Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Measurement and Information Systems

Parallel algorithms in design space exploration

MSc Thesis

Author:

András Szabolcs Nagy

Advisors:

Ákos Horváth, PhD
Dániel Varró, DSc

December 2014

HALLGATÓI NYILATKOZAT

Budapest, December 18, 2014

Nagy András Szabolcs
hallgató

Kivonat

A tervezésitér-bejárás (Design Space Exploration - DSE) célja olyan különböző rendszerterv javaslatok félautomatikus elkészítése, amelyek kielégítik a rendszerrel szemben támasztott numerikus és strukturális kényszereket. A DSE széles körben alkalmazott megközelítés a modellvezérelt rendszertervezésben (Model Driven System Design - MDSD) tervezési folyamatok részleges automatizálására vagy autonóm rendszerek dinamikus újrakonfigurálására. A szabály alapú DSE megközelítések egy kezdeti modellből kiindulva, transzformációs szabályok alkalmazásával érik el a kívánt célállapotot, amelyet jellemzően deklaratív modell-lekérdezések definiálnak. Ilyenkor a tervezési tér bejárás eredményeként transzformációs szabályok egy sorozatát kapjuk, amely a kezdeti modellt egy a célokat kielégítő állapotba viszi át.

A DSE kihívásait sokszor többcélú optimalizációs (Multi-Objective Optimization - MOO) problémaként is felfoghatjuk, amikor néhány numerikus érték minimalizálása vagy maximalizálása mellett kell érvényes megoldást találni. A genetikus algoritmusok (GA) és egyéb metaheurisztikus módszerek elterjedten használtak MOO problémák esetén. Ezek a módszerek egy előre meghatározott mennyiségű egyedet (azaz lehetséges megoldást) tartanak számon és iteratívan bővítik ezek halmazát mutációs és keresztező operációk felhasználásával, miközben folyamatosan eldobják a célfüggvény által gyengébbnek minősített egyedeket.

Ebben a dolgozatban egy olyan (az Université de Montréal egyetem kutatóival közösen kidolgozott) megközelítést mutatok be, amely a többcélú optimalizálás technikáit és a genetikus algoritmusokat használja fel a tervezési tér bejáráshoz, megtartva a szakterület függetlenség és a magas absztrakciós szint előnyeit. Az egyedeket szabályok egy sorozata reprezentálja, a mutációs operátorok ezeket a sorozatokat módosítják például új szabály beszúrásával, a keresztező operátorok pedig szabályokat cserélnek ki két egyed között. Az optimalizálandó numerikus értékeket deiniálhatják a szabálysorozat által elérhető modellállapotra tett modell-lekérdezések, illetve származtathatóak szabály végrehajtásokból is. A szelekciós operátor az NSGA-II genetikus algoritmuson alapul.

A TDK munkám keretében egy Eclipse alapú prototípus implementációt is elkészítettem a VIATRA-DSE keretrendszerre építve, ahol a modellek reprezentálására az Eclipse modellező keretrendszerét (EMF) használtam, míg a modell-lekérdezések és transzformációs lépések definiálását az EMF-IncQueryvel végeztem el. Az új keretrendszer biztosítja a DSE probléma (1) egyszerű definiálását a célok leírásától az operátorok megadásáig, (2) részletekbe menő konfigurálását (például megállási feltétel, valószínűségek), továbbá (3) szabadon testreszabható új mutáció, keresztező és szelekció operátorokkal. Ezenfelül támogatja a többszálú végrehajtást is és egy lazán csatolt tesztkeretrendszer könnyíti a működés analizálását. Két különböző alkalmazásterületről származó esettanulmányon elvégzett mintakísérletekkel és mérésekkel támasztottam alá a kidolgozott megoldás gyakorlati alkalmazhatóságát.

Abstract

The goal of Design Space Exploration (DSE) is to semi-automatically synthesize various design candidates satisfying numerical and structural constraints. DSE is frequently used in Model Driven Systems Design (MDSD) to partially automate design processes or to dynamically reconfigure autonomous systems. Rule-based DSE aims to achieve this by starting from an initial model which evolves by transformation rules until the desired goals are reached (which are captured by model queries). As a result, rule-based DSE finds a sequence of rules which transforms the initial model to reach a target state satisfying the goals.

Many DSE challenges can be seen as Multi-Objective Optimization (MOO) problems i.e. it should find valid solutions while maximizing or minimizing several numerical values derived from the model. Genetic Algorithms (GA) and other meta-heuristics are widespreadly used for MOO. They maintain a predefined number of solutions or individuals and iteratively create new ones by mutation and crossover operations while it drops the low quality candidates.

In this report I present an approach (developed in collaboration with researchers from BME-MIT and Université de Montréal) to exploit multi-objective optimization techniques and use genetic algorithms for rule-based design space exploration keeping both domain independence and high level abstraction. Individuals are represented as rule trajectories, mutation operations modify the trajectory, crossover operations exchange rules between trajectories, while objectives are defined by model queries or are derived from rule executions. The selection operator is based on the Non-dominated Sorting Genetic Algorithm (NSGA-II).

I developed a prototype implementation built upon the VIATRA-DSE framework which uses Eclipse-based tools like EMF for model representation and EMF-IncQuery for model queries. The newly created framework supports extensive configurations like different stop conditions, custom genetic operators and multithreaded execution. The practical feasibility of the approach is demonstrated by experimental evaluation carried out on two case studies from different application domains.

Contents

Chapter 1

Introduction

As a challenging branch of search based software engineering (SBSE), design space exploration (DSE) aims at searching through different design candidates to fulfill a set of constraints and then proposing optimal designs with respect to certain objectives. It frequently supports activities like configuration design of avionics and automotive systems. Many of such traditional static DSE problems can be solved by using advanced search and optimization algorithms or constraint satisfaction programming techniques [1, 2, 3, 4, 5].

In model-driven engineering (MDE), *rule-based DSE* [5, 6, 4] aims to find instance models of a domain that are (1) reachable from an initial model by applying a sequence of exploration rules, while (2) constraints simultaneously include complex structural and numerical restrictions. Model driven techniques offer expressive modeling languages and advanced tools to capture the DSE problem of different domains independently on a high level of abstraction close to the domain itself. However, solving a rule-based DSE problem is a difficult challenge due to the inherently dynamic and incremental nature of the problem. Such dynamic and incremental DSE problems may arise in complex reconfiguration challenges of supervising cyber-physical systems (CPS) or IT infrastructure [5] or quick fix generation in domain-specific modeling environments [7].

As a practical observation, the solution space of a rule-based DSE problem is dense in principle, but one cannot put an *a priori upper bound* on the number of model elements used in a design candidate (i.e. model elements may be created and deleted during exploration). Unfortunately, this makes the exhaustive exploration of the design space intractable. Furthermore, many practical problems necessitate to continue the exploration of the design space incrementally from a previous solution (instead of starting the search from scratch each time). Such incremental solving is rarely handled by state-of-the-art constraint solvers (as demonstrated in [8]).

Rule-based model-driven DSE problems have additional challenges also from an optimization perspective. First, some objectives are not values of simple cost attributes but complex model metrics calculated by model queries. Furthermore, certain cost calculations may depend on the sequence of exploration rules applied on the design model. Finally, we may not find a single combined objective function but multiple objectives may need to be incorporated to identify the best design candidates.

Existing rule-based DSE solutions exploit 1) model checking with powerful graph-based symmetry reduction [6], 2) dependency analysis and hints by formal abstractions [5] or 3) different search strategies (e.g. hill climbing, simulated annealing) [4]. As a commonality in these approaches, the core exploration procedure follows a *local-search based approach*, i.e. it gradually extends the search towards promising candidates by priorities defined by local heuristics. However, if these heuristics fail, no solutions will be retrieved.

Parallel (or global) search techniques (like genetic algorithms or multi-objective optimization) have already proved to be successful in various MDE scenarios for finding constraints [9], model transformations [10] or solving static DSE problems [11] where an exhaustive search algorithm becomes infeasible. Moreover, they provide graceful degradation for problems where no solutions exist which meet all the objectives and constraints by relaxing hard constraints to soft constraints.

In the current report, I present an approach (developed in collaboration with researchers from BME-MIT and Université de Montréal) to integrate multi-objective optimization techniques by using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [12] to drive rule-based design space exploration. For this purpose, finite populations of the most promising design candidates are maintained with respect to different optimization criteria. In our context, an individual of a generation is defined as a sequence of rule applications leading from an initial model to a candidate model. Populations evolve by mutation and crossover operations which manipulate (change, extend or combine) rule execution sequences to yield new individuals. However, a key technical challenge that we face in genetic rule-based DSE is to preserve the feasibility of candidate solutions which are generated using genetic operators. Indeed, in rule-based DSE, crossing two feasible solutions, and/or randomly mutating a feasible solution, may yield infeasible candidates if the corresponding rule execution sequence is infeasible. In our approach, candidate solutions generated using genetic operators are automatically corrected to preserve their feasibility.

The main added value of our multi-objective optimization approach for rule-based DSE is to seamlessly lift multi-objective optimization techniques to a domain-independent model-level exploration process while preserving a high-level of abstraction. Design candidates will still be represented as models and the evolution of these models as rule execution sequences. Constraints are captured by model queries while objectives can be derived both from models and rule applications. On the theoretical level, models are formalized as graphs, model queries as graph patterns and exploration rules as graph transformation rules, thus our work can be considered as a multi-objective exploration of a graph transformation system.

I developed a prototype implementation built upon the VIATRA-DSE framework (developed by Miklós Földényi and me and presented in [13]) which uses Eclipse-based tools like EMF for model representation and EMF-IncQuery for model queries. The newly created framework supports extensive configurations like different stop conditions, custom genetic operators and multithreaded execution. The practical feasibility of the approach is demonstrated by experimental evaluation carried out on two case studies from different application domains.

The report is structured as follows:

- Chapter 2 presents two motivating examples along with their challenges and gives a brief introduction to modeling techniques, including the definitions of metamodel and graph transformation.
- Chapter 3 explains the different approaches for design space exploration.
- Chapter 4 describes the different metaheuristic techniques for optimization problems, including local search techniques and genetic algorithms.
- Chapter 5 briefly describes the basic concepts and difficulties of concurrent programming.
- Chapter 6 presents our approach for rule-based design space exploration for multi-objective optimization with the NSGA-II algorithm.

- Chapter 7 gives an overview my contribution to the approach and implementation details.
- Chapter 8 evaluates the approach with an extensive analysis on the two case studies.
- Chapter 9 summarizes the report.

In order to maintain a consistent appearance of the report, the following rules are imposed: (1) the four background chapters are written in third person singular, (2) as the approach and the evaluation is a result of a collaboration with researchers from BME and Université de Montréal chapter 6 and section 8.1 are mainly written in first person plural to emphasize joint work, while (3) chapter 7, section 8.2 and section 8.3 are written in first person singular as those results are built exclusively on my own work.

Our initial results have been published in [14] at the IEEE International Conference on Automated Software Engineering. This report uses text and figures from this paper, but also significantly and conceptually extends it by (1) presenting the background in more details, (2) providing insights on how NSGA-II can be adapted to rule-based DSE and (3) introducing a new case study on business process modeling.

Furthermore, this work is presented on this year's Scientific Student Conference (BME-VIK TDK 2014) [15], but was corrected, made some part more readable and extended by chapter 5, subsection 7.1.2, subsection 7.1.4 and section 8.3. The main added value is explaining concurrent programming, the generation of the initial population, parallelization of VIATRA-DSE and the evaluation of the parallelization of the genetic algorithm.

Chapter 2

Preliminaries

In this chapter section 2.1 describes two motivating examples: configuration of cyber-physical systems (2.1.1) and optimization of business process models (2.1.2). Section 2.2 takes an introduction to modeling, including the definition of metamodels (2.2.1), how validation of a model can be achieved (2.2.2) and rules which embrace the possible modifications of a model. Lastly related tools are discussed.

2.1 Motivating scenarios

This section presents two case studies which are used to evaluate the approach presented in this paper and to demonstrate definitions and concepts required to understand the approach.

2.1.1 Case study - cyber-physical systems

Cyber-physical systems (CPS) integrate computation, networking and physical processes [16]. Services deployed on a network of computers, monitor physical processes with sensors and intervene into them using designated actuators, while changes in the physical world can also affect computations. Thus CPS creates a very strong connection between the cyber and physical parts of the system.

Smart buildings are frequently considered as a cyber-physical system. In such a building, companies can rent offices consisting of multiple rooms with highly configurable services such as fire alarm, air conditioning and security monitoring as depicted by orange boxes in Figure 2.1. These services require certain types of applications (presented in the middle with green boxes including smoke detection): temperature measurement (*Measure Temp*) air conditioning (*Set Temp*), heat map, video recording and motion checking. This cyber-physical system has four types of sensors, namely *Smoke Sensor*, *Temperature Sensor*, thermographic camera (*Infra Cam*) and *Video Camera* which are presented at the bottom of the figure in purple boxes with a *Compute Server* handling computationally intensive tasks. Applications require certain host types to be able to fulfill their functionality, this dependence is represented as dashed arrows, while numbers on them mean further requirements: exclusive usage of a host is presented by a number one, while x/y like numbers denote the memory unit / permanent storage unit required from the Compute Server.

Rooms are offered to rent with four types of packages (see Figure 2.2):

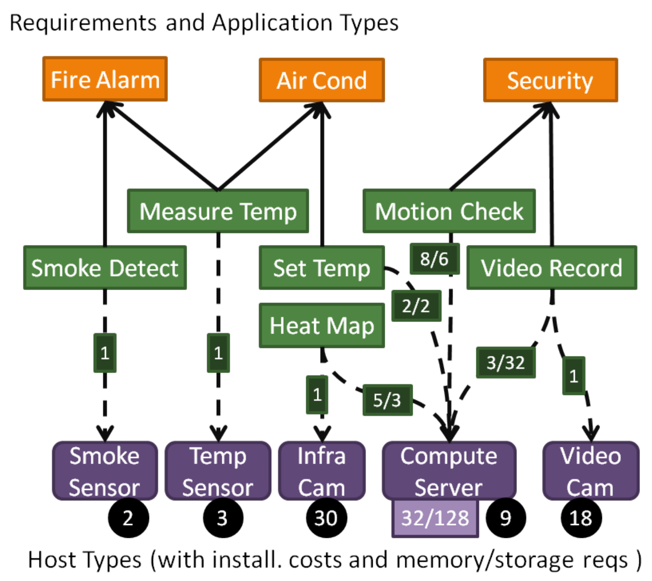
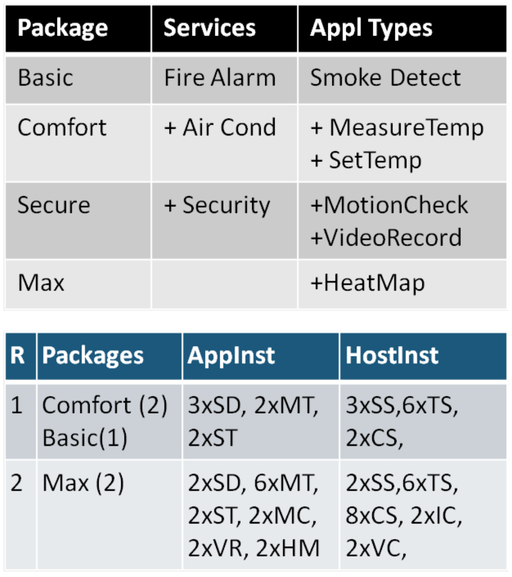


Figure 2.1: Model of the smart building



Package	Services	Appl Types
Basic	Fire Alarm	Smoke Detect
Comfort	+ Air Cond	+ MeasureTemp + SetTemp
Secure	+ Security	+MotionCheck +VideoRecord
Max		+HeatMap

R	Packages	ApplInst	HostInst
1	Comfort (2) Basic(1)	3xSD, 2xMT, 2xST	3xSS,6xTS, 2xCS,
2	Max (2)	2xSD, 6xMT, 2xST, 2xMC, 2xVR, 2xHM	2xSS,6xTS, 8xCS, 2xIC, 2xVC,

Figure 2.2: Services are organized into packages

- **Basic**: This package runs the compulsory fire alarm service which requires one smoke sensor for each room.
- **Comfort**: This package also offers air conditioning by measuring temperature by three sensors (per room) and setting the required temperature. Measuring the temperature also offers a backup solution for fire alarm in case the sensors fail.
- **Secure**: This package extends the *Comfort* package to offer security surveillance by a video camera continuously recording events in the room and a motion check application which highlight critical events automatically.
- **Max**: This package enhances the *Secure* package by providing a heat map of the room which can be used for fire alarm as well as for surveillance purposes.

