameter.

Rule 3 Association Instance Reference

```
<BinAssoc>(<context1>,<context2>)
  ⇒  <binFuncID>(<cntxt-var1>) ==  <cntxt-var2>
   |  where <BinAssoc> is refined to a binary function
  ⇒  <binRelationID>(<cntxt-var1>,<cntxt-var2>)
   |  where <BinAssoc> is refined to a binary relation
```

where `n1`, `n2` are natural numbers that uniquely index the constraints and ‘`cntxt-var1`’ and ‘`cntxt-var2`’ are temporary variables introduced corresponding to these contexts using Rule 1. If the binary association ‘`BinAssoc`’ is refined to a function ‘`binFuncID`’, then the instance access is processed to boolean expression of the form ‘`f(a) == b`’, to check the existence of the association instance. If the binary association ‘`BinAssoc`’ is refined to a relation ‘`binRelationID`’, then the instance access is processed to replace the context keywords with the corresponding context variables.

Rule 4 Association Instance Property Access

```
<BinAssoc>(<context1>,<context2>).<propID>

  ⇒  <propFuncID>(<cntxt-var1>, <cntxt-var2>)
```

where ‘`cntx-var1`’ and ‘`cntx-var2`’ are the variables introduced corresponding to these contexts using Rule 1.

Rule 5 Association Forward Navigation

```
<BinAssoc>(<context1>, _ )
⇒   <binFuncID>(<cntxt-var1>)
  | where <BinAssoc> is refined to a binary function
⇒   <binRelationID>(<cntxt-var1>,_)
  | where <BinAssoc> is refined to a binary relation
```

where '`cntx-var`' is the variable introduced by processing using Rule 1. The forward navigation of a binary association returns a set of all the target instances associated to the source instance. If the binary association is refined to a function, then forward navigation is refined to a functional notation : `f(a)`. If the binary association is refined to a relation, then forward navigation is processed to replace the context keyword with the variable.

Rule 6 Association Backward Navigation

```
<BinAssoc>( _,<context2>)
⇒   <binFuncID>(~ <cntxt-var2>)
  | where <BinAssoc> is refined to a binary function
⇒   <binRelationID>(_ ,<cntxt-var2>)
  | where <BinAssoc> is refined to a binary relation
```

where '`binFuncID`' is the name of the function generated from the '`BinAssoc`'. The reverse navigation of the association returns a set of all the source instances associated to a target instance by '`BinAssoc`'. If the binary association is refined to a function then reverse navigation is reduced to a functional notation. If the binary assocation is refined to a relation then the reverse navigation is processed to replace the context keyword with the variable.

Rule 7 Association Sources

```
          sources(<BinAssoc>)
      ⇒   <setID>
```
| where `<BinAssoc>` is refined to a binary function

```
      ⇒   projection(<binRelationID>, <setID>)
```
| where `<BinAssoc>` is refined to a binary relation

where '`setID`' is the set created by transformation of association source context component of the association. The method returns a set of source instances associated with any target component instance using the '`BinAssoc`' association. If the association is transformed to a function, this means that the entire source set is returned because we consider only total functions. In case the association is refined to a relation, this returns projection of the relation over source set.

Rule 8 Association Targets

```
          targets(<BinAssoc>)
      ⇒   image(<binFuncID>)
```
| where `<BinAssoc>` is refined to a binary function

```
      ⇒   projection(<binRelationID>, <setID>)
```
| where `<BinAssoc>` is refined to a binary relation

where the method'**`targets`**' returns a set of target component instances that are connected to any instance of the source component using the binary association '`BinAssoc`'. If the association is refined to a function, then the method call is replaced by '**`image`**' call on function '`binFuncID`'. In case the association is refined to a relation, this method call returns the projection of relation over the destination set.

APPENDIX D

IRL2CP TRANSLATOR

This section describes the rules for translating the IRL data types and constraint expressions to corresponding Minizinc data types and expressions. The translation rules are generic and can be reused for translation of an IRL model to any other constraint programming language without much effort. The translation rules are grouped into two sets: data type translation rules and constraint expression translation rules.

Table 12: Translation rules for Basic Type

	IRL Concept	**CP Concept**
1	Input Parameter ⟨Type⟩ p	⟨Type⟩ p
2	Decision Variable ⟨Type⟩ p	var ⟨Type⟩ p
3	Type	Type
4	Set *A*	Interval Set
5	Compound Set *A* containing Set B and Set C	Inteval Set A $\|A\|= \|B\|+\|C\|$

D.1 Data Type Translation Rules

Most CPLs (including Minizinc) are over finite domains and support input variables of '**int**', '**bool**', '**string**' and '**float**' types. An output variable is strictly integer or boolean. The compound data types supported by most Minizinc are: '**array**' and '**set**' of the basic types. This section presents rules for translating IRL data types to Minizinc data types.

Basic Types. Table 12 summarizes the simple translation rules for basic data types and sets. A set A in the IRL model is translated to an interval set A_{cp} in the Minizinc, such that the set consists of integers from '**[1..|A|]**'. An index map is created associating each element of the set with A with a unique integer in the range. For example '*Set A*' is translated to '**set of int A = [1..num_A]**', where '**num_A**' is the cardinality of A.

Compound Types. Relations and Functions in the IRL model are not directly supported by Minizinc. They are translated to arrays in the Minizinc according to the translation rules summarized in Table 13. A relation R is translated to a 2 dimensional boolean array, where '`1`' represents membership in the relation and '`0`' represents the absence. The array is indexed by A and B.