Two sample company requests are also listed in Figure 2.2 which summarizes the selected packages for a certain number of rooms together with the application instances to be deployed and hardware devices to be installed. For instance, the first request consists of 2 rooms with comfort package and 1 room with basic package, and it necessitates to run smoke detector (SD) service (for 3 rooms, 1 device per room), the measure temperature (MT) service (for 2 rooms, 3 devices per room) and the set temperature application (for 2 rooms, jointly installed on a compute server).

Configuring a cyber-physical system (i.e. installing devices, allocating and running applications) can be considered as a design problem as multiple constraints and optimization objectives must be incorporated. However, this configuration is a dynamic problem as (1) requests may change over time (new requests arrive, existing ones are canceled) and (2) certain faulty devices may no longer function. This way, the system must be reconfigured at run-time calculating a new design which starts from the last configuration and incorporates the changes in the context and requirements of the system.

2.1.2 Case study - business process modeling

Business process modeling and notation (BPMN) [17] is an Object Management Group (OMG) [18] standard providing an abstract syntax and graphical notation for business workflows.

BPMN is widespreadly used where business processes are complex and collaboration between organizations is required as well as between different software components.

BPMN defines different types of tasks as basic elements, such as manual tasks which have to be done by a person without aid of the software, while a script task should be executed by the business process engine. Directed edges between tasks describe data and control flow, which can also have conditions on them. Events can start, end or interrupt a process. Resource types can be allocated to tasks and resource instances can be used by the running tasks with mutual exclusion. Furthermore parallel gateways can divide a flow into multiple parallel tasks.

For example, Figure 2.3 presents a simplified business process of a web shop. After *authorization* (i.e. the user is signed in) the request can be either about *purchasing* (e.g. editing user information, paying with a bank card, etc.) or about browsing items. In the later case *recommendations* are calculated in parallel with *fetching item data* and *logging*. All tasks need a particular resource either a *web server*, a *SQL* or a *NoSQL* database.

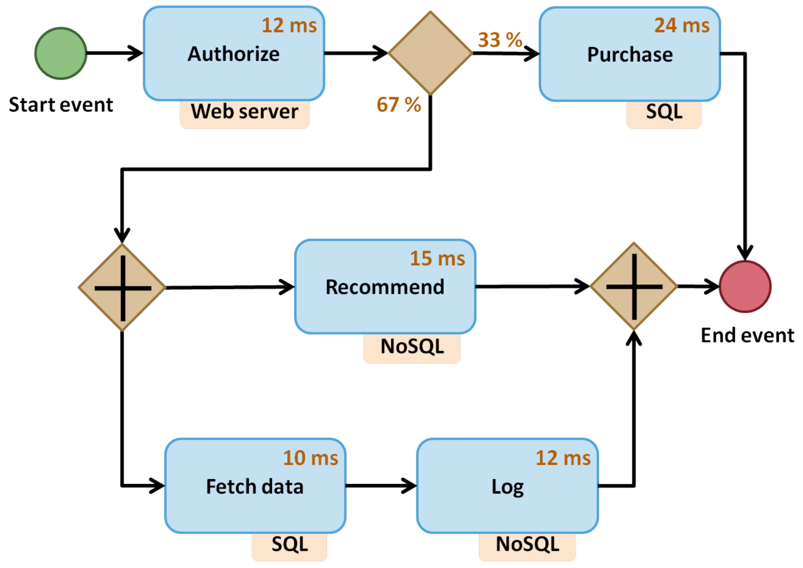
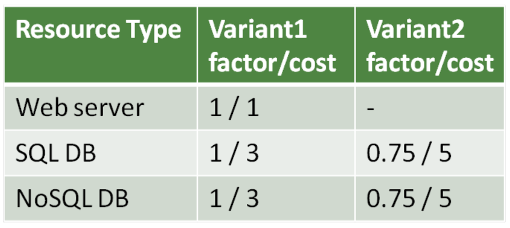


Figure 2.3: An example BPMN model



Resource Type	Variant1 factor/cost	Variant2 factor/cost
Web server	1 / 1	-
SQL DB	1 / 3	0.75 / 5
NoSQL DB	1 / 3	0.75 / 5

Figure 2.4: Resources type variants

Simulation of business processes can reveal performance bottlenecks or analyze sensitivity to parameters. For a simulation run certain number of tokens are sent through the system, which are processed for a predefined time, and flow probability is assigned to the directed edges to randomly drive the business process execution to different paths. Simulation needs several extra pieces of information, e.g. task execution time, decision flow probability, etc. which can be found on Figure 2.3 with brown colour. In this example resources can have different variants, which influence the execution time and are shown on Figure 2.4.

However, optimizing a process with respect to multiple objectives (e.g. response time, resource usage, etc.) is a time consuming and erroneous task. Running simulations with different parameters is required to find an optimal solution, while evaluating results to identify a promising configuration is essential. On the other hand, bad design can lead to an inefficient workflow, making the optimization stuck in suboptimal solutions. Therefore automating the optimization is crucial. By simulating the business process and evaluating the results, one can simultaneously search for an optimal parametrization and design flaws. In addition, contradicting objectives may be present, for example increasing resource utilization can lead to reduced throughput.

2.1.3 Challenges

The common challenges between the two case studies are the following:

- Optimization includes multiple objectives which potentially conflict with each other.
- Objectives can have different priorities (e.g. response time is more important than resource usage).
- Simultaneously with the optimization process, validation of complex structural constraints is also necessary.
- Further constraints and restrictions can filter out acceptable solutions.
- Reconfiguration might be needed at run-time. Therefore incremental solutions have to be created, i.e. the previous solution has to be reevaluated upon changes in the environment and objectives.
- The system has to apply the solution of a reconfiguration to itself, therefore a solution has to provide well-defined steps or rules which can be executed by the system.

2.2 Introduction to modeling techniques

This section introduces several definitions and concepts used in model driven system design (MDSD) including metamodels, graph patterns and transformation rules.

2.2.1 Metamodels and instance models

Model driven system design (MDSD) aims to lift the abstraction level of a problem allowing a better overview of it. For this purpose a **domain model** is created which describes the possible elements of the problem, their properties and their relations. For example the domain model of cyber-physical systems must somehow define sensors and how applications can use them, as they are part of the core concept. However, a domain model lacks describing a concrete system like a smart room from subsection 2.1.1 with several devices. This information is included in **a(n instance) model**. Domain models are also called **metamodels** as a metamodel describes the possible components of a semantic model, also called an instance model. In a metamodel *classes* describe objects from the domain which can have *attributes*, while *references* specify the relations between classes.

To present an example of a domain model Figure 2.5 shows a possible metamodel of any cyber-physical system (while the BPMN metamodel can be found in Appendix A because of space restrictions). Then the smart building case study on Figure 2.1 can be interpreted in this metamodel in the following way: rental companies will issue *Requests* for implying a set of *Requirements* (defined by the number of rooms and desired packages) each of which identifies a required application type (*ApplType*) and the number of redundant instantiations for satisfying the requirement (*count* attribute). Sensors and computation units are uniformly called *HostTypes*. Each application type may claim several host types and sufficient amount of resources (e.g. memory, storage) for its execution as defined by a resource requirement (*ResReq*). For instance, calculating a heat

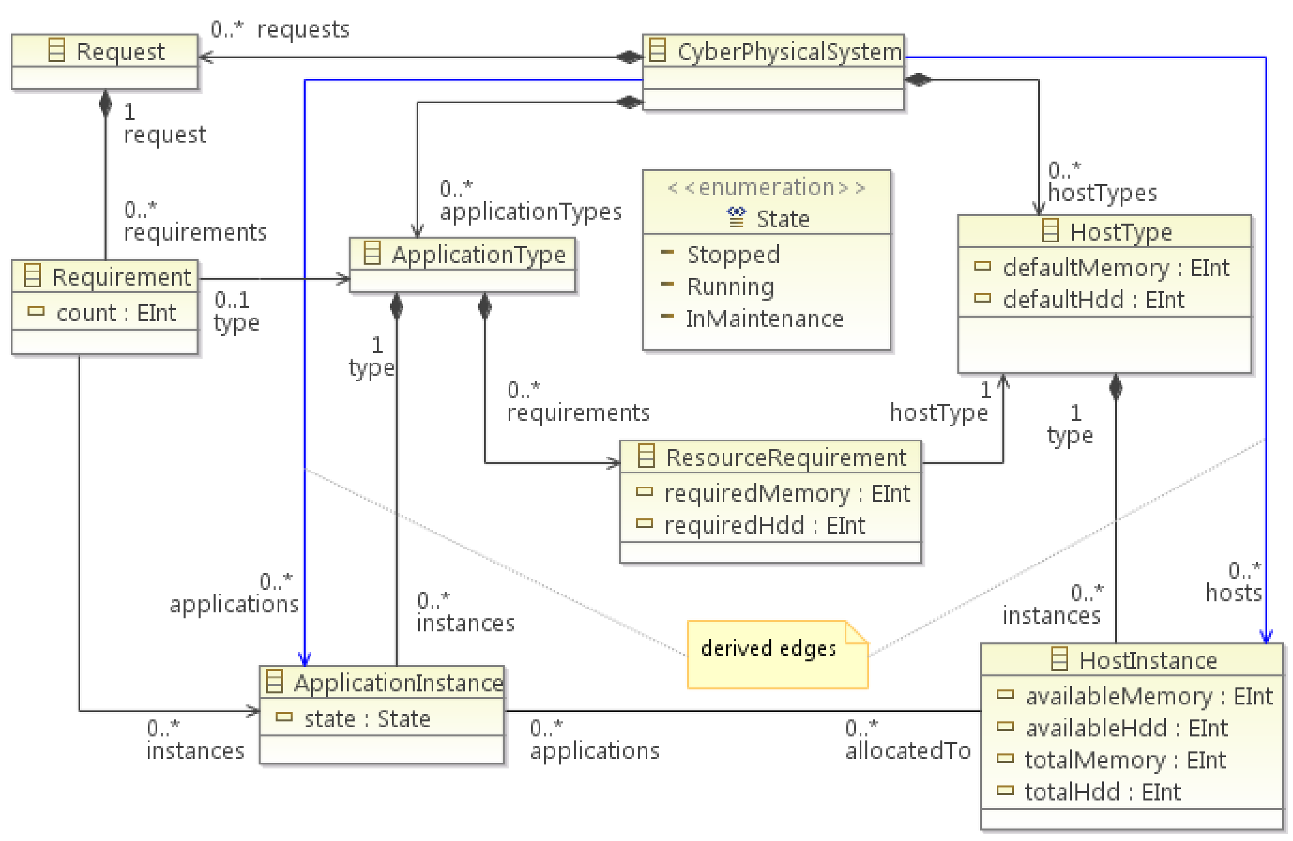


Figure 2.5: Metamodel of cyber-physical system

map requires five units of memory and five units of permanent storage as defined by the labels of dashed arrows in Figure 2.1.

In the running system, multiple application instances (abbreviated as *ApplInst*) of an application type may exist each of which is deployed on a host instance (*HostInst*) corresponding to a specific host type. Such deployment consumes a certain amount of resources in the host instance (up to its total memory and storage) as specified by the corresponding resource requirement. Application instances need to be started after they are allocated to a host instance, and stopped when they are no longer needed, which fact is represented by the *state* attribute.

2.2.2 Well-formedness constraints

Well-formedness **constraints** extend the expressive power of metamodels by capturing further restrictions for instance models. For example a cyber-physical system requires that all the application instances are actually running. Constraints can be used to capture these valid or invalid configurations. They are frequently defined by **model queries** and formalized by **graph patterns**.

A graph pattern can be seen as a small instance model, which should be found within the actual instance model. Given a graph pattern p and an instance model M, $m : p \mapsto M$ denotes a graph morphism identifying a **match** fragment of M. For example, a single application instance without defining its attributes can be a graph pattern and it has as many matches as many application instances are in the instance model. A graph pattern can also capture relations, attributes and negative conditions. Figure 2.6 presents four constraints (graph patterns) as an example, where the first three constraints define desired situations, while the fourth constraint captures an undesired case:

- **satisfiedReq(E)** identifies a requirement E of a request R which is instantiated into a sufficient number of application instances (i.e. the number of instances is equal to the required redundancy);
- **allocatedAppl(AI)** identifies an application instance AI which is allocated to a host instance HI to fulfill the resource requirement between the corresponding application type AT and host type HT;
- **appInstRun(AI)** identifies an application instance of a configuration which is running;
- **extraHost(H)** identifies a host instance H which does not host any application instances.

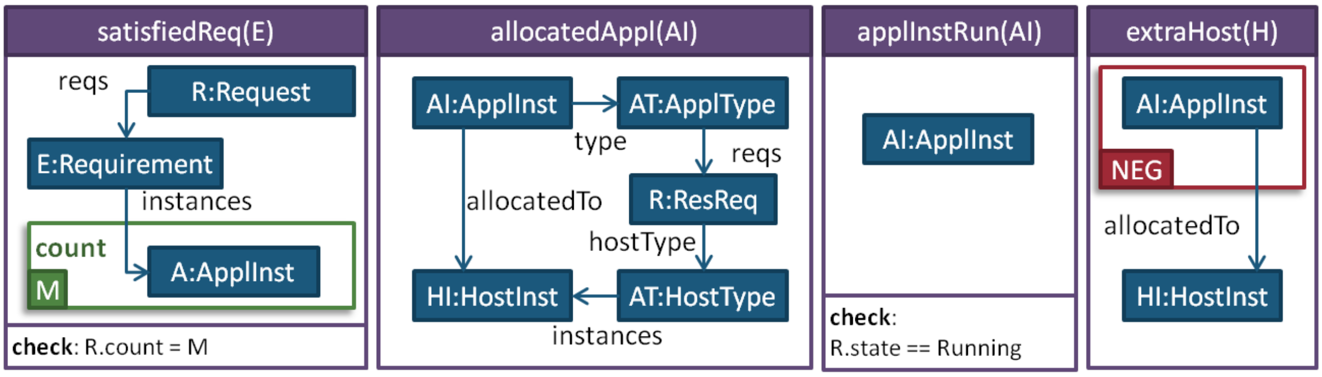


Figure 2.6: Constraints for smart building configuration

2.2.3 Transformation rules

Modifications to an instance model are often described as **transformation rules**. A rule $R = (LHS, RHS)$ consists of a precondition or a left hand side (LHS) which is captured by a graph pattern and a right hand side (RHS) which declaratively defines the effects of the operation. A rule is applied on a model by (1) finding a match of graph pattern LHS (also called an **activation** of the rule), then (2) removing elements from the model which have an image in $LHS \setminus RHS$, then (3) changing the value of attributes which are reassigned in RHS and finally (3) creating new elements $RHS \setminus LHS$. The rules of the examples are depicted in Figure 2.7 and in Appendix A (as a combined notation such as in GROOVE [6]).

- Rule **newHostInst** installs a new host instance HI of a host type HT and sets the available resource parameters to that of the host type.
- Rule **newApplInst** creates a new application instance AI in accordance with the *count* attribute of requirement E (by reusing the condition defined by graph pattern *unsatisfiedReq(E)*).
- Rule **start** initializes a stopped application instance AI which is already allocated to a host device while rule **stop** stops a running instance.

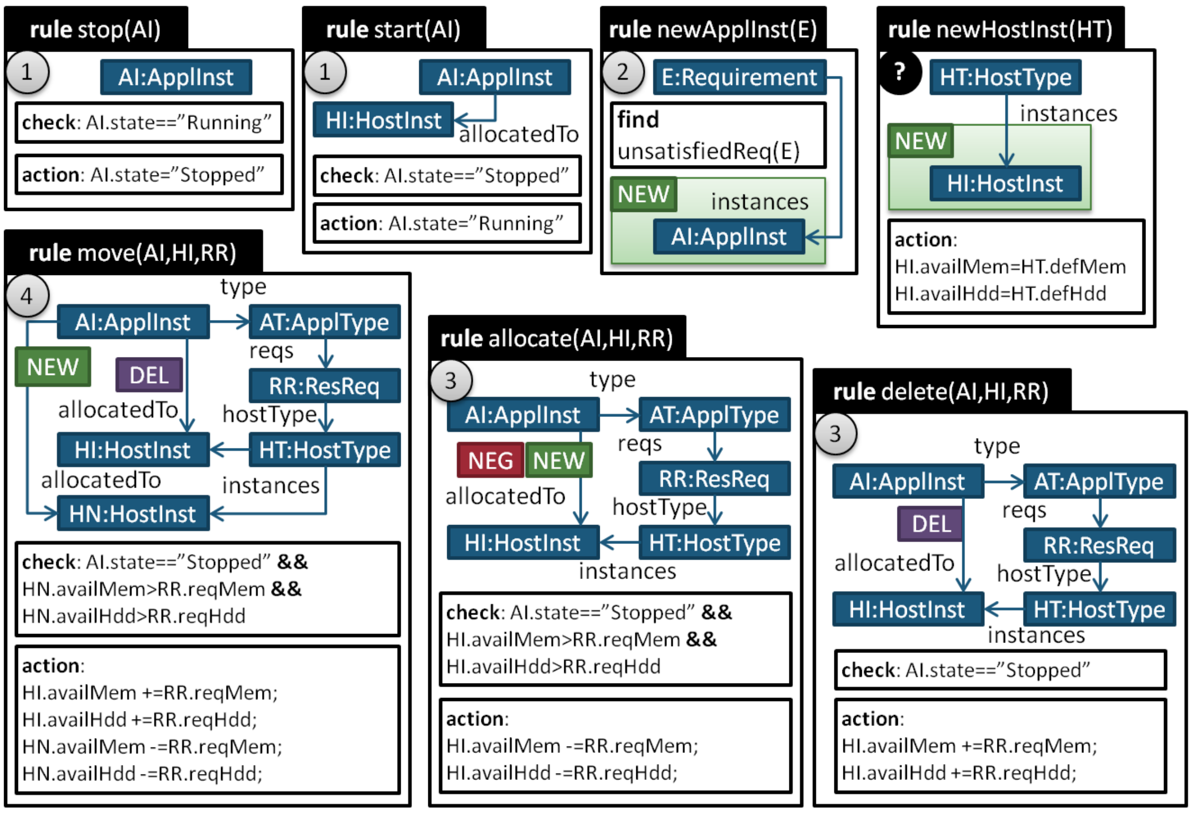


Figure 2.7: Exploration rules of the smart building example

- Rule **allocate** aims to allocate (an unallocated and stopped) application instance AI to a host instance HI in accordance with the resource requirement RR provided that sufficient memory and storage space is still available at HI.
- Rule **delete** removes an existing allocation of a stopped application instance AI from a host instance HI, and frees the related memory and storage resources of HI.
- Rule **move** combines the allocate and delete rule into one, and changes the allocation of an application instance AI from host instance HI to HN, and adjusts the resource usage accordingly.

2.2.4 Related work

There are various tools which can help the modeling process. Domain models can be represented in Unified Modeling Language (UML) [19] which is an Object Management Group (OMG) [18] standard and there is a wide range of tools for UML including Visual Paradigm for UML [20] and Papyrus [21]. Eclipse Modeling Framework (EMF) [22] is a de facto standard for creating UML class diagram like domain models and generating Java code from it and widely used in the industry. Resource Description Framework (RDF) [23] is originally designed for metadata modeling for web technologies, but can be used for metamodeling too. Web Ontology Language (OWL) [24] is an ontology language and tools like Protégé [25] support it.

Model queries, expressions, constraints on UML models can be defined with Object Constraint Language (OCL) [26] which was originally developed by IBM and is now an OMG standard. SPARQL Protocol and RDF Query Language (SPARQL) [27] is an RDF database query language and is managed by the World Wide Web Consortium (W3C) [28]. EMF Query [29] is a query language for EMF models, while the purpose of EMF-IncQuery[30] is the same, but has a declarative language and evaluates the expression incrementally on model changes.

For graph transformation tools see the related work of the next chapter (section 3.3).

In this report domains are modeled with EMF, constraints are defined by EMF-IncQuery and model transformations are executed via the VIATRA framework [31]. Conceptually, many other tools could potentially be used.

Chapter 3

Design space exploration

Design space exploration (DSE) aims to find optimal design candidates of a domain with respect to different objectives where design candidates are constrained by complex structural and numerical restrictions. It can be either used to partially automate a design process or to dynamically reconfigure an autonomous system at run-time.

Design space exploration can be categorized as follows:

- **Constraint-based DSE** $CDSE = (M_0, C)$, consists of an initial model M_0 and a set C of constraints. As result, design space exploration returns a solution model M_{Si} (there can be more than one solution), which is a modified version of M_0 and satisfies all the constraints from C.
- **Rule-based DSE** $RDSE = (M_0, C, R)$, also consists of an initial model M_0 and a set C of constraints, but allows the modification of the initial model M_0 along a set R of rules. As a result, it returns a sequence of rules which if applied to the initial model, it will satisfy all of the constraints.

There are two options to carry out design space exploration:

- **Model-driven DSE** solves a problem directly over models.
- A DSE problem can be solved on a mathematical representation by **back-end solvers** and **back-annotation**.

Table 3.1 presents the available approaches for design space exploration in the previously defined categories. GROOVE, Henshin and VIATRA are frameworks.

	Constraint-based	**Rule-based**
Back-end solver	CSP, SMT, SAT	Planners, Model-checkers
Model-driven	Alloy	GROOVE, Henshin, VIATRA

Table 3.1: Design space exploration classification

This chapter introduces design space exploration and its two main approaches. Section 3.1 describes how to use constraint satisfaction problem solvers for design space exploration, while model-driven, rule-based design space exploration and its challenges are explained in section 3.2.

3.1 Constraint based design space exploration

The input of constraint satisfaction problems can be defined as a triple $CSP = (Z, D, C)$, where Z is a set of variables $x_1, x_2, \ldots x_n$, D defines the domains of the variables (e.g. all of them are integers) and C a set of constraints between the variables (e.g. $x_1 > x_2$). A solution to a CSP is an assignment of domain values to variables which also satisfies the constraints of the CSP problem. There are three major types of CSP according to the domain: 1) CSP(B) consist of boolean variables and boolean expressions as constraints, 2) CSP(FD) takes integers as variables and 3) CSP(R) is defined by real numbers.

Famous CSP problems include the map colouring problem, the eight queen puzzle, minimum set cover or popular logic puzzles like Sudoku. These problems are old enough that mature solvers and constraint satisfaction methods exist. CSP(B) are solved by SAT solvers, CSP(FD) are solved with methods from the field of artificial intelligence [32] (like constraint propagation and backtracking), while Gauss elimination and simplex method are used to solve CSP(R).

Solving a DSE problem as a constraint satisfaction problem is traditionally carried out by automatically generating the CSP problem by a forward model transformation from high-level engineering models and then back-annotating the results of the analysis to the source domain [2]. Thus $DSE = (M_0, G)$ is mapped to $CSP = (Z, D, C)$, where the variables Z and domains D are derived from M_0, while constraints C is derived from the goals G.

For example, Figure 3.1 illustrates an allocation problem where software modules should be deployed on a hardware architecture and all the possible allocations are depicted by the arrows. These arrows can be represented as binary variables $x_{ij} \in \{0, 1\}$ where 1 (true) means it is used in a solution. Goals such as 1) all software modules should be allocated to a single hardware and 2) maximum two software modules can be allocated to *HW1* can defined by constraints on x_{ij} as shown in Equation 3.1 and Equation 3.2.

$$\forall i : \quad x_{i1} + x_{i2} = 1 \tag{3.1}$$

$$\forall i : \quad \sum_i x_{i1} <= 2 \tag{3.2}$$

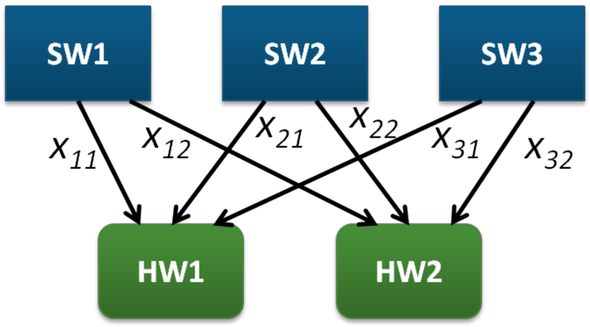


Figure 3.1: All possible allocations of an allocation problem

However, in model driven system design the representation of the problems are graph like structures and constraints are defined by graph patterns (explained in section 2.2), which can be hard to map into variables and equations. Especially constraints like connectedness are mathematically impossible to map as graph patterns have a bigger expressive power. For these reasons a new approach appeared which raises abstraction level by introducing existing algorithms and heuristics directly on the model level.

Apart from CSP other back-end solvers can be used for design space exploration. SMT solvers and alloys are logic based techniques which can find counter-example models for a set of constraints.

3.2 Rule-based design space exploration

This section presents the definition and the challenges of the rule-based approach of model-driven design space exploration.

3.2.1 The problem of rule-based design space exploration

The aim of model-driven design space exploration is to seamlessly integrate the search for design candidates to modeling techniques where models are given as graph like structures and constraints as graph patterns which are described in section 2.2. The rule-based approach of model-driven DSE also defines a set of domain dependent graph transformation rules over the model.

Note: from now on design space exploration will stand for only the model-driven, rule-based approach.

The input of rule-based DSE consists of four element $RDSE = (M_0, G, R, GC)$: 1) an initial model M_0, 2) a set G of goals given by graph patterns which should be satisfied by the solution model M_{Si}, 3) a set R of transformation rules $(r_1, r_2, \dots r_r)$ which defines how the initial model M_0 can be manipulated to reach a solution model M_{Si} and 4) a set GC of **global constraints** (also graph patterns) which has to be satisfied along each valid execution path i.e. **trajectory** $\vec{r_n} = M_0 \overset{r_1,m_1}{\longrightarrow} M_1 \dots \overset{r_n,m_n}{\longrightarrow} M_n$. As a result it produces several solutions $(M_{S0}, M_{S1} \dots M_{Sn})$ each described by a sequence of rules $(\vec{r_{S0}}, \vec{r_{S1}} \dots \vec{r_{Sn}})$ which if applied to the initial model, it will satisfy all of the goals and global constraints.

For example a cyber-physical system can be reconfigured by rule-based design space exploration. DSE requires the actual state of the system model, the set of constraints defined on Figure 2.6 and the rules on Figure 2.7 (global constraints are not necessary). During design space exploration, the exploration rules are applied to the system model to drive reconfiguration as illustrated by the trajectory of Figure 3.2.

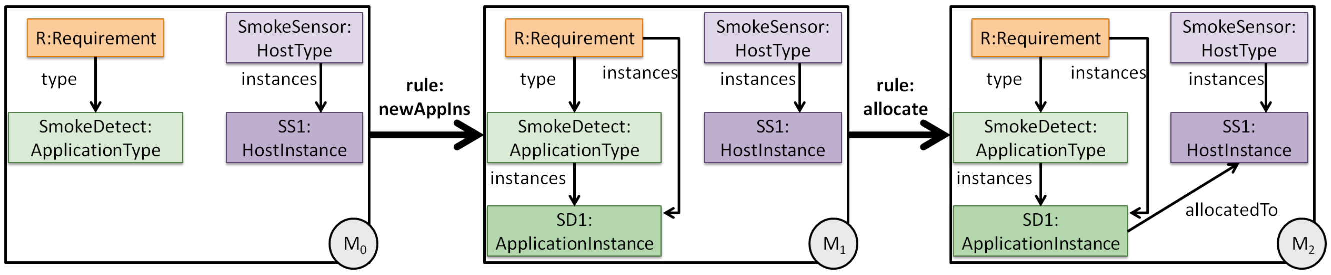


Figure 3.2: A solution trajectory

This approach has two major advantages compared to using constraint satisfaction problem solvers. Rule-base DSE has the ability to keep the high level abstraction of the domain model therefore lacking the mapping problem to CSP. Furthermore it returns a trajectory how to produce a particular solution from the initial model by using the given rules, while a CSP solver only produce an assignment of the variables.

3.2.2 Challenges of Rule-based design space exploration

To comprehend the challenges it is necessary to understand the structure of design space. The design space (or state space) can be imagined as a directed graph as in many search based tasks. Nodes represent certain states of the model M_i, while arcs serve as rule activations (r_i, m_k). Node of the initial model M_0 is called a root node, while certain nodes can be solution nodes (M_{S0}) which satisfy the goals.

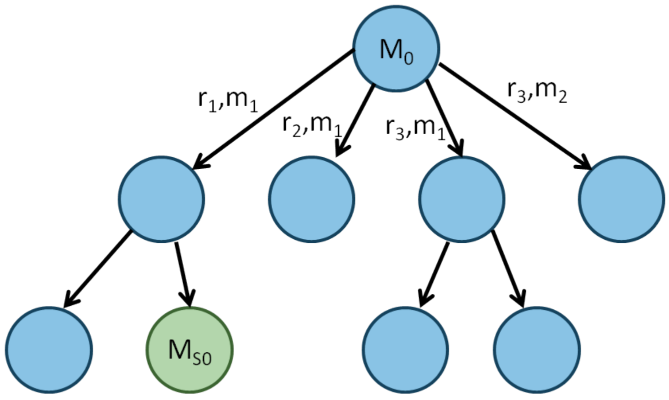


Figure 3.3: Design space as a directed graph

Rule-based design space exploration has two major challenges. Firstly, DSE must find the best solutions according to the objectives and goals fast. Secondly, identifying already traversed model states requires state encoding and rapid comparing of these codes, while encoding activations during exploration is a challenge too.

Challenge 1 As in many search based tasks, finding solution states is hard because of the size of the design space. The size of the model and the number of rules influence the design space exponentially as rules have more activations in bigger models and further rules imply even more raising the number of outgoing arcs from nodes. Furthermore the design space can be growing unboundedly if the problem includes rules which can create objects without limitations.

There are two options to overcome this challenge. The size of the design space can be reduced by using extra global constraints, which restrict the solution space by pruning some potential solutions, and by choosing the rules carefully. The other option is to guide the exploration and to use a proper traversal strategy. Possible strategies are explained in chapter 4.

Challenge 2.1 The other challenge is to identify already traversed states. Ignoring these states can lead to infinite loop when traversing the design space, while same solutions can be returned multiple times reducing efficiency and effectiveness. Detection can be achieved by saving the model itself after every step and comparing it to all the previous ones but this would be a computationally intensive approach. Another option is to store deltas between model states as comparing them is a less difficult task (as used in GROOVE [33]). Alternatively compact IDs can be generated from the model after every rule execution and saved in a data structure where efficient comparing is possible like a hash map as proposed in VIATRA-DSE [13] (and be referred as **state code** or **state id**).

Challenge 2.2 Encoding of rule activations is beneficial as most traversal strategies need to know which activations were tried. For example a depth first search exploration strategy must backtrack if all of the activations were traversed from a given state. Also if a solution is found (a sequence of activations) and it should be applied to a copy of the initial model, identifying the activations in the process is required. Although these problems can be bypassed with working on the initial model itself and using an appropriate strategy, it limits the capabilities of an actual tool, hence it remains a challenge. The problem of comparing activations is very similar to comparing states, thus encoding activations is a more feasible approach.

3.3 Related work

Rule-based design space exploration frameworks Model checking approaches to analyze graph transformation systems are similar to DSE as they also perform state space exploration. One can categorize them as *compiled approaches* such as [34, 35, 36, 37, 38], which translate graphs and graph transformation rules into off-the-shelf model checkers to carry out verification, and *interpreted approaches* like [39, 40, 41], which store system states as graphs and directly apply transformation rules to explore the state space, similarly to our approach.

In [6] the state space explored by the GROOVE framework is stored as a structured graph model that can be queried using logical expressions. This approach allows the evaluation of trajectories using cost functions defined after the exploration and even the combined assessment of multiple solutions.

It is common in these approaches that they place emphasis on exhaustive traversal (e.g. by optimizing the storage of individual states), while we aim at finding solutions quickly as stated in subsection 3.2.2.

In [4] the T-Core framework is used for implementing typical meta-heuristic exploration strategies, such as hill climbing and simulated annealing using the transformation primitives of the framework while the operations are specified as graph transformation rules.

Other design space exploration The *DESERT* tool suite [2] provides model synthesis and constraint-based DSE for DSMLs with structural semantics using ordered binary decision diagrams for encoding and pruning the design space. [42] presents a generic DSE framework extending upon DESERT by supporting arbitrary analysis tools and includes model transformations for mapping design problems to intermediate and low-level formats.

The *OCTOPUS Toolset* [1] uses an intermediate representation for design problem specification and performs DSE using integrated analysis tools. It has been successfully applied to design software-intensive embedded systems [43].

The GASPARD *Framework* [3] is specifically focused on the design of massively parallel embedded systems and uses multilevel modeling where high-level UML models are automatically refined to allow design space exploration to evaluate performance characteristics through simulations.