All function in IRL are total functions, such that every element in the domain set is mapped

Table 13: Translation rules for Compound IRL Types

	IRL Concept	**CP Concept**
1	Relation $R \subseteq A \times B$	2-D Array $RD2[A,B]$ st. $RD2[A,B] \in \{0,1\}$
2	Function $F : A \rightarrow B$	1-D Array $FD1[A]$ st. $FD1[A] \in B$
3	Surjective Function $F : A \rightarrow B$	1-D Array $FD1[A]$ st. $FD1[A] \in B$ $\bigwedge\ \forall b \in B \cdot \exists a \in A \cdot FD1[a] = b$
4	Injective Function $F : A \rightarrow B$	1-D Array $FD1[A]$ st. $FD1[A] \in B$ $\bigwedge\ \forall a,b \in A.a \neq b \rightarrow FD1[a] \neq FD1[b]$
5	Bijective Function $F : A \rightarrow B$	1-D Array $FD1[A]$ st. $FD1[A] \in B$ $\bigwedge\ \lvert A\rvert = \lvert B\rvert$

to one element in the range. A binary function is translated to a 1-D array in Minizinc, such that each element of the array belongs to the range of the function. This representation of function is valid because of the interval representation of the set variables. Translation of a surjective function generates an additional constraint stating that each element of the range is mapped to an element of the domain. Similarly, translation of an injective function generates an additional constraint stating that two distinct elements of the function domain are mapped to distinct elements of the range. A *bijective* function is an injective function with an additional constraint that the cardinalities of the sets *iA* and *iB* are the same.

Table 14: Translation rules for Compound IRL Types

	IRL Concept	**CP Concept**
1	Function $F : A \rightarrow \mathscr{P}(B)$	1-D Array $FD1[A]$ st. $FD1[A] \subseteq B$
2	Surjective Function $F : A \rightarrow \mathscr{P}(B)$	1-D Array $FD1[A]$ st. $FD1[A] \in B$ $\bigwedge\ \forall b \in B \cdot \exists a \in A \cdot b \in FD1[a]$
3	Function $F : A \times B \rightarrow C$	2-D Array $FD2[A,B]$ st. $FD2[A,B] \in C$
4	Surjective Function $F : A \times B \rightarrow C$	2-D Array $FD2[A,B]$ st. $FD2[A,B] \in C$ $\bigwedge\ \forall c \in C \cdot \exists a \in A, \exists b \in B \cdot c \in FD2[a,b]$
5	Injective Function $F : A \times B \rightarrow C$	2-D Array $FD1[A,B]$ st. $FD1[A,B] \in B$ $\bigwedge\ \forall a_1, a_2 \in A, b_1, b_2 \in B\cdot$ $(a_1 \neq a_2 \vee b_1 \neq b_2)$ $\rightarrow FD2[a_1,b_1] \neq FD2[a_2,b_2]$
6	Function $F : A \rightarrow B \times C$	3-D Array $FD3[A,B,C]$ st. $FD3[A,B,C] \in \{0,1\}$
7	Surjective Function $F : A \rightarrow B \times C$	3-D Array $FD3[A,B,C]$ st. $FD3[A,B,C] \in \{0,1\}$ $\bigwedge\ \forall b \in B, \forall c \in C \cdot \exists a \in A \cdot FD3[a,b,c] = 1$
8	Injective Function $F : A \times B \rightarrow C$	3-D Array $FD3[A,B,C]$ st. $FD3[A,B,C] \in \{0,1\}$ $\bigwedge\ \forall a_1, a_2 \in A, \exists b_1, b_2 \in B, \exists c_1, c_2 \in C\cdot$ $(a_1 \neq a_2 \rightarrow FD1[a_1,b_1,c_1] = 1$ $\wedge FD1[a_2,b_2,c_2] = 1$ $\wedge (b_1 \neq b_2 \vee c_1 \neq c_2)$

D.2 pCSL Constraints to Minizinc Constraints

Minizinc supports constraints and predicates to express relationship between the different variables. These constraints and predicates are generated by translation of the IRL constraint classes and parsing the contained pCSL expressions to generate Minizinc constraints written in terms of the variables in the CP.

We use conditional rewrite rules, here written as follows:

$$L \longrightarrow\ R \mid C \tag{D.1}$$

meaning that, if condition C holds, then expression L is rewritten into R. The constraint definition is divided into 5 sections:

Table 15: pCSL Quantified Expressions to Minizinc Constraints

	pCSL		Minizinc

1	**`forall`**`( <i,j> `**`in`**` F)`
	`(P(i,j))`
		$\longrightarrow$
		
		|
			`forall``(i `**`in`**` D1, j `**`in`**` D2)`
			`((FD1[i]==j) -> P'(i,j))`
			`)`
			where $F : D1 \to D2$

2	**`forall`**`( <i,j> `**`in`**` F)`
	`(P(i,j))`
		$\longrightarrow$
		
		|
			`forall``(i `**`in`**` D1, j `**`in`**` D2)`
			`((j in FD1[i]) -> P'(i,j))`
			`)`
			where $F : D1 \to \mathscr{P}(D2)$

3	**`forall`**`( <i,j> `**`in`**` R)`
	`(P(i,j))`
		$\longrightarrow$
		
		|
			`forall``(i `**`in`**` D1, j `**`in`**` D2)`
			`((RD2[i,j]) -> P'(i,j))`
			`)`
			where $R : D1 \times D2$

D.2.1 Translating pCSL Quantified Expressions

A quantified expression in pCSL is of the following form: The constraint definitions in the IRL model are pCSL expressions in terms of sets, functions and relations. A pCSL constraint in the IRL model has the following form:

$$[\{\langle Quantifier\rangle\ \underline{(}\langle identifier\rangle\ \textbf{in}\ \langle SetAtom\rangle)\underline{)}\}]\ (\langle BooleanExpression\rangle); \qquad \text{(D.2)}$$

where the $Quantifier$ represents either '`forall`' or '`exists`', the *Variables* is a list of identifiers that range over *SetAtom* and whose scope is limited to the *BooleanExpression*. Table 15 summarizes the parser rules for parsing pCSL constraints to generate expressions in Minizinc. The rules for parsing universally quantified expressions are also applicable to the existentially quantified expressions. The parsing is dependent on the quantifying-variables. In pCSL, functions and relations are seen as a set of tuples. Therefore, the quantifying variables can be expressed as tuple in a function or relation. The predicate $P(i, j)$ is applicable only to the members of the function or relation. Therefore, the predicate parsed to an implication, where $P'(i, j)$ is the parsed predicate corresponding to $P(i, j)$ and is true only when $\langle i, j\rangle$ is a member of F, that is $FD1[i] == j$, where $FD1$ is a 1D array generated from function $F : A \to B$. Similar rule applies for parsing a quantified expression where the domain is a relation.