An efficient design space exploration approach was also presented in [44] which is built on the *FORMULA* framework. The design problem is described using domain-specific languages and exploration is done with symbolic execution and automatic theorem proving by an SMT solver.

These are all compiled approaches, where the design problems are specified as models and model transformations are applied to derive inputs for third party analysis tools (e.g., SMT or SAT solvers). These analysis tools then perform the exploration and propagate the results back to the original model. However, as the analysis tools are usually used as black boxes when exploring the design space, they cannot be easily extended to support conceptually novel exploration algorithms (e.g., NSGA-II).

[45] presents a framework for the automatic deployment of software components to hardware architecture that uses design space exploration to find deployment alternatives that offer near-optimal reliability characteristics. The design problem consists of architecture models annotated with reliability-relevant properties, while the exploration uses an evolutionary algorithm to find possible alternatives. Unlike our approach, in this work (and also a follow-up paper [46]) global constraints are set as hard selection criteria to prevent the exploration (optimization) of invalid solutions.

Schätz *et al.* [47] developed an interactive, incremental process using declarative transformation rules for driving the exploration. The rules are modified interactively (user guided) to improve the performance of the exploration.

Chapter 4

Meta-heuristic techniques for optimization problems

This chapter summarizes problem-independent (meta-heuristic) search-based techniques which can be applied to a broad range of optimization problems. Section 4.1 overviews local search techniques, global search techniques are described in section 4.2. Finally, section 4.3 presents challenges of multi-objective optimization.

Optimization seeks the best solution from all feasible solutions. For example, the traveling salesman problem is a famous optimization problem, where the shortest path is searched in a graph from a specific starting node, touching all nodes and finishing with the starting node.

As stated in [48] an optimization problem can be defined as a triple $OOP = (S, C, f)$, where (1) S is the search space usually defined over a finite set of decision variables $X_1, X_2, \ldots X_n$, (2) C is a set of constraints between the variables and (3) $f : s \to \Re$ is a function which maps all possible states to a real number. The goal is to find a solution $s \in S$ such that $f(s) > f(s'), \forall s' \in S$ in case OOP is a maximization problem.

To solve optimization problems many techniques emerged, which are frequently classified as local search techniques or global search techniques.

4.1 Local search techniques

Local search techniques aim to find the best solution by traversing the solution candidates using single steps [32]. Usually, the state space of the optimization problem can be represented as a directed graph, where nodes denote solution candidates and edges represent the available steps to new feasible solutions. Considering the traversal strategy (i.e. how to choose the next step to take) the local search algorithms can be classified in two categories:

- **Systematic local search techniques** include depth first search (DFS) and breadth first search (BFS). These techniques explore the state space in a predefined pattern. They have many different variations (like iterative deepening depth first search). Usually randomization is used to avoid determinism.
- **Guided local search techniques** use knowledge from either previously explored states (e.g. cost of the path) or from prediction (e.g. estimation of solution distance) to decide the

next state to unfold. They are often based on systematic searches. For example, the greedy search which always goes to the most promising direction ignoring previously explored states uses a DFS, while A* is based on BFS, keeping in mind all of the previously found states.

A well-known guided local search technique is **hill climbing** [32] which is a greedy algorithm. It follows the next steps:

1. Start from a random initial solution.
2. Generate all neighbour solution (stochastic hill climbing [32] creates only several alternatives).
3. If none of them are better then it stops returning with the actual state.
4. Choose the best neighbour solution to continue with.

A possible state space traversal is shown on Figure 4.1. Hill climbing can stuck in a suboptimal solution by converging to a local maximum. To overcome this situation variations of hill climbing (e.g. random-restart hill climbing [32]) can be used.

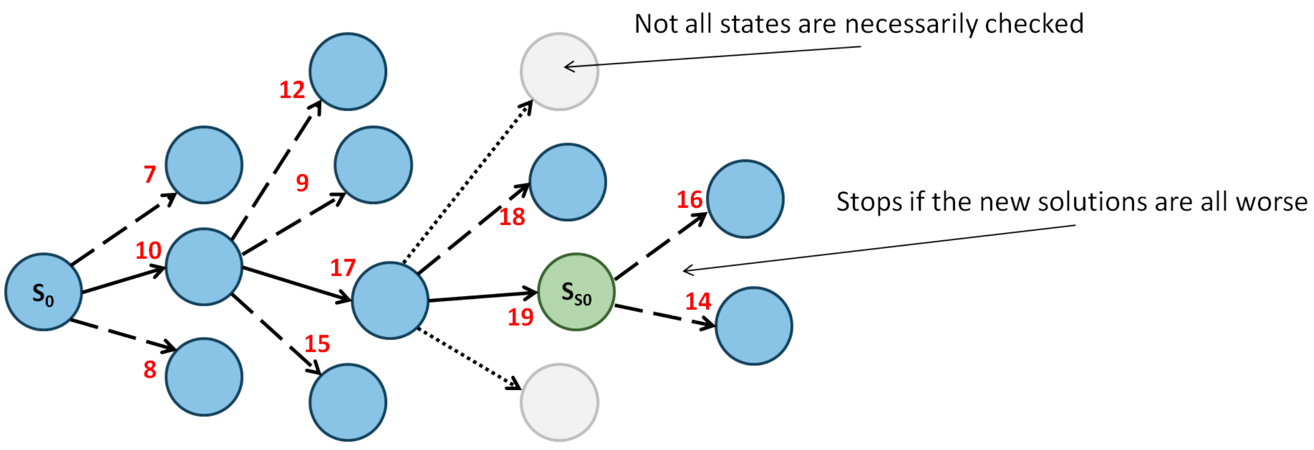

Not all states are necessarily checked
Stops if the new solutions are all worse
S_0
S_{S0}
12
7
9
10
17
18
16
8
15
19
14

Figure 4.1: Trajectory of the hill climbing algorithm

4.2 Global search techniques

The main difference between global and local search is that while local search explores along a single path, global search simultaneously traverses the state space along multiple trajectories. By looking for solutions in more than one direction it avoids stucking in a suboptimal region of the state space and can find the global maximum or minimum where a local search could fail. However, global search needs more memory and computational power and is harder to implement. There are many global search techniques, this report provides an overview on: swarm-based and genetic algorithms.

4.2.1 Swarm intelligence

Swarm intelligence is a collection of global search techniques which takes inspiration from the collective behaviour of social insects such as ants, bees and wasps, as well as from other animal societies such as flock of birds or schools of fish [48]. Although these animals are simple individually, they can solve complex tasks with cooperation. Ants can find food source in a large area, while bees can explore huge fields of plants with flowers. The main characteristics of swarm intelligence are: 1) individuals or agents are many in number, 2) they all do simple tasks, 3) they communicate with each other along the exploration and 4) they are decentralized and self-organized without a mastermind.

In solving optimization problems all individuals represent a feasible solution and modify it based on simple mechanics and communication with near mates. For example, in **ant colony optimization** [49] ants are individually modifying the solution at hand, while choosing its path with a probability calculated from *attractiveness*, a heuristic implying the desirability of the step, and *trail level*, which indicates how many ants passed that route recently.

Other notable algorithms are the particle swarm optimization [50], the bee colony algorithm [51] and the grey wolf optimizer [52].

4.2.2 Genetic algorithms

Genetic algorithms are global search methods mimicking the process of natural selection [12]. It performs an exploration by creating and evolving finite populations of candidate solutions using selection and genetic (mostly random) operators. Figure 4.2 presents a general process of the genetic algorithms:

1. Create an initial set of solutions. Usually, a random function is used for that purpose.
2. Generate new solutions from the initial solutions using mutation and crossover operations. A mutation operation modifies a single solution, while a crossover operation merges two solutions into new ones.
3. Group the initial and the new population into one set.
4. Evaluate solutions based on the **fitness function** which can calculate the achieved objective values on them.
5. Sort solutions based on their fitness value and select the best ones (the first half).
6. Start over from step 2 using the newly selected solutions as an initial population.
7. Usually, the process stops after a pre-defined number of iterations.

There are number of challenges when implementing a genetic algorithm. 1) Diversity of the population must be maintained, as if the solutions are too close to each other the algorithm can degrade to a random local search nullifying the advantages of global search techniques. Maintaining diversity should be carried out by the selection operator and the population generator (e.g. deleting duplicates, filtering or correcting unfeasible solutions). 2) Crossover operations should create new individuals keeping the good characteristics from the parents while omitting the bad ones. 3) The initial population must be healthy in diversity and feasibility, and it should lack bad characteristics if possible, as the population might be unable to grow out those.

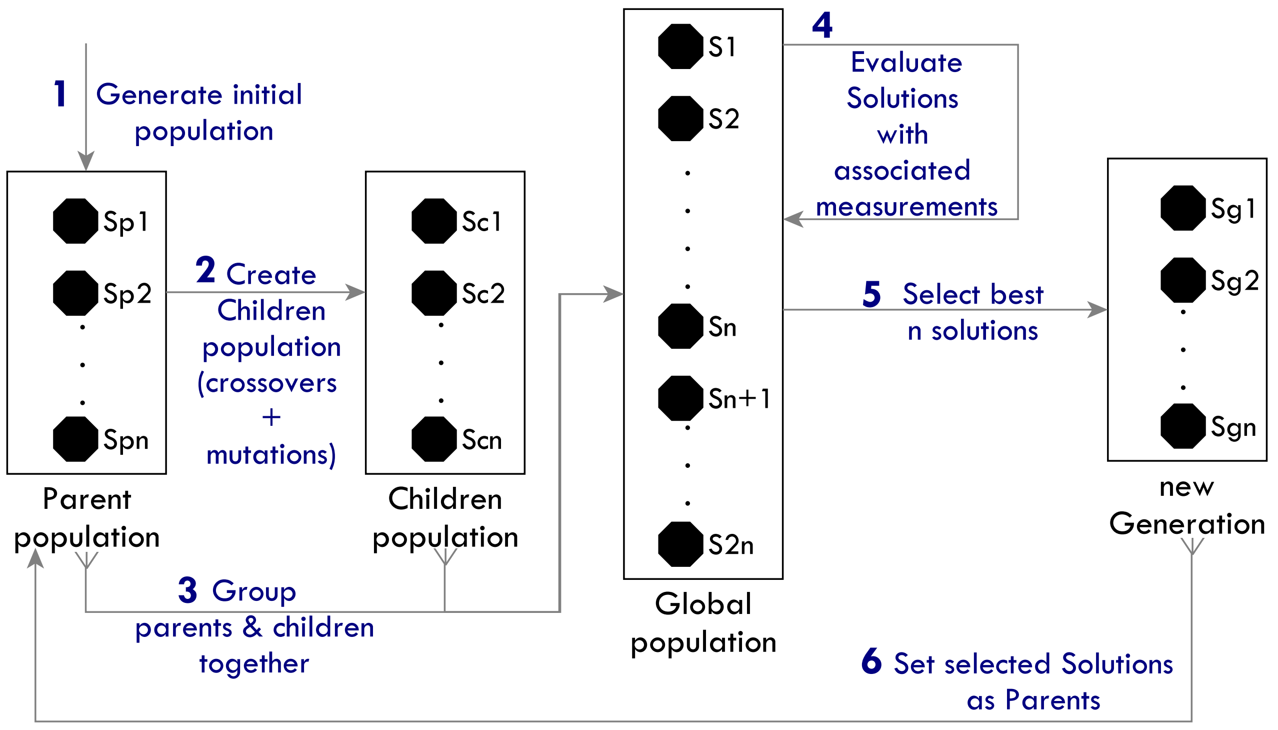

1 Generate initial population
Sp1
Sp2
Spn
Parent population
2 Create Children population (crossovers + mutations)
Sc1
Sc2
Scn
Children population
3 Group parents & children together
S1
S2
Sn
Sn+1
S2n
Global population
4 Evaluate Solutions with associated measurements
5 Select best n solutions
Sg1
Sg2
Sgn
new Generation
6 Set selected Solutions as Parents

Figure 4.2: Overview of genetic algorithms

4.3 Multi-objective optimization

A multi-objective optimization problem is an optimization problem with multiple objective functions. For example, minimizing response time and maximizing resource utilization is both important in business processes. Multi-objective optimization requires different techniques compared to regular (single-objective) optimization since objectives can be contradictory. Minimizing time and cost while maximizing quality at the same time is impossible in many application domains. Decreasing the invested money leads to bad quality, while reducing the available time might raise costs. Therefore optimization should find the best trade-offs between those objectives.

A multi-objective optimization problem $MOOP = (S, C, O)$ can be defined in the same way as single-objective optimization $OOP = (S, C, f)$ defined at the beginning of this chapter. The only difference is that there are multiple fitness function $\{o_1, o_2 \ldots o_n\} \in O, o_i : s \rightarrow \Re$ instead of a single one f.

If all objective functions O are for maximization, a feasible solution s_1 **dominates** another feasible solution s_2 ($s_1 \succ_O s_2$), if and only if, s_1 is better than s_2 for at least one objective, while s_2 is not better than s_1 regarding all the objectives in O [12]:

$$\exists o_j \in O: \ o_j(s_1) > o_j(s_2) \ \wedge \ \nexists o_i \in O: \ o_i(s_2) > o_i(s_1) \tag{4.1}$$

The solution for a multi-objective optimization problem is the **Pareto front** instead of a single solution. The Pareto front S_P is a set of solutions which dominates all the other solutions (Equation 4.2), but do not dominate each other (Equation 4.3).

$$\forall s_i \in S_P,\ \forall s_j \in S \setminus S_P : \quad s_i \succ_O s_j \tag{4.2}$$

$$\forall s_i, s_j \in S_P : \quad s_i \not\succ_O s_j, s_j \not\succ_O s_i \tag{4.3}$$

The possible solutions can be illustrated in a coordinate system, where every axis represents an objective and a point is a solution with the corresponding objective values. The possible solutions form a bounded or a semi-bounded shape (assuming the objectives to be maximized have a maximum value) and the best solutions can be found on the front of this shape which is the Pareto front as it can be seen on Figure 4.3 (for two objectives). Solutions are considered to be equal as they non-dominate each other, therefore a domain expert can decide which solution should be used by analyzing the trade-offs.

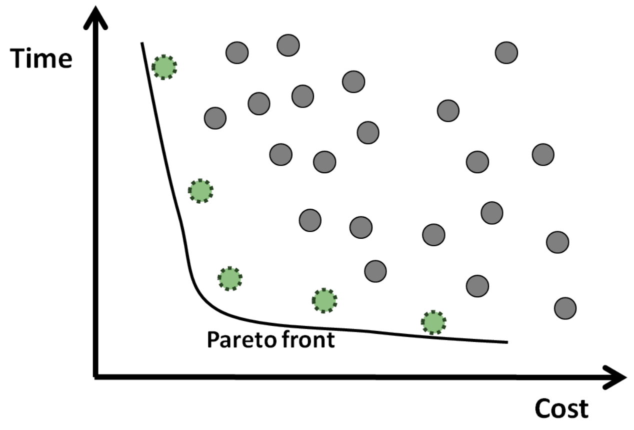


Figure 4.3: The Pareto front of a solution set with two objectives to minimize

Using global search techniques is a popular option for solving multi-objective optimization as they work with multiple solutions and they aim to find the Pareto front. In contrast local search techniques may find only one element of the Pareto front.

Non-dominating Sorting Genetic Algorithm (NSGA-II) [12] is a well-known genetic algorithm used in multi-objective optimization. Its main contribution to the genetic algorithms is selecting the individuals into **fronts** by using the domination function. The first front F_1 is the Pareto front of the population P, the second front F_2 is the Pareto front of $P \setminus F_1$, the third front F_3 is the Pareto front of $P \setminus (F_1 \cup F_2)$, etc. as shown on Figure 4.4. After the grouping it selects the first k front $(F_1, F_2, \dots F_k)$ to the next population. As the next population needs an exact number of individuals, usually a few solutions need to be dropped from the front F_k. For this purpose it calculates the **crowding distance** of the individuals, which is a similar number of the distance of the closest individual in the same front, and chooses the individuals with highest crowding distance maintaining diversity.

Figure 4.5 shows an overview of the NSGA-II. The aim of NSGA-II is to find a set of Pareto optimal solutions in a single run. As a genetic algorithm, NSGA-II performs a global exploration of the search space by making and evolving finite populations of candidate solutions using selection and genetic operators. The output of the algorithm is a set of the fittest solutions (i.e. the Pareto front) produced along all generations. When searching for solutions to a problem using Pareto optimality (i.e. multi-objective), the search yields a set of solutions that are not-dominated [53]. The decision maker can select one of the fittest solutions according to his/her preference (e.g. the solution which satisfies all the requirements).