D.2.2 Translating pCSL Set Expressions

The set expression in pCSL are translated to set expressions in Minizinc. Table 16 summarizes the translation rules for translating the set expressions. The set atoms in pCSL include the sets retrieved by projection on functions and relations. Minizinc supports binary set operators '**union**' and '**intersect**', therefore there is one-to-one mapping from the pCSL

Table 16: pCSL Quantified Expressions to Minizinc Constraints, where D1 is a pCSL set expression and D1' is the corresponding parsed set expression in Minizinc

	pCSL		Minizinc
1	`D1 union D2`	$\longrightarrow$ |	`D1' union D2'` where $D1$ and $D2$ are set expressions
2	`D1 intersect D2`	$\longrightarrow$ |	`D1' intersect D2'` where $D1$ and $D2$ are set expressions
3	`F(a)`	$\longrightarrow$ |	`FD1[a]` where $F : D1 \rightarrow \mathscr{P}(D2)$

4	`F(∼a)`	$\longrightarrow$	`{let set of int: FCOIMG}`
			 `{ function_reverse_image(FD1,FCOIMG,a)}`

5	`image(F)`		`let set of int: FIMG`
			 `function_image(FD1, FIMG)`

6	`R(a, _)`	$\longrightarrow$	`{let set of int: RIMG }`
			 `{relation_image(RD2, RIMG, a)}`

7	`R(_, b)`		`let set of int: RCOIMG`
			 `relation_reverse_image(RD2, RCOIMG, b)`

8	`projection(R, A)`		`let set of int: RIMG`
			 `relation_projection(RD2, RIMG, AINDX)`

set expressions to Minizinc set expressions. Minizinc also supports set literals '**{}**'. We have built a library of minizinc predicates to simplify the translation of complex set expressions that involve functions and relations. For example, '`image(F)`', for a function $F : A \rightarrow \mathscr{P}(B)$ represents a set $FIMG$, such that $FIMG \subseteq B$. This expression is translated to a predicate '`function_image`' and a temporary set variable '`FIMG`'.

D.2.3 Translating pCSL Boolean Expressions

Minizinc support binary and unary operators like '$\wedge$', '$\vee$', '**not**', '$\rightarrow$' and '$\leftrightarrow$'. Therefore, the translation of such boolean expressions is trivial. Minizinc supports overloaded equality and inequality operators for set expression, which was not the case in pCSL. Table 17 summarises the translation of set operators that create boolean expressions.

Table 17: pCSL Boolean Expressions to Minizinc

	pCSL		Minizinc
1	`D1 sneq D2`	$\longrightarrow$ |	`D1' != D2'` where *D*1 and *D*2 are set expressions
2	`D1 seq D2`	$\longrightarrow$ |	`D1' == D2'` where *D*1 and *D*2 are set expressions
3	`d1 in D1`	$\longrightarrow$ |	`d1 in D1'` where *d*1 is set element and *D*1 is a set expression

D.2.4 Translating pCSL Arithmetic Expressions

Minizinc supports the basic binary and unary operators on the arithmetic expression. It also supports quantified sum operator. The only significant rules involve the translation of quantified sum when the quatifying variable is a tuple over a function or a relation, shown in Table 18. In this case a boolean condition is added to ensure the arithmetic expression is calculated only over the elements of the function or relation.

Table 18: pCSL Arithmetic Expressions to Minizinc

	pCSL		Minizinc

1	`sum( <i,j> in F)`
	`  (P(i,j))`
		$\longrightarrow$	`sum(i in D1', j in D2')`
			`((bool2int (FD1[i]==j))`
			` * X'(i,j))`
			`)`

2	`sum( <i,j> in R)`
	`  (X(i,j))`
		$\longrightarrow$	`sum(i in D1', j in D2')`
			`(bool2int(RD2[i,j]) * X'(i,j))`
			`)`

3	`card( F)`	$\longrightarrow$	`sum(i in D1', j in D2')`
			`(bool2int(RD2[i,j]))`
			`)`

D.2.5 Translating pCSL Assignment Statements

The assignment statements and function definition in the IRL model are translated to equality constraints in the Minizinc. A function definition of the form $F(p) = x$, where $p \in domain(F)$, is translated to the constraint expression- '`forall(p in D)(A'(p))`'.

D.3 Predicate Library

```
predicate function_image(array[int] of var int: x, var set of int: s,
                                 var set of int: t) =
   assert(ub(s) subset index_set(x),
      "range: upper bound of 's' must be a subset of the index set of 'x
         '",

      % All values in 's' must map to a value in 't'.
      forall(i in ub(s)) (
         i in s -> x[i] in t
      ) /\
      % All values in 't' must be mapped from a value in 's'.
      forall(i in ub(t)) (
         i in t -> exists(j in ub(s)) ( j in s /\ x[j] == i )
      )
   );

predicate inverse_image( array[int] of var int:F, var set of int: X, int
   :k)=
  forall(i in ub(X))(
       (i in X) <-> (F[i] == k)
  );


predicate bij_func_inverse_image( array[int] of var int:F, var int: X,
   int:Y)=
  forall(i in index_set(F))(
       (i = X) <-> (F[i] == Y)
  );

predicate relation_image(array[int, int] of var int: r, var set of int:
   img, int: a) =
     forall(i in index_set_2of2(r)) (
       (r[a,i] == 1) <-> (i in img)
     ) ;

predicate relation_reverse_image(array[int, int] of var int: r, var set
   of int: img, int: a) =
     forall(i in index_set_1of2(r)) (
       (r[i,a] == 1) <-> (i in img)
     ) ;

predicate relation_domain(array[int, int] of var int: r, var set of int:
    img) =
      forall( i in index_set_1of2(r))(
            i in img <-> exists(j in index_set_2of2(r))(r[i,j] == 1)
      ) ;

predicate relation_range(array[int, int] of var int: r, var set of int:
   img) =
      forall( j in index_set_2of2(r))(
            j in img <-> exists(i in index_set_1of2(r))(r[i,j] == 1)
      ) ;
```

```
predicate binary_relation_projection(array[int, int] of var int: r, var
    set of int: img, int:a) =
      if (a==1) then (
            forall( i in index_set_1of2(r))(
                  i in img <-> exists(j in index_set_2of2(r))(r[i,j] ==
                     1)
            )
      ) else(
            forall( j in index_set_2of2(r))(
                  j in img <-> exists(i in index_set_1of2(r))(r[i,j] ==
                     1)
            )
      )endif;
```