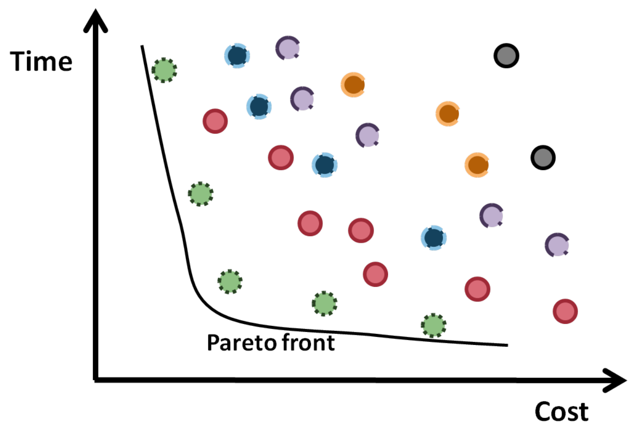


Figure 4.4: NSGA-II sorts the population into fronts

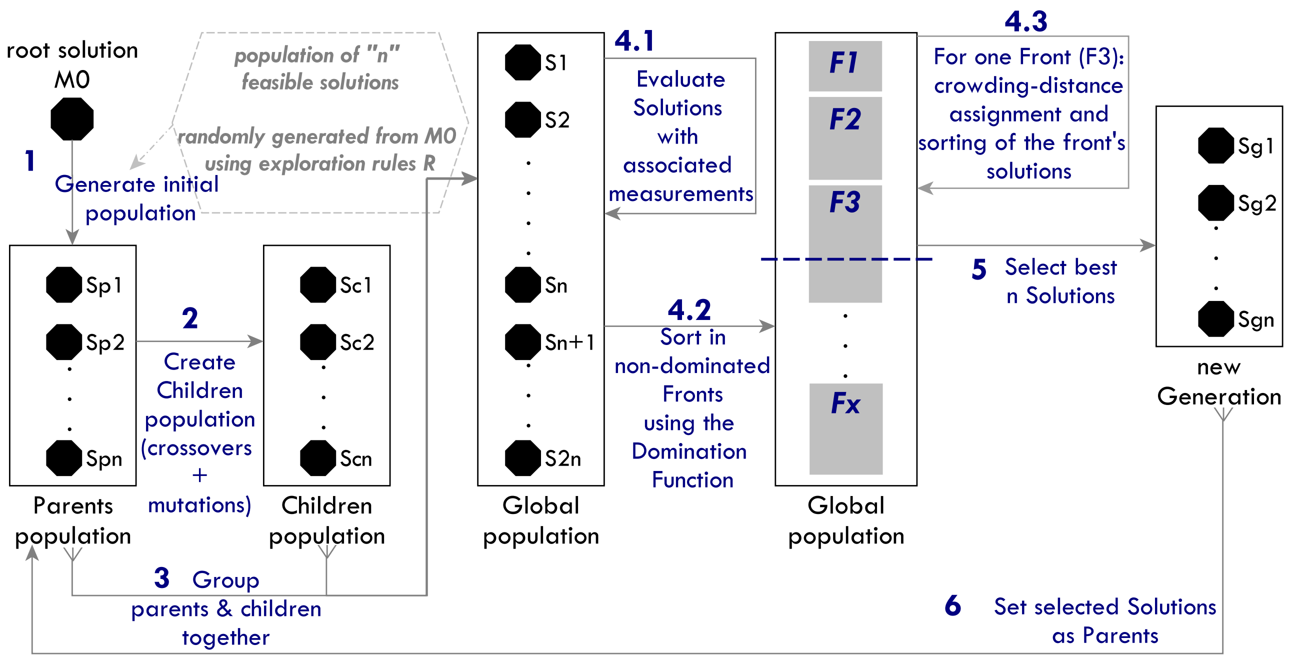

root solution M0
population of "n" feasible solutions
randomly generated from M0 using exploration rules R
1 Generate initial population
Sp1
Sp2
Spn
Parents population
2 Create Children population (crossovers + mutations)
Sc1
Sc2
Scn
Children population
3 Group parents & children together
S1
S2
Sn
Sn+1
S2n
Global population
4.1 Evaluate Solutions with associated measurements
4.2 Sort in non-dominated Fronts using the Domination Function
F1
F2
F3
Fx
Global population
4.3 For one Front (F3): crowding-distance assignment and sorting of the front's solutions
5 Select best n Solutions
Sg1
Sg2
Sgn
new Generation
6 Set selected Solutions as Parents

Figure 4.5: Overview on NSGA-II process

However, from a practical MDSD viewpoint, constrained multi-objective optimization is important in the context of DSE. This is due to the fact that the primary objective of a DSE approach is to find valid solutions that satisfy *all the requirements* of the underlying problem. Other objectives, such as reducing the cost of obtained solutions, could not add an effective value to the optimization results unless the obtained solutions are valid. Hence, we adapt the constraint-handling strategy with NSGA-II that was proposed in [12, 54].

In the presence of constraints (i.e. requirements), a solution can be either valid (i.e. it satisfies all the constraints) or invalid (i.e. it does not satisfy all the constraints, totally or partially). Therefore the domination function in Equation 4.1 is modified as follows: A solution s_1 is said to **constrained-dominate** a solution s_2, if one of the following conditions is true:

1. s_1 is valid and s_2 is not.
2. Both solutions, s_1 and s_2, are invalid, but s_1 has *a smaller overall constraint violation.*

3. s_1 and s_2 are valid and s_1 dominates s_2 with the usual domination function (Equation 4.1).

The idea behind this constrained-domination strategy is that, on the one hand, any valid solution has a better non-domination rank than any invalid solution. On the other hand, valid solutions are ranked into their non-domination level based on their associated quality as measured by the values of objective functions. And, invalid solutions are ranked into their non-domination level in descending order according to the value of their associated constraint violation.

Multi-objective optimization in model driven engineering Multi-objective optimization techniques are widely used in Model Driven Engineering (MDE) field [55, 9, 56, 11]. Recently, Kessentini *et al.* [57] proposed an MDE-based framework for easing the adoption of search-based techniques (such as genetic algorithms) to MDE problems. In this work, the authors describe the logic layer of their MDE-based framework based on previous experiences in using SBSE (Search-Based Software Engineering) in hand-crafted applications. However, the realization of the framework is only planned as future work. Moreover, it is not clear how the proposed framework can be adopted for rule-based DSE. Etimaadi and Chaudron [56] proposed the AQOSA tool which uses a model-based approach to evaluate component-based software architecture quality. It uses multi-objective evolutionary algorithms to automatically optimize software architecture design with regard to multiple quality objectives, such as response time, processor utilization, safety, etc.

Despite the popularity of applying search-based techniques for MDE problems, to the best of the authors' knowledge, there is no existing work in the literature dealing with rule-based DSE using multi-objective optimization techniques. Indeed, existing work on Automatic Design Space Exploration (ADSE) using multi-objective optimization techniques are not rule-based DSE, and they are proposed for specific domain problems. For example, Calborean *et al.* [58] proposed recently the FADSE (Framework for ADSE) for DSE of computer systems using different multi-objective search-based algorithms. In this paper, the authors compare the results produced by different genetic algorithms for optimizing the parameters of the Grid ALU Processor (GAP) microarchitecture and the post-link code optimizer GAPtimize. In their framework, application-specific rules that describe existing knowledge can be defined and used as constraints to constrain the DSE process [59]. A similar work is performed by Bolchini *et al.* [60]. Bolchini *et al.* propose a framework based on the multi-objective genetic algorithm NSGA-II for DSE of reliable Field Programmable Gate Array devices.

Chapter 5

An introduction to concurrent programming

The aim of concurrent programming is to accelerate computationally intensive tasks. It is achieved by using multiple computational units (exploiting the multiple cores of modern processors or the power of graphical processing units), each working only a part of the problem, and producing the result by synchronization and adding up the partial results. For example, these problems can be algorithmic tasks like sorting or matrix operations, but web servers also use concurrency to efficiently handle incoming requests.

In software development **threads** are used as independently managed program instructions and are executed separately by the scheduler of the operating system. All threads have its own state, regarding the data it is working with and the actual instructions it executes, but they share the memory and other resources like I/O and can also share the same flow (program code).

However, using **shared resources** concurrently usually needs special care. While reading data in parallel from the shared memory is error-free, writing at the same time can lead to inconsistency. If a shared piece of data is being written by a thread and multiple CPU instructions are needed for a consistent change in that particular data structure, an intermediate read from an other thread can lead to unpredictable results.

A part of a sequence of program instructions is called a **critical section**, when it accesses a shared resource. There are two ways to handle these critical sections and avoid inconsistency:

- **Mutual exclusion** (mutex in short) allows only one thread to run in the critical section blocking all other threads from executing the same code until the first thread is finished. Thus it enables to execute the critical section as an atomic step. There are multiple ways to implement mutual exclusion such as semaphores, locks and monitors.
- **Interlocked operations**, which are supported in the hardware level, are atomic operations which would normally use more than one instruction. For example, a shared integer value can be tested (read) and be written in a single operation, avoiding the reader-writer problem explained above.

It is a common practice to use locks to protect critical sections as it is relatively easy and maintainable. However, using locks makes the algorithm a **blocking algorithm** which have the following disadvantages: 1) **deadlocks** can occur, if threads waiting for each other to release a lock, 2) a thread can **starve**, if it is unable to acquire a specific lock as other threads constantly blocking it to do so, 3) acquiring a lock always has an overhead, even if no other threads try to run in that critical section and 4) a thread can fail to release a lock if it dies for some reason, blocking all other threads from the critical section until intervention. Therefore, this technique is inadvisable in critical systems.

There are two types of non-blocking algorithms by definition: **wait-free** [61] and **lock-free** [62]. Wait-free algorithms ensure that every thread will succeed within a limited time in executing the critical section, while lock-free guarantees this to only at least one thread. Usually, such algorithms are hard to achieve and there are a few drawbacks: 1) the code can be harder to read and maintain, 2) data structures may have to be redesigned to be able to use non-blocking algorithms and 3) any wait-free or even lock-free solutions (especially when universal techniques are used) won't necessarily outperform a well made lock-based implementation [63, 64].

There is a major design pattern to create lock-free implementations: 1) copy the required data, if it can be modified by other threads, 2) do the work on it, 3) modify shared data through an interlocked operation and finally 4) be ready to retry, if the operation has failed, just like in an optimistic concurrency control. Following this pattern the solution will also be **cache friendly**, which means that a high ratio of the data to be processed will be in the thread's (the core's) cache and don't have to wait for synchronizing it with the other thread's caches, which would drastically reduce performance. However, making data structures suitable for interlocked modification is often a hard task.

Speedup factor means how many times faster a parallel version of an algorithm is than running with one thread. It is usually used for evaluating the efficiency of parallelism. To determine the speedup factor, the run time of the algorithm is measured with one thread and n threads. Then the speedup factor $SU_n = T_1/T_n$ where T_i is the run time of the algorithm with i threads. The theoretical maximum of the speedup factor is the number of threads used $SU_n^{max} = n$, as two thread can finish the same work twice as fast, three threads thrice as fast, etc. However, it is an over-estimation if there are parts of the algorithm is sequential regardless the number of used threads. **Amdahl's law** gives a better estimation for these cases by defining the fraction of the algorithm which is sequential $B \in [0, 1]$. It declares that the theoretical minimum of a run time with n threads is $T_n^{min} = T_1(B + (1 - B)/n)$, hence the theoretical maximum of the speedup factor is $SU_n^{max} = T_1/T_n^{min}$.

Chapter 6

Multi-objective optimization for rule based design space exploration

This chapter describes our approach to integrate multi-objective optimization techniques by using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [12] to drive rule-based design space exploration. In the following, section 6.1 defines rule-based DSE as an optimization problem, section 6.2 overviews the approach, while section 6.3 describes how the objective can be captured and section 6.4 presents the genetic operators for rule-based DSE.

6.1 Rule-based design space exploration as an optimization problem

Rule-based design space exploration $RDSE = (M_0, G, R, GC)$ can also be seen as an optimization problem. The state space S is the set of possible states of the the initial model M_0 using the set R of rules and is shown on Figure 3.3. The constraints C are the global constraints GC and also implicitly the rules R. The objectives O (or f in single objective optimization) is defined by the goals G, although the definition of design space exploration in section 3.2 uses a single binary objective: it is a solution or not.

Most local search techniques can be applied in rule-based design space exploration. Different strategies specify the order of choosing the next rule activation to execute. Depth first search applies a random activation after each step and backtracks if there are no more activations to apply, while breadth first search executes every n long trajectory after all $n-1$ long was tried, thus systematic local searches can be used easily in design space exploration. Guidance can rely on calculating objectives on the actual model and the traversed trajectory, while priorities assigned to rules and rule dependency analysis [65] can also be used for directing the traversal.

Applying global search techniques to rule-based design space exploration is still a research field and wasn't carried out until now. In the following we present our adaptation of the NSGA-II to the problem of rule-based DSE (which demonstrates the feasibility of global search techniques in rule-based DSE) by defining our implementation of the following elements:

- Representation of candidate solutions (individuals) in section 6.2.
- Optimization objectives in section 6.3.

- Genetic operators used to explore the search space in section 6.4.

6.2 Optimization inputs, procedure and results

Figure 6.1 shows an overview of our approach. Given an initial model M_0 belonging to a domain model DM, a set of requirements to be satisfied Req, and a set R of exploration rules, our multi-objective search-based approach for rule-based DSE explores the design space starting from M_0 by maintaining finite populations of the most promising design candidates with respect to the set of requirements Req and other optimization criteria O. In our approach, each individual of evolving populations is a map of a candidate model M_n obtained as a sequence of rule executions, denoted as $\vec{r_n}$, leading from the initial model M_0 to M_n: $\vec{r_n} = M_0 \xrightarrow{r_1,m_1} M_1 \ldots \xrightarrow{r_n,m_n} M_n$, where $r_i \in R$ and m_i is the match (or activation) of r_i in M_i.

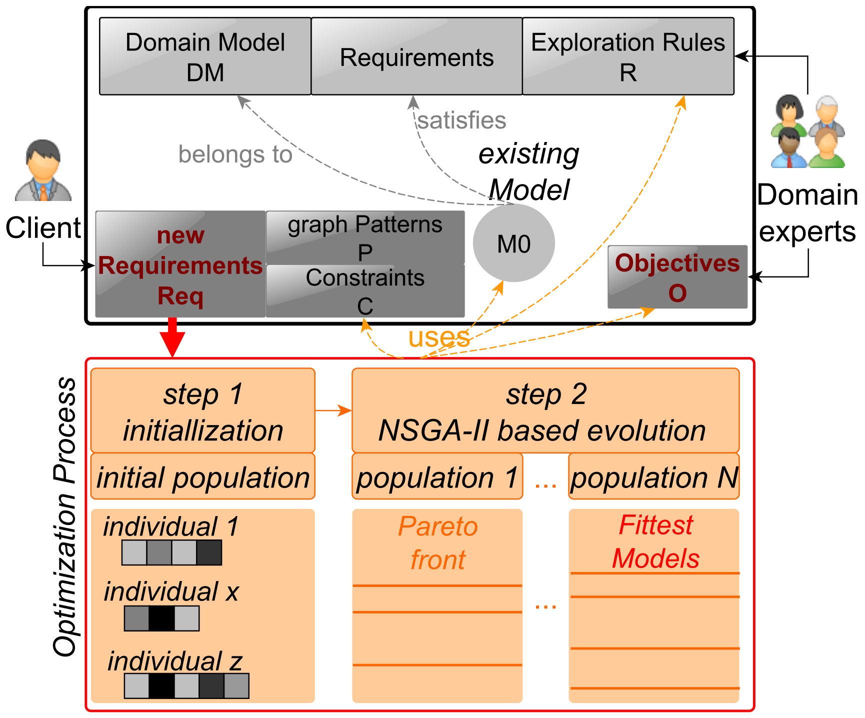


Figure 6.1: Approach Overview

The generation of the initial population starting from M_0 (*step 1* of the optimization process in Figure 6.1) is achieved by applying random (but executable) exploration rules from R on M_0. The second step of the optimization process in Figure 6.1 is based on the NSGA-II as explained with Figure 4.5, where populations evolve by applying mutation and crossover operations which change and/or extend existing rule execution sequences to yield new individuals. The requirements are defined as soft constraints C that are captured by model queries (see Figure 2.6) while other optimization objectives can be derived both from models or rule execution sequences. Each constraint c_i in C is assigned to a specific weight w_i describing the relative importance of this constraint, thus each violation of c_i (calculated as a match of the corresponding graph pattern) will be weighted accordingly.