APPENDIX E

IRL2MOP TRANSLATOR

This section describes the rules for translating the IRL data types and constraint expressions to Java types and expressions. The translation rules are grouped into two sets: (1) data type translation rules, and (2) constraint expression translation rules.

E.1 pCSL Constraints to Java Functions

A pCSL constraint is a boolean expression that is translated to a Java function that returns a boolean value. Similarly, an arithmetic expression in pCSL is translated to a Java function with integer return value. The assignment statements in the IRL model are refined to function that include Java statements assigning values to the global variables (parameters or decision variables).

E.1.1 Translating pCSL Quantified Expression to Java Functions

The constraint definitions in the IRL model are pCSL expressions in terms of sets, functions and relations. A pCSL constraint in the IRL model has the following form:

$$[\{\langle Quantifier\rangle \ \underline{(}\langle identifier\rangle \ \mathbf{in} \ \langle SetAtom\rangle\underline{)}\}] \ (\langle BooleanExpression\rangle); \tag{E.1}$$

where the *quantifier* represents either '`forall`', '`exists`', the *identifier* is a list of identifiers that range over *SetAtom* and whose scope is limited to the *BooleanExpression*.

(1) If the quantifying variable is an element of a set '`d1 in D1'`' and the constraint holds for all the elements of the function, then the following rule is used to rewrite the constraint in imperative language, where '`A(d1)`' is a boolean expression in pCSL. '`K`' is the type of the set `D1'` which can be String, Integer, Boolean or Double.

```
forall( s in S) ( A(s) )   ⟶   boolean curr = true;
                               for(String d1:  D1'){
                                if( A(s) && curr){
                                curr = true;
                                }else{
                                curr = false;}
                               };
```

(2) If the quantifying variable is a tuple in a function $F : S \to T$ and the constraint holds for all the elements, then the following rule is used to rewrite the constraint in Java where K is the domain set type and V is the range set type.

```
forall(<d1,d2> in F)  (A(d1,d2))

                   →  boolean curr = true;
                      for(<K>d1:F.getDomain()){
                       for(<V>d2:F.getRange()){
                       if( A(d1,d2)
                       || !(F.getImage(d1).equals(d2))
                       &&(curr)){
                       curr = true;
                       }else{
                       curr = false;}
                      }};
                   |  where F : A → B
```

(3) Similar translation rule is used for translation of quantified expression where the identifier is a tuple in a relation $R \subseteq S \times T$.

```
forall(<d1,d2> in R)  (A(d1,d2))
                   →  boolean curr = true;
                      for(<K>d1:R.getProjection(1)){
                       for(<V>d2:R.getProjection(2)){
                       if( A(d1,d2)
                       || !(R.getImage(d1).equals(d2))
                       &&(curr)){
                       curr = true;
                       }else{
                       curr = false;}
                      }};
                   |  where R : A × B
```

E.1.2 pCSL Boolean Expression to Java Functions

Each boolean expression is translated to a function with boolean return value. Table 19 shows a summary of the translation rules used, where Boolean expressions are denoted with uppercase letters from the beginning of the alphabet (*A*, *B*, *C*, etc.); integer expressions are denoted with lower-case letters from the ending of the alphabet (x, y, z, etc). Set expressions are denoted by uppercase alphabets (*D*1, *D*2, etc.) and set elements are denoted by lowercase letters (*d*1, *d*2, etc.)

Table 19: pCSL Boolean Expressions to Java Code

	pCSL		Java

1	A	$\longrightarrow$	if(A){
			return true
			}else { return false }

2	**not**(A)	$\longrightarrow$	if(!A){
			return true
			}else { return false }

3	A **->** B	$\longrightarrow$	if(!(A) || (B))
			{return true}
			else { return false }

4	A **<->** B	$\longrightarrow$	if((!(A) && !(B)) || (A && B))
			{return true}
			else { return false }

5	x **==** y	$\longrightarrow$	int x′ = x;
			int y′ = y;
			if((x′ = y′){
			return true}
			else { return false }

6	d1 == d2	$\longrightarrow$	String d1′ = d1;
			String d2′ = d2;
			if((d1′.equals(d2′)){
			return true}
			else { return false }

7	d1 != d2	$\longrightarrow$	String d1′ = d1;
			String d2′ = d2;
			if((not(d1′.equals(d2′))){
			return true}
			else { return false }

8	D1 seq D2	$\longrightarrow$	HashSet<String> D1′ = D1;
			HashSet<String> D2′ = D2;
			if(D1′.equals(D2′)){
			return true}
			else { return false }

E.1.3 pCSL Set Expression to Java Functions

Java supports HashSets that are used to represent the IRL sets. A library of classes containing template definition for functions and relations was developed to simplify the translation of set expressions. For example, '**`image`**`(F)`', for a function $F : A \rightarrow \mathscr{P}(B)$ represents a set $FIMG$, such that $FIMG \subseteq B$. This expression is translated to a predicate '`F.getImage()`'. Table 20 presents an excerpt of the translation rules translation of set expressions.