The **multi-objective rule-based design space exploration problem** can be defined as: $MRDSE = (M_0, C, O, R)$ where M_0 denotes the initial model, C denotes the (structural and attribute) constraints characterizing valid design candidates, O is a set of numerical objectives which need to be optimized, while R is a set of exploration rules describing valid evolutions of the design.

A **candidate solution** of a DSE problem is a pair $S_{cand} = (M_{cand}, \vec{r}_{cand})$ where (1) the candidate model M_{cand} fulfills all (or some) constraints in C and (2) it is reached from the initial model M_0 by a sequence of rule executions $\vec{r}_{cand}$. A **model objective** o_m is evaluated for a candidate solution S_{cand} on the candidate model M_{cand} while a **trajectory objective** o_t for S_{cand} is evaluated on the sequence of rule executions $\vec{r}_{cand}$ leading to S_{cand}. Considering our running example, reducing the number of constraint violations is a model objective or increasing utilization, while reducing the cost associated to the trajectory $\vec{r}_{cand}$ is a trajectory objective.

By definition, a candidate solution is called **feasible** if its rule execution sequence $\vec{r}$ is executable. In a genetic approach for rule-based DSE, infeasible solutions can occur when applying genetic operators (e.g. crossover) on existing feasible solutions. However, in our approach, such infeasible solutions are automatically corrected (or truncated) to guarantee their feasibility, or omitted if they cannot be corrected.

Consider $newHostInst(HI); allocate(AI, HI, RR)$ (see Figure 2.7) which is a rule execution sequence that aims to create a new host instance HI and then allocate an application instance AI to it is not executable if there is no application instance AI to be allocated. To correct this infeasible sequence, the creation of a new application instance AI by rule $newAppInst(AI)$, should precede the $allocate(AI, HI, RR)$ rule. Otherwise, the sequence must be truncated so that it includes only the rule execution $newHostInst(CS1)$.

A feasible solution $S = (M, \vec{r})$ is defined as a **valid** solution if its associated model M fulfills *all constraints* in C: i.e. $\forall c \in C\ \nexists m : c \mapsto M$.

6.3 Objectives

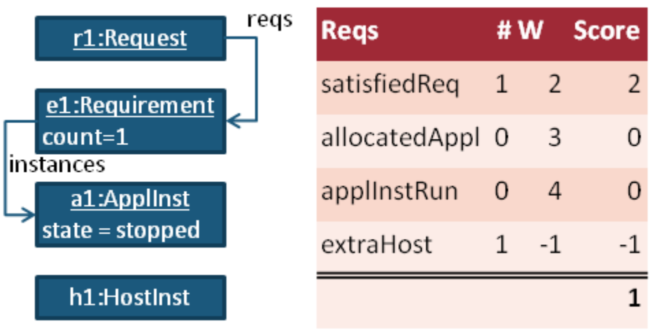
The objective of constraints fulfillment As constraints are formalized by graph patterns, in order to evaluate the degree of well-formedness constraints are met or ill-formedness constraints are violated in a candidate model M_n, we use the weighted sum of the number of matches for the corresponding graph patterns P.

For instance, let us evaluate the constraints of Figure 2.6 on a sample model M of Figure 6.2. For this model, our optimization approach will return that the graph pattern *satisfiedReq* has 1 match and the graph pattern *extraHost* has 1 match. In our example, the weight of *satisfiedReq* is set to $w_1 = 2$ and the weight of *extraHost* is set to $w_2 = -1$, therefore, the degree of constraint violations in M is: $ConstViol(M) = 1 \times w_1 + 1 \times w_2 = 1$.

Formally, let $matches(p, M)$ return the number of matches of the graph pattern $p \in P$ in the model M, and let w_p denote the weight associated to the constraint described by p (where w_p is a positive value for well-formedness constraints and a negative value for ill-formedness constraints). Then our objective of constraints fulfillment is defined as follows:

$$ConstFulfillment(M) = \sum_{\forall p \in P} w_p \times matches(p, M) \tag{6.1}$$

The primary optimization objective of our approach, as explained in section 6.2, is to maximize $ConstFulfillment(M)$, i.e. to maximize the fulfillment of positive constraints and minimize the degree of negative constraint violations.



Reqs	#	W	Score
satisfiedReq	1	2	2
allocatedAppl	0	3	0
applInstRun	0	4	0
extraHost	1	-1	-1
			1

Figure 6.2: Objective of constraint fulfillment

Model-specific objectives Our approach allows to define *domain-specific objectives* captured by graph patterns over the underlying model. Thanks to the incremental query evaluation approach (EMF-IncQuery), the re-evaluation of such model objectives is instantaneous upon model changes.

In the context of our motivating example in this report, we define the model-specific objective of maximizing the utilization of compute servers ($CSUtil$), so that the best utilization of memory or storage is incorporated for each server.

Let $Util(CS_i)$ return the (normalized) resource utilization for the computer server CS_i while the system-level utilization of computer servers $CSUtil$ for a solution $S = (M, \vec{r})$ is defined as the mean of the utilization of each individual computer server element in M:

$$CSUtil(S) = \frac{1}{n} * \sum_{j=1}^{j=n} Util(CS_j) \tag{6.2}$$

In the above equation, n is the number of computer server instances in the underlying model M.

Rule sequence objectives Two valid solutions can be achieved via two different feasible sequences of rule executions. Therefore, we may define objectives specific to rule execution sequences to evaluate the cost incorporated in achieving a valid solution along a specific path.

For this purpose, we define the cost of a feasible solution $S = (M, \vec{r})$ as the sum of costs of all rule executions in its sequence of rule executions $\vec{r}$:

$$Cost(S) = \sum Cost(M_{i-1} \xrightarrow{r_i, m_i} M_i) \quad \forall M_{i-1} \xrightarrow{r_i, m_i} M_i \in \vec{r} \tag{6.3}$$

$Cost(S)$ takes its values in the interval $[0..\infty[$. Minimizing $Cost(S)$ is an objective of our optimization approach. Computing the cost $Cost(M_{i-1} \xrightarrow{r_i, m_i} M_i)$ of a rule execution in $\vec{r}$ depends on three parameters to be defined by the domain experts:

- *Fixed cost:* the fixed cost C_b of the applied rule r_i; formally, $Cost(... \xrightarrow{r_i, m_i} ...)^F = C_b(r_i)$. As defined in Figure 2.7, creating an application instance will always have the same cost, which is $C_b(newApplInst) = 2$.
- *Match cost:* the match cost C_m which is associated to the match m_i in M_i; formally $Cost(... \xrightarrow{r_j, m_j} ...)^M = C_b(r_j) + C_{m_j}(r_j)$, where $C_{m_j}(r_j)$ returns the cost of r_j according to its match m_j. For instance, the cost of rule $newHostInst$ is specific to the matched host type element HT (as defined by the black circles in Figure 2.1).

- *Sequence cost:* the sequence cost C_s may depend on the position of the rule execution ... $\xrightarrow{r_i,m_i}$ ... in $\vec{r}$. For this purpose, we define the cost of such a rule execution as relative to its position in $\vec{r}$ on the same match m: $Cost(... \xrightarrow{r_k,m_k} ...)^S = position(r_k, m_k) \times Cost(... \xrightarrow{r_k,m_k} ...)^M$, where $position(r_k, m_k)$ returns the position of r_k application on the same match m_k in $\vec{r}$.

6.4 Genetic operators (mutation and crossover)

In this report, we define and use different types of mutation and crossover operators for exploring the design space. In our context, mutating a solution $S = (M, \vec{r})$ means to modify the sequence of rule executions $\vec{r}$, which is conceptually different from most genetic approaches used for DSE purposes. This can be achieved in different ways:

- *Add new rule execution:* a new sequence of rule executions $\vec{r'}$ is generated by selecting an appropriate exploration rule r' from R, that can be applied on M, and execute it: $\vec{r'} = \vec{r} + \{M \xrightarrow{r',m} M'\}$
- *Delete a random rule execution:* a new sequence of rule executions $\vec{r'}$ is generated by deleting a random rule execution re_i in $\vec{r}$, so that: $\vec{r'} = \{re_0, \ldots, re_{i-1}, re_{i+1}?, \ldots ?\}$. The question marks in the aforementioned sequence denote the execution rules which will be checked for executability after deletion is performed. Indeed, after deleting re_i, the executability of the new sequence $\vec{r'}$ is checked starting from the rule execution re_{i+1}, so that if re_{i+1} is not executable anymore it is then ignored (removed from the sequence), and so on for each re_{i+k}, $k > 1$.
- *Swap between two rule executions:* a new sequence of rule executions $\vec{r'}$ is generated by selecting a random rule execution re_i in $\vec{r}$, then selecting another rule execution re_j $(j > i)$ in $\vec{r}$, that can replace re_i and still executable, then swap between re_i and re_j so that: $\vec{r'} = \{re_0, \ldots, re_j, re_{i+1}?, \ldots\ ?, re_i?, re_{j+1}?, \ldots\ ?\}$. Similarly to the case of delete a random rule execution discussed above, the executability of the new sequence $\vec{r'}$ is checked starting from the rule execution re_{i+1}.

The crossover operators apply to two individuals represented by the sequences of rule executions of two parent solutions $S^1 = (M^1, \vec{r}^1)$ and $S^2 = (M^2, \vec{r}^2)$, and generate two new offspring individuals (children). Figures 6.3, 6.4 and 6.5 describe the three crossover operators that our optimization process uses:

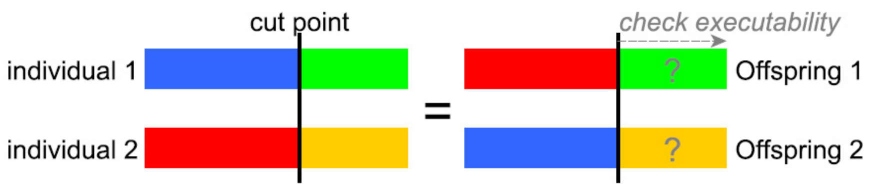


Figure 6.3: One-point crossover

- *One-point crossover (Figure 6.3):* a single crossover point on both sequences of rule executions of parents is selected. All rule executions beyond that point in either sequences of rule executions are swapped between the two-parent sequences. The resulting sequences of rule executions are the children.

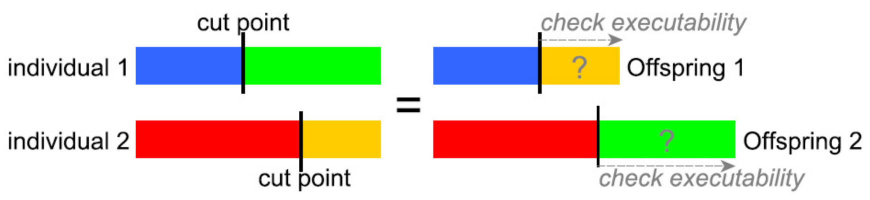


Figure 6.4: Cut-and-splice crossover

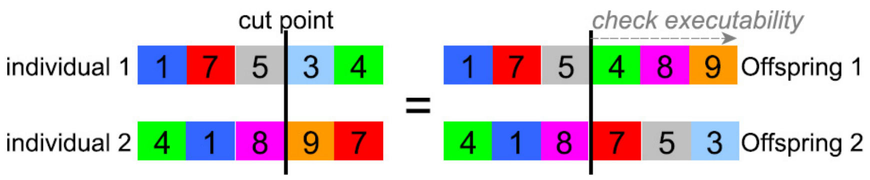


Figure 6.5: One-point permutation crossover

- *Cut-and-splice crossover (Figure 6.4):* cut-and-splice crossover is a variation of the one-point crossover where the difference is each parent's sequences of rule executions has a separate choice of crossover point. As a result, the children sequences of rule executions will have different length than that of their parents.

- *One-point permutation crossover (Figure 6.5):* in this crossover operator, every rule execution re_i in either sequences of rule executions (parents) will have an ID. This ID is based on the applied rule r_i and the match element m_i of re_i. Hence, two rule executions, r_i and r_j, can have the same id if they are applications of the same rule r on the same match m. Representing the rule executions by their IDs, a sequence of rule executions will have a permutation representation, as in Figure 6.5. With the one-point permutation crossover, one crossover point is selected on both sequences of rule executions of parents, from the first (second) parent the permutation is copied up to this point, then the second (first) parent is scanned and if the ID of the rule execution is not yet in the offspring, the rule execution is added.

For every crossover operation, our approach performs an automatic executability check of rule executions occurring after the cut point(s) in children sequences of rule executions. The correction mechanism is identical to that used in the delete and swap mutations that we described above.

Chapter 7

Architecture and prototype implementation

This chapter presents three more conceptual challenges about the details of the approach in section 7.1 and further insight about the implementation in section 7.2.

7.1 Conceptual challenges

Chapter 6 proposed an approach to integrate multi-objective optimization techniques by using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [12] to drive rule-based design space exploration. It defined how an individual can be interpreted in the context of DSE, what is a feasible and valid solution, how to capture objectives on the individuals and proposed a few mutation and crossover operations. However, additional conceptual questions need to be clarified:

1. How to encode individuals.
2. How to generate the initial population for the genetic algorithm.
3. How to apply global search techniques to VIATRA-DSE.
4. How to prepare VIATRA-DSE for parallelization.
5. How to exploit VIATRA-DSE's support on parallelization.

The next sections address these questions in this specific order.

7.1.1 Encoding of individuals

As stated in section 6.2 individuals are represented as sequences of rule activations. VIATRA-DSE creates activation codes and state codes, thus storing individuals as a list of activation codes seems an obvious choice. The built in encoder for VIATRA-DSE is domain independent and incremental [13]. Thus it can be used for any DSE problem and its incrementality makes it efficient for a wide range of problems as it recalculates only the parts of the code which have been modified by the rule execution. However, it has two shortcomings: 1) although it is incremental a domain specific state coder designed directly for a problem may outperform it by an order of magnitude and 2) the generated activation codes are unique, which makes crossover operations impossible to implement properly.

The latter issue is illustrated in Figure 7.1. It shows a part of a design space, where activations are encoded symbolically by the built in state coder to *different* Latin letters. Also there are two individuals (solutions) $S_1(a, e, k)$ and $S_2(b, g, n)$. If a *single point crossover* operation takes place between S_1 and S_2, the new individuals would be $S_3(a, g, n)$ and $S_4(b, e, k)$. When feasibility of these new individuals are checked, the last two activations for both solution will be omitted, because it is impossible to match an activation encoded as g in M_1 for solution S_3 and vice versa. Therefore, the crossover operation will only generate children with a truncated trajectory and will fail to fulfill its function.

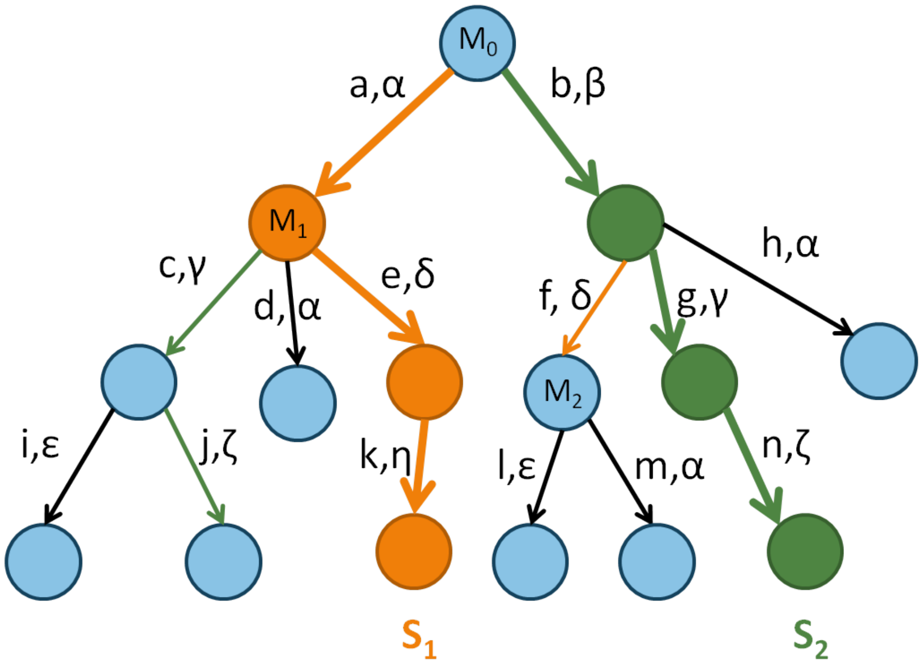


Figure 7.1: Encoding of activations

Nevertheless, a domain specific state coder can overcome this limitation. Most of the activations are independent from each other and executing one will make only a small set of activations to appear and/or disappear. For example, on Figure 7.2 *activation 1* is independent of the execution of *activation 2*. Hence, activations in the state space (e.g. the two *activation 1*) can have the same code, independently from the their source model state. Figure 7.1 shows such a state coder by symbolically encoding the activations into Greek letters. Using this state coder, the crossover operation can perform its function and the parents $S_1(\alpha, \delta, \eta)$ and $S_2(\beta, \gamma, \zeta)$ will produce children $S_3(\alpha, \gamma, \zeta)$ and $S_4(\beta, \delta)$. In the example, the child S_4 loses the last activation η from the parent S_1 as it is unavailable from M_2.