Table 20: pCSL Set Expressions to Java Code

	pCSL		Java
1	`D1 union D2`	$\longrightarrow$	`HashSet<String> D1' = D1;` `HashSet<String> D2' = D2;` `D1'.addAll(D2')` `return D1';`
2	`D1 intersect D2`	$\longrightarrow$	`HashSet<String> D1' = D1;` `HashSet<String> D2' = D2;` `D1'.retainAll(D2');` `return D1';`
3	`F(a)`	$\longrightarrow$	`F.getImage(a)`
		|	where $F : A \rightarrow \mathscr{P}(B)$
4	`F(~a)`	$\longrightarrow$	`F.getCoImg(a)`
5	`R(a, _)`	$\longrightarrow$	`R.getImage(a)`
6	`R(_, b)`	$\longrightarrow$	`R.getCoImage(b)`
7	`projection(R, D1)`	$\longrightarrow$	`R.getProjection(1)`
		|	where $R \subseteq D1 \times D2$
8	`image(F)`	$\longrightarrow$	`F.getImage()`

E.1.4 pCSL Arithmetic Expression to Java Functions

Java supports all arithmetic operators used in the constraint expressions. Only significant translation required rules are the ones used for translation of quantified sum.

Table 21: pCSL Arithmetic Expressions to Java Code

	pCSL	Java
1	`card(D)`	
	$\longrightarrow$	`int R;`
		`HashSet D' = D;`
		`R = D'.size();`
2	`sum( s in S) (x(s))`	
	`(x(s))` $\rightarrow$	`int tsum = 0;`
		`HashSet<String> S'= S;`
		`forall(String s:  S'){`
		` int x'= x;`
		` tsum = tsum + x';`
		`}`
		`return tcost;`
3	`sum( <i,j> in F)`	
	`x(i,j)` $\rightarrow$	`int tsum = 0;`
		`HashSet<String> S1'= F.getDomain();`
		`HashSet<String> S2'= F.getRange();`
		`forall(String s1:  S1'){`
		` forall(String s2:  S2'){`
		` if(F.contains(s1,s2)){`
		` int x'= x;`
		` tsum = tsum + x';`
		` }`
		`return tcost;`

APPENDIX F

SONET RESOURCE ALLOCATION PROBLEM

```
%--------------INPUT-------%

int: Ring_capacity;
int: Ring_cardinality;
var set of 1..Ring_cardinality: Ring;

int: num_Node ;
set of int: Node = 1..num_Node ;

array[Node,Node] of 0..1: Edge ;

%-----------OUTPUT--------%
array[1..Ring_cardinality,Node] of var 0..1: RingNode::is_output;
var int: Z;


%---------CONSTRAINTS--------------%
%
%-------constraint 2-----%
constraint
forall(n1 in Node, n2 in Node)(
      let{
        var set of 1..Ring_cardinality: COIMG1,
        var set of 1..Ring_cardinality: COIMG2
      } in
      Edge[n1,n2] == 1 ->
      ( relation_reverse_image(RingNode, COIMG1, n1)
        /\ relation_reverse_image(RingNode, COIMG2, n2)
        /\ ( COIMG1 intersect COIMG2 != {}))
);
%
%-------constraint 3-----%
constraint
forall(ctx1 in ub(Ring))(
      let{
            var set of 1..num_Node: IMG
      } in
      relation_image(RingNode, IMG, ctx1)/\
      card(IMG) <= Ring_capacity
)
;

%------assignment cnstr--%
constraint
  Z = sum(r in ub(Ring), n in Node)(RingNode[r,n]);

```

```
%---------search option----
ann: search_ann = int_search([RingNode[i,j] | i in ub(Ring), j in Node],
     input_order, indomain_split, complete);
```

REFERENCES

[1] A Catalogue of Problems and their specifications in ESSENCE and other languages.

[2] IBM ILOG Solver, 2000-2004. URL http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/.

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

[4] Multicube fp7-ict project. http://www.multicube.eu/, 2000-2004.

[5] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley and Sons, Chichester, UK, 1997.

[6] J. R. Abrial. *The B-Book. Assigning programs to meaning*. Cambridge University Press, 1996.

[7] Ian Miguel Barbara M. Smith Alan M. Frisch, Brahim Hnich and Toby Walsh. Towards csp model reformulation at multiple levels of abstraction. In *CP-'02: International Workshop on Reformulating Constraint Satisfaction Problems*, 2002.

[8] Nuno Amálio, Fiona Polack, and Susan Stepney. Uml + z: Uml augmented with z.

[9] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. pages 436–450. 2007. doi: http://dx.doi.org/10.1007/978-3-540-75209-7_30. URL http://dx.doi.org/10.1007/978-3-540-75209-7_30.

[10] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0521825830.

[11] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007. ISBN 0521866286.

[12] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. Oxford Univ. Press, 1997.

[13] Tapan P. Bagchi. *Multiobjective Scheduling by Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792385616.

[14] A. Bakshi, V. K. Prasanna, and A. Ledeczi. MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 82–93, New York, NY, USA, 2001. ACM. ISBN 1-58113-425-8. doi: http://doi.acm.org/10.1145/384197.384210.

[15] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36:45–52, 2003. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2003.1193228.

[16] Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of Week of Doctoral Students (WDS99)*, pages 555–564, 1999.

[17] Don S. Batory, David Benavides, and Antonio Ruiz Cortés. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, 2006.

[18] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-CortÃľs. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[19] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortï£¡s. Automated Reasoning on Feature Models. In *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAISE 2005*, page 2005. Springer, 2005.

[20] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-cortï£¡s. Fama: Tooling a framework for the automated analysis of feature models. In *In Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VAMOS*, pages 129–134, 2007.

[21] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. "PISA-A Platform and Programming Language Independent Interface for Search Algorithms". In C. M. Fonseca et al., editors, *Conference on Evolutionary Multi-Criterion Optimization (EMO 2003)*, volume 2632 of *LNCS*, pages 494–508, Berlin, 2003. Springer.