7.1.2 Generating the initial population for the genetic algorithm

Traditionally, genetic algorithms (and global search algorithms) start from a randomly generated population. As individuals are encoded by numbers it is fairly easy to implement an initial population generator by using random numbers. However, in rule-based design space exploration individuals are represented as sequences of rule activations, which is almost impossible to generate in the same way, and the resulting trajectory might be unexecutable as activations can depend on

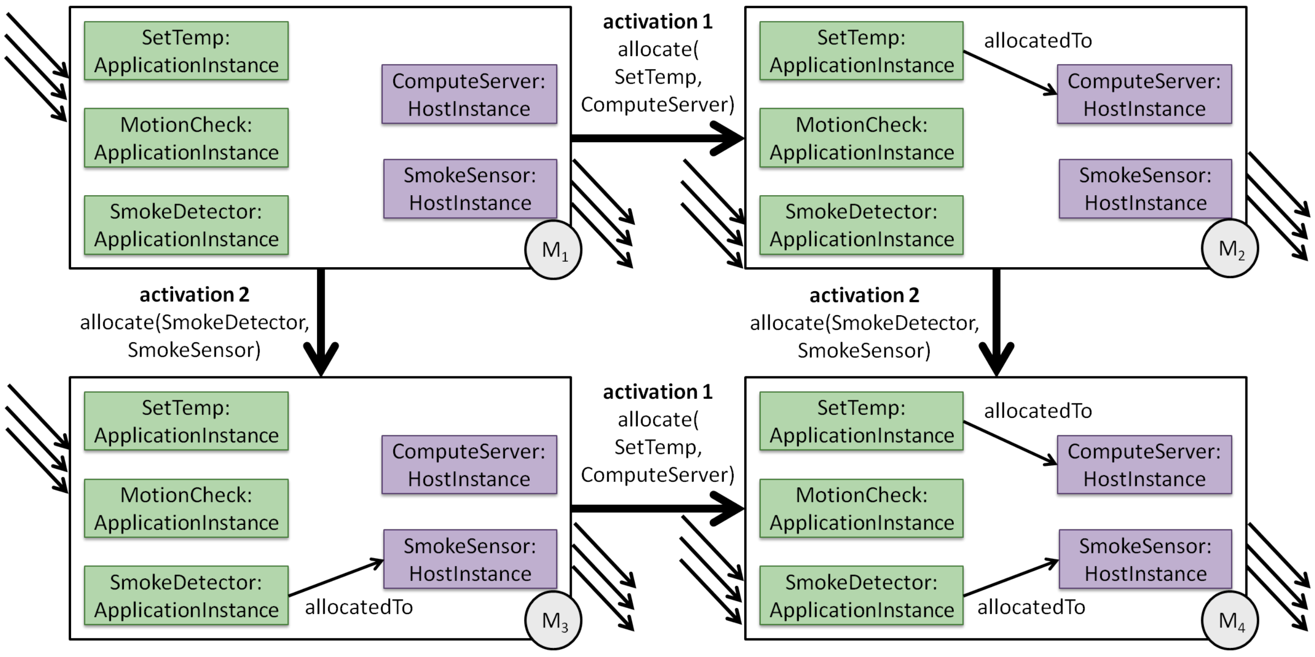

SetTemp: ApplicationInstance
MotionCheck: ApplicationInstance
SmokeDetector: ApplicationInstance
ComputeServer: HostInstance
SmokeSensor: HostInstance
M1
activation 1
allocate(SetTemp, ComputeServer)
allocatedTo
M2
activation 2
allocate(SmokeDetector, SmokeSensor)
M3
M4

Figure 7.2: Independence of activations

previous rule executions.

Thus, I propose to generate the initial population using other meta-heuristic exploration strategies such us breadth first search. The only restriction for the chosen strategy that it has to return with a predefined number of solutions due to the population size. I present two such implemented strategy: 1) breadth first search and 2) priority-based local search.

- Breadth first search is executed in the well-known way: try all possible one-long trajectories, after that all two-long trajectories, etc. In addition, a trajectory is chosen to the initial population randomly corresponding to a predefined probability. It stops when enough trajectory has been chosen.

- Priority-based local search is basically a depth first search with two additional inputs: 1) numbers as priority assigned to the rules and 2) a set of constraints given as graph patterns. Priorities are used to guide the exploration. It tries only the activations with locally the highest priority, choosing randomly between activations with the same priority, and backtracks if they are all explored. A trajectory is selected into the initial population if all the given constraints are satisfied by the model. To keep the initial population diversive, the exploration is restarted every time a solution candidate is found by backtracking to the initial model. The stop condition is the same as previously, it stops when enough trajectory has been chosen.

7.1.3 Applying global search techniques to VIATRA-DSE

VIATRA-DSE, a rule-based design space exploration framework supports local search based state space traversal, i.e. a traversal strategy needs to be defined by the next, yet unexplored activation it takes in the already traversed design space. Until the deciding of the next step, traversal among the explored states is possible via basic operations like backtracking and executing an activation again from the corresponding model state.

However, global search techniques (like genetic algorithms) need several or even hundreds of separate local search units. For example, ants of ant colony optimization randomly explore their immediate environment based on heuristics and pheromone trails. Thus a solution is needed to carry out global search as a reduction to a single local search.

I propose the following solution to this problem:

- ***Step 1.*** Generate an initial set of trajectories (or individuals) through a local search as stated in section 7.1.2.
- ***Step 2.*** Store the context of the individuals, i.e. the sequences of rule activations and calculated fitness values. During the initial selection the local search have to store these pieces of information as they are needed in the next steps.
- ***Step 3.*** Carry out additional calculations, which are necessary from a global point of view such as communication between the individuals (e.g. fading of pheromone in case of ant colony optimization), dropping individuals with bad characteristics based on fitness values or generating new ones. Also any guidance from a global point of view should be executed in this step.
- ***Step 4.*** For each individual: 1) load its context and execute its trajectory, 2) execute one step based on the rules defined by the global search technique, 3) calculate its fitness values and save its context (as in setp 2.).
- ***Step 5.*** Continue from *step 3.* until a pre-defined stop condition is satisfied. For example, a stop condition can be an upper bound on the number of iterations, the exploration found a solution within a certain range of a predefined limit or unable to reach better solutions.

The genetic algorithm can be interpreted to the above solution by specifying *step 3.* and *step 4.* *Step 3.* executes the process presented in Figure 4.2 by selecting the best individuals to the parent population (this can be omitted at the first iteration), then creating the child population using only the rule activation ids (and not performing the transformations themselves). In addition, *step 4.* is executed only for the new individuals and *step 4.*2 is used only for correction if the trajectory is not executable (see Figure 6.3).

7.1.4 Parallelization of rule-based design space exploration

In order to support parallel exploration for rule-based design space exploration the following points have to be considered.

- **Responsibility of threads.** In order to explore the design space (as a directed graph) with multiple threads each thread should have an individual trajectory. This means, that every thread should own a different copy of the instance model to be able to fire activations from different model state at the same time. Their task is to 1) transform the model along a predefined strategy, 2) generate state and activation codes and 3) calculate any domain specific parameter such as objectives and global constraints.
- **Shared resources.** The definition of the design space exploration problem (objectives, constraints and rules) is shared data. The design space is also a shared resource as if each thread would manage its own design space, it would mean that the same trajectories could

be explored by different threads unintentionally and would also increase the memory foot print. Other shared resources include data for synchronization (strategy, stop condition) and a data structure for storing potential solutions.

- **Critical sections.** The inputs of the design space exploration don't change during exploration, therefore these can be read safely by multiple threads. Solutions can also be stored in a basic concurrent data structure. However, the state space of explored states is a directed graph, which is extended continuously and concurrently by adding new nodes and arcs. This is especially relevant when two (or more) threads try to add the same new node (or arc) to the design space, resulting in a race condition.

To demonstrate the approach, Figure 7.3 presents the process of a thread from the aspect of a multi-threaded depth first search (DFS) traversal strategy without optimization objectives. The dashed frames (and lighter colour) indicates, that the synchronization of the threads are necessary when using shared resources and rounded boxes are used where decisions are made. The main steps of a process are: 1) decide, whether a new thread can be started, 2) choose an unexplored activation and modify the model by executing the rule, 3) update the design space with the new state, 4) if the state is a solution, save the current trajectory, 5) backtrack if needed and 6) repeat from start. A thread backtracks, if a) all arcs are already explored from the actual state, b) it was already visited by an other thread, c) the global constraints are not satisfied, or d) the model state satisfies the goals. In the following, details of the synchronization tasks are discussed in the order of the flow of the process on Figure 7.3.

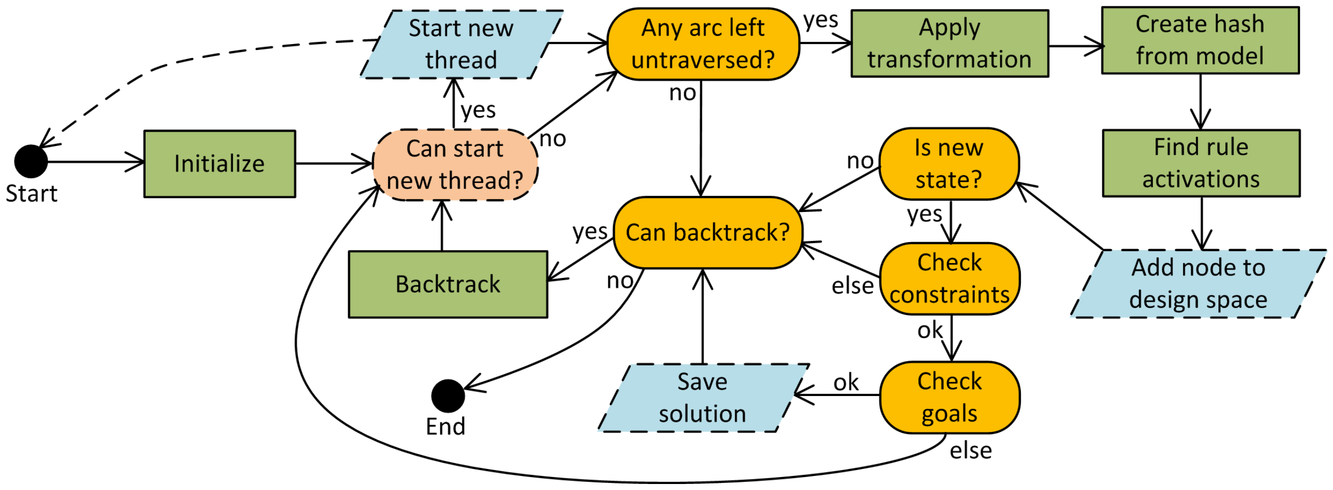


Figure 7.3: The process of DSE with multithreaded depth search exploration strategy

Starting a new thread In this DFS strategy example, a thread should only check if there are multiple executable arcs (so the threads can explore different rule activations) and the number of threads is lower than a threshold, which is usually limited by the CPU for performance optimization. Although this is synchronized easily, the logic can be more complex. For example, a thread can be stopped to start a new one from a more promising state. New thread can be started concurrently using simple locks, assuming it does not happen often.

State space update algorithm After a thread finished the exploration of a new model state, it has to synchronize with the shared state space representation. In this approach, arcs are added to the state space along with the new node, therefore the critical section is simply a synchronized *addNode* method, which also checks if the state is already traversed.

The *addNode* function (see Algorithm 1) uses a general optimistic lock-free pattern, which is mentioned at chapter 5: 1) if the node already exists with the current state code then it is connected with the traversed and already existing transition and return (line 2-5), 2) otherwise the new node is created locally, added to the state space and linked with the transition (line 6-11) and 3) if the node is already created by an other thread, that node is linked with the transition (line 13).

A blocking mechanism is used in case two threads take the same arc from the same node at the same time. This results in a race condition to create the new node and could cause inconsistency if the "winner" thread stops before setting the arc's reference to the node (line 11), while the other thread can go on and use that reference. To overcome this issue, the node has a *processed* flag, which will block other threads until it is set (lines 16-17). To entirely bypass this situation an arc is flagged when a thread goes in that direction. Therefore, a strategy that respects flagged arcs and chooses from unexplored arcs will not cause blocking, like our DFS implementation.

Algorithm 1 Non-blocking state update

```
Require: nodes              ▷ A concurrent hash map, where the key is the id of the model state
 1: function addNode(inArc,nodeId,outArcs)
 2:     node = nodes.get(nodeId)
 3:     if node != null then                                          ▷ Check if the node exists
 4:         inArc.setResultsIn(node)
 5:         return false
 6:     node = createNode(inArc,nodeId,outArcs)                          ▷ Create node locally
 7:     elderNode = nodes.putIfAbsent(nodeId,node)            ▷ Returns the node with nodeId
 8:     if elderNode == null then                         ▷ It was the first node put, with that id
 9:         inArc.setResultsIn(node)
10:         node.setProcessed()
11:         return true                       ▷ Return true indicates, that a new node was created
12:     else                                              ▷ An other thread created the node faster
13:         inArc.setResultsIn(elderNode)
14:         if elderNode.isProcessed() == false then
15:             waitForWinnerThread()                       ▷ Wait until lines 9-10 were executed
16:         return false
```

Saving solution trajectories Finally, if a model state satisfies the goals, the corresponding solution trajectory needs to be stored in a shared resource. Assuming that solution states are found relatively rarely, a simple lock on modifying the set of solutions or a basic concurrent data structure is sufficient.

7.1.5 Parallelization of the genetic algorithm

As presented in subsection 7.1.4, VIATRA-DSE gives support for parallelization of a traversal strategy. For that purpose, it copies the initial model for every new thread and initializes an EMF-IncQuery engine on it. Thus, threads can explore the design space independently and have to synchronize only for building the design space representation, allowing to recognize already traversed states by other threads.

Implementation of the NSGA-II algorithm exploits this feature by creating worker threads which check the feasibility of an individual and compute its fitness, while the master thread performs the rest of the work. Figure 7.4 shows an overview of the implemented process. The *master thread* is responsible for 1) *generating the initial population*, 2) *selecting* the population into fronts, 3) *generating the child population* using simply the state and activation IDs and 4) removing potential *duplications*. When new valid children are generated they are immediately sent to the *worker threads* for processing. The trajectory of the individual is applied to the initial model for (i) *checking its feasibility*, (ii) *correcting* if needed (i.e. omitting all the non-executable activations) and the objectives of the new individual are *calculated*. After the necessary amount of children are generated by the master thread and processed by the worker threads, the master thread starts a new iteration by executing its *selection* operator.