[22] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35:268–308, September 2003. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/937503.937505. URL http://doi.acm.org/10.1145/937503.937505.

[23] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992. ISSN 0360-0300. doi: 10.1145/136035.136043. URL http://dx.doi.org/10.1145/136035.136043.

[24] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.1986.1676819.

[25] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of*

the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07, pages 547–548, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321737. URL http://doi.acm.org/10.1145/1321631.1321737.

[26] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of uml/ocl class diagrams using constraint programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3388-9. doi: 10.1109/ICSTW.2008.54. URL http://dx.doi.org/10.1109/ICSTW.2008.54.

[27] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. Finite satisfiability of UML class diagrams by constraint programming. In *In Proc. of the CP 2004 Workshop on CSP Techniques with Immediate Application*, 2004.

[28] Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, Luis Quesada, Marco Ruffini, David Payne, and Linda Doyle. A combinatorial optimisation approach to designing dual-parented long-reach passive optical networks. *CoRR*, pages –1–1, 2011.

[29] Joseph Chabarek, Joel Sommers, Paul Barford, Cristian Estan, David Tsiang, and Steve Wright. Power awareness in network design and routing. In *In Proc. IEEE INFOCOM*, 2008.

[30] Choco. Choco: an open source Java constraint programming library. http://sourceforge.net/projects/choco/.

[31] Abhijit Davare. *Automated Mapping for Heterogeneous Multiprocessor Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2007. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-115.html.

[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321033.321034. URL http://doi.acm.org/10.1145/321033.321034.

[33] K. Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In Marc Schoenauer, K. Deb, G. Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN VI*, pages 849–858, Berlin, 2000. Springer.

[34] Brandon K. Eames, Sandeep K. Neema, and Rohit Saraswat. Desertfd: a finite-domain constraint based tool for design space exploration. *Design Automation for Embedded Systems*, 2009.

[35] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/software partitioning based on simulated annealing and tabu search, 1997.

[36] Matthew Emerson and Janos Sztipanovits. Techniques for Metamodel Composition. In *OOPSLA ï£¡ 6th Workshop on Domain Specific Modeling*, pages 123–139, 2006. URL http://chess.eecs.berkeley.edu/pubs/289.html.

[37] Niklas Eï£¡n and Niklas Sorensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. URL http://dblp.uni-trier.de/db/conf/sat/sat2003.html.

[38] Cagkan Erbas, Selin Cerav-erbas, and Andy D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation,vol.10,no.3*, 10:358–374, 2006.

[39] Rational Software Corporation et al. *Object Constraint Language Specification ver 1.1*, Sept 1997.

[40] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[41] Thomas A. Feo and Mauricio G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[42] Carlos M. Fonseca and Peter J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evol. Comput.*, 3:1–16, March 1995. ISSN 1063-6560. doi: http://dx.doi.org/10.1162/evco.1995.3.1.1. URL http://dx.doi.org/10.1162/evco.1995.3.1.1.

[43] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123743796, 9780080558363, 9780123743794.

[44] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006, Riva del Garda*, pages 98–102. IOS Press, 2006.

[45] Michael GlaÃ§, Martin Lukasiewycz, Rolf Wanka, Christian Haubelt, and JÃijrgen Teich. Multi-objective routing and topology optimization in networked embedded systems. In *ICSAMOS'08*, pages 74–81, 2008.

[46] Fred Glover. Tabu search fundamentals and uses. Technical report, 1995.

[47] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1 edition, January 1989. ISBN

0201157675. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201157675.

[48] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, 2004. ISSN 0167-9260. doi: http://dx.doi.org/10.1016/j.vlsi.2004.06.001.

[49] object M. Group. Object Constraint Language (OCL).

[50] Sumit Gupta and Lubomir Bic. Distributed adaptive simulated annealing for synthesis design space exploration. 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.38.2781.

[51] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni Vincentelli, and M. Di Natale. Software components for reliable automotive systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 549–554, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1. doi: 10.1145/1403375.1403508. URL http://dx.doi.org/10.1145/1403375.1403508.

[52] Bart Selman Henry, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. pages 337–343. MIT press, 1994.

[53] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie DÃľplanche, and Narendra Jussien. How to solve allocation problems with constraint programming. In *IN PROC. OF THE WORK IN PROGRESS OF THE 17TH EUROMICRO*, pages 25–28, 2005.

[54] Brahim Hnich. Function variables for constraint programming: Thesis. *AI Commun.*, 16(2):131–132, April 2003. ISSN 0921-7126. URL http://dl.acm.org/citation.cfm?id=1218655.1218662.

[55] Holger H. Hoos and Thomas Stützle. Local search algorithms for sat: An empirical evaluation. *J. Autom. Reason.*, 24:421–481, May 2000. ISSN 0168-7433. doi: 10.1023/A:1006350622830. URL http://portal.acm.org/citation.cfm?id=594139.594297.

[56] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN 0262101149.

[57] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 39–48, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-904-6. doi: 10.1145/1879021.1879027. URL http://doi.acm.org/10.1145/1879021.1879027.

[58] Jan Jonsson and Kang G. Shin. A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 158–165, 1997.

[59] Shinjiro Kakita, Yosinori Watanabe, Douglas Densmore, Abhijit Davare, and Alberto Sangiovanni-Vincentelli. Functional model exploration for multimedia applications via algebraic operators. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, ACSD '06, pages 229–238, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2556-3. doi: 10.1109/ACSD.2006.8. URL `http://dx.doi.org/10.1109/ACSD.2006.8`.

[60] Milind B. Kamble and Kanad Ghose. Analytical energy dissipation models for low power caches. pages 143–148, 1997.

[61] Sri Kanajan, Haibo Zeng, Claudio Pinello, and Alberto Sangiovanni-Vincentelli. Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 548–553, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. ISBN 3-9810801-0-6. URL `http://dl.acm.org/citation.cfm?id=1131481.1131632`.

[62] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998. ISSN 1022-7091.

[63] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, December 1984. ISSN 0209-9683. doi: 10.1007/BF02579150. URL `http://dx.doi.org/10.1007/BF02579150`.

[64] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of the IEEE*, pages 145–164, 2003.

[65] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992. URL `http://dblp.uni-trier.de/db/conf/ecai/ecai92.html#KautzS92`.

[66] Minyoung Kim, Sudarshan Banerjee, Nikil Dutt, and Nalini Venkatasubramanian. Design space exploration of real-time multi-media MPSoCs with heterogeneous scheduling policies. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 16–21, New York, NY, USA, 2006. ACM. ISBN 1-59593-370-0. doi: http://doi.acm.org/10.1145/1176254.1176261.

[67] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[68] Mathias Kleiner, Marcos Didonet Del Fabro, and Patrick Albert. Model search: Formalizing and automating constraint solving in mde platforms. In *ECMFA*, pages 173–188, 2010.

[69] S. Kogekar, S. Neema, and X. Koutsoukos. Dynamic software reconfiguration in sensor networks. In *Proc. Systems Communications*, pages 413–420, August 14–17, 2005. doi: 10.1109/ICW.2005.44.

[70] Hessam Kooti, Elaheh Bozorgzadeh, Shenghui Liao, and Lichun Bao. Transition-aware real-time task scheduling for reconfigurable embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 232–237, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. URL http://dl.acm.org/citation.cfm?id=1870926.1870981.

[71] Krzysztof Kuchcinski. Integrated resource assignment and scheduling of task graphs using finite domain constraints. In *DATE*, pages 772–773, 1999.

[72] Krzysztof Kuchcinski. Constraints-driven design space exploration for distributed embedded systems. *J. Syst. Archit.*, 47(3-4):241–261, 2001. ISSN 1383-7621. doi: http://dx.doi.org/10.1016/S1383-7621(00)00048-5.

[73] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8:355–383, July 2003. ISSN 1084-4309. doi: http://doi.acm.org/10.1145/785411.785416. URL http://doi.acm.org/10.1145/785411.785416.

[74] P. J. M. Laarhoven and E. H. L. Aarts, editors. *Simulated annealing: theory and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[75] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. URL http://jmvidal.cse.sc.edu/library/land60a.pdf.

[76] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, May 2001.

[77] M. Lemoine and G. Gaudière. Formal methods for embedded distributed systems. chapter From UML to Z, pages 65–88. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-4020-7996-6. URL http://dl.acm.org/citation.cfm?id=1139062.1139067.

[78] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated Local Search. Technical Report 513, Department of Economics and Business, Universitat Pompeu Fabra, 2000. URL http://ideas.repec.org/p/upf/upfgen/513.html.

[79] Inês Lynce. *Propositional Satisfiability: Techniques, Algorithms and Applications*. PhD thesis, IST, Tecnical University of Lisbon, 2005.

[80] Jan Madsen and Peter Bjørn-Jørgensen. Embedded system synthesis under memory constraints. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 188–192, New York, NY, USA, 1999. ACM. ISBN 1-58113-132-1. doi: http://doi.acm.org/10.1145/301177.301526. URL http://doi.acm.org/10.1145/301177.301526.

[81] Mohammad Mahdavi Mazdeh, Mansoor Sarhadi, and Khalil S. Hindi. A branch-and-bound algorithm for single-machine scheduling with batch delivery and job release times. *Comput. Oper. Res.*, 35(4):1099–1111, April 2008. ISSN 0305-0548. doi: 10.1016/j.cor.2006.07.006. URL http://dx.doi.org/10.1016/j.cor.2006.07.006.

[82] Luis Mandel and MarÃ■a Victoria. On the expressive power of ocl. In *World Congress on Formal Methods*, volume 1708/1999, page 713, Toulouse, France, September 1999. URL http://dx.doi.org/10.1007/3-540-48119-2_47. Dated as it refers to OCL 1, which lacked tuples.

[83] Kim Marriott and Peter J. Stuckey. *Programming with constraints. An introduction.* Cambridge, MA: MIT Press. , 1998.

[84] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: software product lines online tools. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: http://doi.acm.org/10.1145/1639950.1640002.

[85] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 18–27, New York, NY, USA, 2002. ACM. ISBN 1-58113-527-0. doi: http://doi.acm.org/10.1145/513829.513835. URL http://doi.acm.org/10.1145/513829.513835.

[86] Matthew W. Moskewicz and Conor F. Madigan. Chaff: Engineering an Efficient SAT Solver, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.7475.

[87] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex fpgas via search-based boolean sat. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 167–175, New York, NY, USA, 1999. ACM. ISBN 1-58113-088-0. doi: http://doi.acm.org/10.1145/296399.296450. URL http://doi.acm.org/10.1145/296399.296450.

[88] John C. Nash. The (dantzig) simplex method for linear programming. In *Computing in Science and Engg.*, volume 2, pages 29–31. IEEE Educational Activities Department, Piscataway, NJ, USA, 2000. URL http://dx.doi.org/10.1109/5992.814654.

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

[90] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[91] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido 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.

[92] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 1997.

[93] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2:165–193, 1997. ISSN 0929-5585. URL http://dx.doi.org/10.1023/A:1008832202436. 10.1023/A:1008832202436.

[94] Hyunok Oh and Soonhoi Ha. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 133–138, New York, NY, USA, 2002. ACM. ISBN 1-58113-542-4. doi: http://doi.acm.org/10.1145/774789.774817.

[95] Hyunok Oh and Soonhoi Ha. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In *In CODESï£¡02*, pages 133–138, 2002.

[96] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, February 1991. ISSN 0036-1445. doi: 10.1137/1033004. URL http://portal.acm.org/citation.cfm?id=103864.103868.

[97] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. A flexible framework for fast multi-objective design space exploration of embedded systems. In *Integrated Circuit and System Design*, volume 2799 of *Lecture Notes in Computer Science*, pages 249–258. Springer Berlin / Heidelberg, 2003. URL http://dx.doi.org/10.1007/978-3-540-39762-5. 10.1007/978-3-540-39762-5.

[98] Claude Le Pape and Ilog S. A. Constraint-based scheduling: A tutorial.

[99] Joseph Porter, Gabor Karsai, and Janos Sztipanovits. Towards a time-triggered schedule calculation tool to support model-based embedded software design. In *EMSOFT*, pages 167–176, 2009.

[100] Shiv Prakash and Alice C. Parker. Synthesis of application-specific multiprocessor architectures. In *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 8–13, New York, NY, USA, 1991. ACM. ISBN 0-89791-395-7. doi: http://doi.acm.org/10.1145/127601.127612.

[101] Mauricio G. C. Resende. Greedy randomized adaptive search procedures (grasp). *Journal of Global Optimization*, 6:109–133, 1999.

[102] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. ISBN 0444527265.

[103] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

[104] Rohit Saraswat. *A Finite Domain Constraint Approach For Placement and Routing of Coarse-Grained Reconfigurable Architectures*. PhD thesis, Utah State University, May 2010.

[105] Rohit Saraswat and Brandon Eames. Finite domain constraints based delay aware placement tool for fpoas. *Reconfigurable Computing and FPGAs, International Conference on*, 0:145–150, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/ReConFig.2008.50.

[106] Rohit Saraswat and Brandon Eames. On the use of desertfd to generate custom architectures for h.264 motion estimation. In *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 359–368, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3141-0. doi: http://dx.doi.org/10.1109/ECBS.2008.34.

[107] Rohit Saraswat and Brandon Eames. Finite domain constraints based delay aware placement tool for fpoas. In *ReConFig*, pages 145–150, 2008.

[108] Tripti Saxena and Gabor Karsai. Mde-based approach for generalizing design space exploration. In *MoDELS (1)*, pages 46–60, 2010.

[109] 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.

[110] Peter H Scmitt. Uml and its meaning. Technical report, UniversitÃd't Karlsruhe, 2003.

[111] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992. ISBN 0-262-51063-4. URL http://portal.acm.org/citation.cfm?id=1867135.1867203.

[112] Hanif D. Sherali and J. Cole Smith. Improving discrete model representations via symmetry considerations. *Manage. Sci.*, 47:1396–1407, October 2001. ISSN 0025-1909. doi: 10.1287/mnsc.47.10.1396.10265. URL http://portal.acm.org/citation.cfm?id=969994.970083.

[113] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, Chantal Ykman-Couvreur, Maryse Wouters, Carlos Kavka, Luka Onesti, Alessandro Turco, Umberto Bondik, Giovanni Marianik, Hector Posadas, Eugenio Villar, Chris Wu, Fan Dongrui, Zhang Hao, and Tang Shibin. Multicube: Multi-objective design space exploration of multi-core architectures. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, ISVLSI '10, pages 488–493, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4076-4. doi: http://dx.doi.org/10.1109/ISVLSI.2010.67. URL http://dx.doi.org/10.1109/ISVLSI.2010.67.

[114] Helmut Simonis. Building industrial applications with constraint programming. In *CCL*, pages 271–309, 1999.

[115] Helmut Simonis. Eclipse elearning: Lecture notes. 2000.

[116] Frank Slomka, Karsten Albers, and Richard Hofmann. A multiobjective tabu search algorithm for the design space exploration of embedded systems. 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.4930.

[117] Barbara M. Smith. Symmetry and search in a network design problem. In *Proceedings of the Second international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'05, pages 336–350, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-26152-4,

978-3-540-26152-0. doi: 10.1007/11493853_25. URL http://dx.doi.org/10.1007/11493853_25.

[118] Colin Snook and Michael Butler. Uml-b: Formal modelling and design aided by uml. *ACM Transactions on Software Engineering and Methodology*, 15:92–122, 2004.

[119] M. Soto, A. Rossi, and M. Sevaux. Exact and metaheuristics approaches for memory cache management. In *24th European Conference on Operational Research - EURO 2010*, page 272, Lisbon, Portugal, 11-14 July 2010. URL ./Publications/p-soto-10a.pdf.

[120] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992. ISBN 0-13-978529-9.

[121] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30 (4):110–111, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.585163.

[122] Guido Tack. *Constraint Propagation - Models, Techniques, Implementation*. phdthesis, Saarland University, Germany, 2009. URL http://www.gecode.org/paper.html?id=Tack:PhD:2009.

[123] El-Ghazali Talbi. *Metaheuristics : from design to implementation*, volume 10 of *The Sciences Po series in international relations and political economy*. John Wiley & Sons, 2009. ISBN 9780470278581. URL http://www.worldcat.org/isbn/9780470278581.

[124] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. 39th Design Automation Conference (DAC)*, pages 880–885, New Orleans, USA, 2002. ACM Press.

[125] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[126] Mandana Vaziri, , Ana Vaziri, and Daniel Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML.

[127] M. Wall. GAlib: A C++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 1996.

[128] Joachim Paul Walser. Domain-independent local search for linear integer optimization, 1998.

[129] Jules White, Brian Dougherty, Chris Thompson, and Douglas C. Schmidt. Scatterd: Spatial deployment optimization with hybrid heuristic / evolutionary algorithms.

[130] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 129–140, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2888-0. doi: http://dx.doi.org/10.1109/SPLC.2007.12.

[131] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *J. Syst. Softw.*, 82(8): 1268–1284, 2009. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2009.02.011.

[132] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. Design space exploration of automotive platforms in metropolis. In *Society of Automotive Engineers Congress*, April 2006. URL http://chess.eecs.berkeley.edu/pubs/112.html.

[133] Chuanjun Zhang and Frank Vahid. Cache configuration exploration on prototyping platforms. In *14TH IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING*, page 164. IEEE Computer Society, 2003.

[134] Eckart Zitzler and Lothar Thiele. PISA Library, November 2008. URL http://www.tik.ee.ethz.ch/pisa/?page=selvar.php.

[135] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.2440.