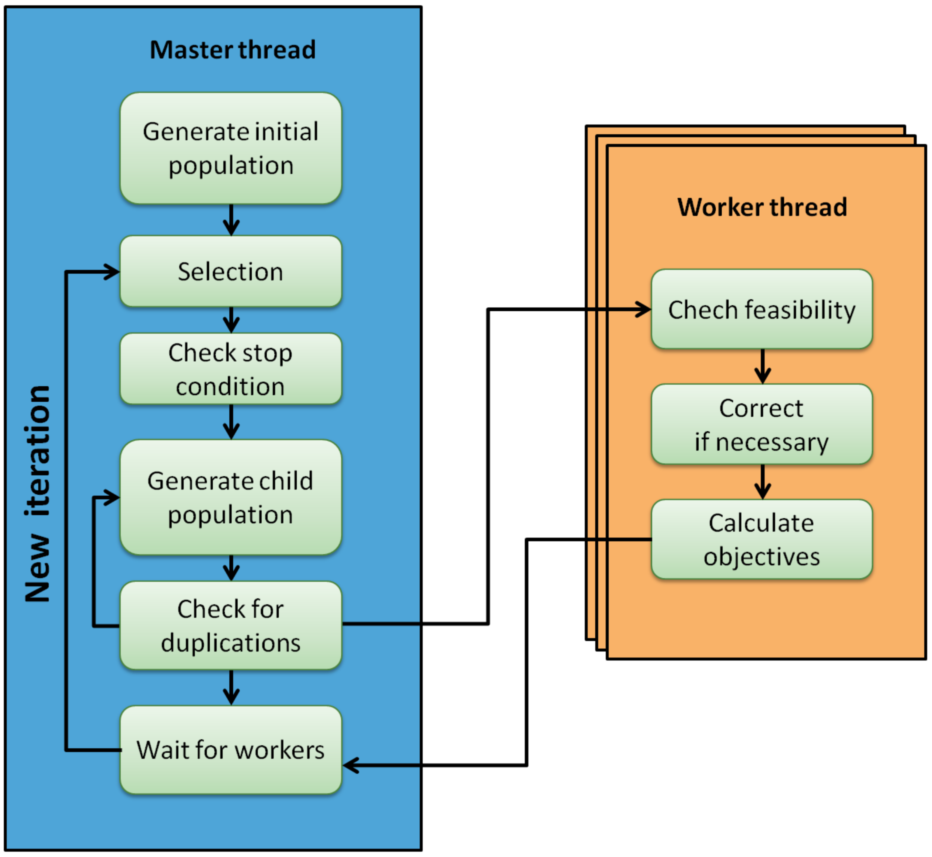


Figure 7.4: Workflow of the main and worker threads

Synchronization of the threads is solved by two queues and one binary flag.

1. Worker threads are created in the initialization phase and they start to listen to the same concurrent queue as soon as they are ready.
2. The master thread puts the newly created candidate children into this queue and the first available worker thread processes the candidate.
3. Workers put the validated children to an other concurrent queue which is monitored by the master thread.
4. If children were dropped for either being completely unfeasible or were corrected to an individual already present in the population, new children may be generated by the master thread to substitute the dropped indiviuals and fill up the population.
5. If the stop condition is fulfilled the worker threads are signaled with a specific flag to stop.

During the *generation of the initial population* and *selection* only one thread is working which may reduce the efficiency of parallelization. The generation of the initial population can be thought of as a local search and can be easily implemented to multiple threads as VIATRA-DSE supports that as one of its main features. Multithreaded selection (i.e. sorting into fronts) is also possible as comparing the individuals sequentially is unnecessary. Parallel computation of these tasks remains future work.

7.2 Implementation details

The conceptual contributions described in the previous section are implemented but not discussed here.

7.2.1 Configuration and logging

Genetic algorithms have many adjustable input parameters. Properties of the initial selection, size of the population, stop condition, chance of mutation and crossover operations, etc. can be configured to run the algorithm. These parameters may have a big impact on the results. Therefore it is crucial to be able to monitor and then analyze the internal behaviour of the various executions to achieve the required parametrization options.

For this reason, I created an executable file, one for each case study, which can run the design space exploration and is configurable via a coma separated values (CSV) configuration file. During the execution it creates and appends the results into several CSV files which includes information about every individual of every population. Figure 7.5 shows the overview of the testing process. Each row in the *config.csv* means a different configuration and run multiple times defined in the row. Runs for the same configuration result in the same output CSV file, while a new is created for each configuration. A new row is appended in a *results overview* file after every run containing information about the Pareto front, run time, memory usage, etc.

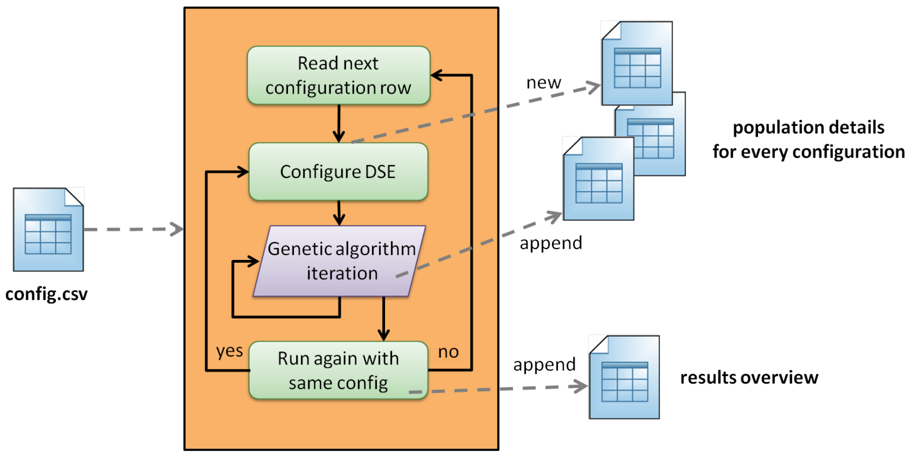

Read next configuration row
Configure DSE
Genetic algorithm iteration
Run again with same config
yes
no
new
append
append
config.csv
population details for every configuration
results overview

Figure 7.5: Architecture of the logging frameworks

7.2.2 API usage

The new algorithm on the VIATRA-DSE framework received a new API by wrapping the original one, keeping the domain independence and high level abstraction and extending it with configuration options for overriding its default behaviour.

```
 1 GeneticDesignSpaceExplorer gdse = new GeneticDesignSpaceExplorer();
 2 
 3 // rule−based dse
 4 gdse.setInitialModel(model);
 5 gdse.setSerializerFactory(new CPSSerializerFactory());
 6 gdse.addTransformationRule(startRule);
 7 gdse.addTransformationRule(allocateRule);
 8 gdse.addTransformationRule(createAppRule);
 9 gdse.addTransformationRule(createHostRule);
10 
11 // objectives
12 gdse.addSoftConstraint("RunningApps", runningAppQuerySpecification, −4);
13 gdse.addSoftConstraint("AllocatedApps", allocatedAppQuerySpecification, −3);
14 gdse.addSoftConstraint("RequirementHasApp", applicationsQuerySpecification, −2);
15 gdse.addSoftConstraint("UnusedHosts",unusedHostQuerySpecification, 1);
16 gdse.setModelObjectiveCalculator(new ObjectivesCalculator());
17 gdse.addObjectiveComparator("ResourceUsage", comparator);
18 
19 // genetic operators
20 gdse.setInitialPopulationSelector(new BFSSelector(0.4));
21 gdse.addMutatitor(new AddRandomTransitionMutation(), 2);
22 gdse.addMutatitor(new ModifyRandomTransitionMutation(), 1);
23 gdse.addCrossover(new CutAndSpliceCrossover(), 1);
24 gdse.addCrossover(new OnePointCrossover(), 1);
25 gdse.setMutationChanceAtCrossover(mutationChance);
26 gdse.setSelector(new NonDominatedAndCrowdingDistanceSelector());
27 
28 // other options
29 gdse.setSizeOfPopulation(15);
30 gdse.setStopCondition(StopCondition.ITERATIONS, 50);
31 
32 gdse.startExploration(timeout);
```

Code 7.1: API usage

Code 7.1 shows a possible configuration of the smart house cyber-physical system case study. *GeneticDesignSpaceExplorer* is used for configuring a problem. Method calls at *lines 4-9* show the configuration options VIATRA-DSE initially has: setting the initial model, the state and activation encoder and the transformation rules (see Figure 2.7).

Lines 12-17 show how to define the objectives and soft constraints (see Figure 2.6). Soft constraints consist of a name, a model query specification (derived from an EMF-IncQuerypattern) and its weight. Objectives derived from the model are calculated via an interface implementation, while whether it is for maximization or minimization is defined by a comparator. Objectives derived from the trajectory are defined on the rules.

Lines 20-26 configure the genetic operators. *Line 20* sets the initial selection rate which is a breadth first search strategy with 0.4 chance to choose a trajectory to the initial population. Then the mutation and crossover operations are defined with a relative usage rate (i.e. *AddRandomTransitionMutation* is used twice as many times as *ModifyRandomTransitionMutation*; same for the crossover operations). At what rate to use mutation or crossover operations is set in a separate method call. The selection operation of the NSGA-II algorithm is set.

Before starting the exploration at *line 32* the size of the population and the stop condition is configured. Exploration can be stopped after a pre-defined number of iterations, after a good enough solution is reached or after the exploration cannot reach a better solution after a number of iterations.

Chapter 8

Evaluation

As there are no widely established benchmarks available for evaluating rule-based DSE approaches, we carried out experimental evaluation in the context of our case study. Section 8.1 carries out evaluation on the cyber-physical system case study, which is a collaborative work, and section 8.2 discusses my own analysis on the BPMN case study.

8.1 Experimental analysis of the cyber-physical system case study

We compare our multi-objective optimization (NSGA) approach with (1) random simulation (Random) and (2) a fixed priority local search (FPLS) strategy used as a basis of comparison in [5]. As a consequence, the DSE problem is identical in both cases, furthermore, the evaluation of graph patterns and execution of graph transformation rules is carried out by the same implementation. This way, any difference between the measurement results is expected to be affiliated to the substantially different search strategies. Our measurements aim to address which DSE approach finds better candidates with respect to different objectives. For this purpose, we test the following hypothesis using two-tailed Wilcoxon tests:

- **Hypothesis** H_0 There is no significant evidence that NSGA outperforms FPLS and/or Random.
- **Hypothesis** H_1 There is a statistical evidence that NSGA outperforms other DSE approaches with respect to different objectives.

8.1.1 Scenario

In our experimental scenario, we generate requests for an increasing number of rooms (4, 6, 8, 12) with an equal use of all packages (and respective model sizes of 130, 200, 230 and 330 graph elements). The initial model only contains the requests with requirements and the application and host types but no host instances are available. Therefore, it is the role of the DSE process to 1) create a sufficient number of application and host instances, 2) allocate application instances to host instances, and then 3) start and stop the application instances by applying the appropriate exploration rules. In the most complex case, the different exploration techniques had to synthesize a rule sequence consisting of over 200 steps.

In a preparatory phase, we experimented with different configurations of our multi-objective DSE approach, and we decided on the most promising configuration parameters, such as population size of 15 individuals, iterations between 400 and 1200 steps, crossover by permutation, and high rate of mutations. Our measurements were run on 8-core desktop computers with 32 GB of RAM running on Linux operating system. The used Java version was 1.7.0_55 and the heap size was 24GB.

Then for the experiments, we set up a timeout of 2 minutes for test cases (except for the largest example where it was 5 minutes) and run 30 experiments on the different problem sizes. One experiment was constructed as follows:

- **NSGA**: We selected population size *pop* and iteration number *it* for a problem size, and run our algorithms. At the end of each run, we selected only one solution from the Pareto front produced by NSGA. We selected the solution which has the best constraints' fulfillment value. If several solutions have the best constraints' fulfillment value, then we select the solution which is characterized by the minimal cost and/or the maximum usage of computer servers.
- **Random**: We executed $pop \times it$ random simulation runs and recorded the best result.
- **Fixed priority LS (FPLS)**: We set up priorities in a way to guarantee that all application instances will eventually be allocated to host instances. Our priorities guaranteed that the first three soft constraints will definitely be guaranteed, but we may generate more host instances than necessary.

8.1.2 Analysis of results

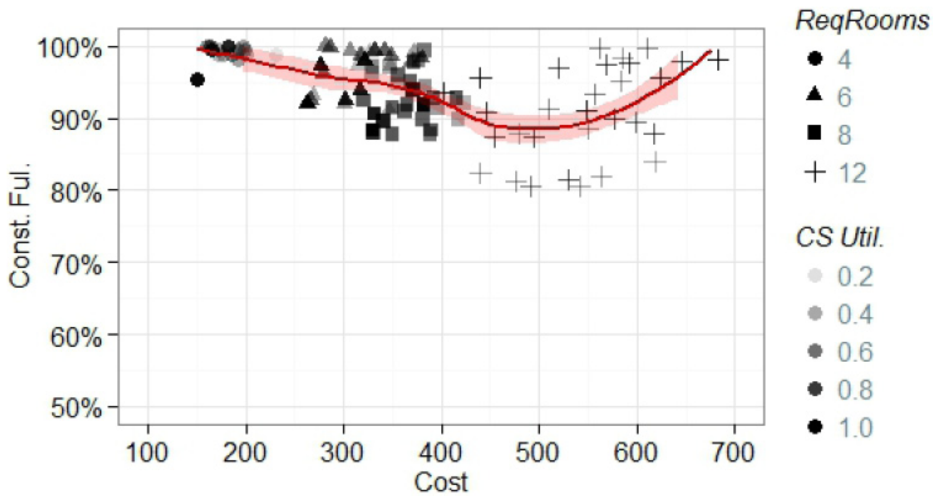


Figure 8.1: Quality of NSGA solutions produced in 30 runs for different problem sizes, as measured by the measurements normalized constraints' fulfillment, cost and computer server utilization

Before comparing the results of our approach NSGA with those of FPLS and Random, we analyse the quality of NSGA solutions. Figure 8.1 shows the distribution of NSGA produced solutions in 30 runs for considered problem sizes (4, 6, 8 and 12 requested rooms). The figure shows that in all considered scenarios, NSGA produced solutions have overall good quality. Considering all the produced solutions ($30 runs \times 4 scenarios = 120 solutions$), the minimum fulfillment of constraints is above 80%, and for the major body of produced solutions, the fulfillment of constraints is above 90%. Analyzing the evolution of the mean value of constraints' fulfillment through different problem sizes, we find that it takes its minimum value, around 90%, in the problem size 12 where the incorporated cost in the sequences of rule executions is relatively small, as compared to the cost of other solutions in the problem size 12. Indeed, this relative low fulfillment of constraints is mainly due to the following fact: the optimization process was stopped before reaching sequences of rule executions that have enough depth to satisfy all the requirements associated to this problem size. Increasing the number of iterations of the optimization process in this problem size, NSGA was able to find better solutions in terms of constraints' fulfillment, as demonstrated by solutions which have high cost. As for smaller problem sizes, such as in the scenarios of 4 and 6 requested rooms, in almost all runs NSGA was able to find fully valid solutions that satisfy all the constraints.

	Δ(NSGA - FPLS)			**Δ(NSGA - Random)**		
Pb size	Const. Ful	Solution Cost	CS Util.	Const. Ful	Solution Cost	CS Util.
4	**+20****	**-369****	**+0.32****	**+44****	**-145****	**+0.19****
6	**+27****	**-559****	**+0.37****	**+91****	**-118****	+0.06
8	**+20****	**-746****	**+0.46****	**+92****	**-239****	**+0.22****
12	+6	**-1058****	**+0.51****	**+55****	-8	**+0.18***

Table 8.1: Comparing the results of our approach NSGA and FPLS, and between the results of NSGA and Random, with different problem sizes with regard to the number of requested rooms: two-tailed Wilcoxon tests with $\alpha = 0.05$ and adjusted $p.value$ using the Benjamani and Hochberg (BH) correction for multiple tests.

To confirm our claim that NSGA produces good results, we compare NSGA's solutions to those produced by FPLS and Random. Table 8.1 shows clearly that NSGA outperforms both FPLS and Random, with statistical evidence, in almost all cases. Indeed, only in the problem size 12, there is no significant statistical evidence that NSGA outperforms FPLS with regard to constraints' fulfillment. However, in this case, NSGA significantly outperforms FPLS in reducing the cost of solutions and increasing the usage of computer server resources. Hence, NSGA overall significantly outperforms FPLS, and the same finding apply for the Random approach. As a consequence, we reject the null hypothesis H_0 and accept H_1.

8.2 Experimental analysis of the BPMN case study

8.2.1 Scenario

In this scenario, the BPMN process of Figure 2.3 is optimized using the transformation rules of Appendix A. The role of the exploration process is to 1) assign a *Resource Type Variant* to each *Task* (reassignment is not possible by the rules), 2) create *Resource Instances*, 3) modify the process with either transforming a sequential execution of tasks to a parallel execution or vice versa and 4) for each solution simulate the business process with a predefined number of tokens (100) and token rate (20 ms). The objectives are the following:

1. Two soft constraints depicted in Appendix A, with weights of 1 and 10. When such a constraint matches the business process cannot be simulated.
2. Minimize average response time based on the tokens spent in the business process.
3. Maximize minimum resource usage based on the individual utilization *Resource Instances.*
4. Minimize rule execution costs: costs of *rule createResource* is dependent on the *resource type variant* and is shown in Figure 2.4, *rule allocateTaskToVariant* has no cost and the other two rules have a cost of one.

Average response time is considered infinite and minimum resource usage becomes zero when a business process cannot be simulated.

The analysis is executed with the following configurations (after several experimental tests):

- 50 iterations,
- population size of 13 individuals,
- breadth first search with 0.3 selection rate as an initial selector,
- crossover by cut and splice,
- two types of mutations:
  - *add new random rule execution*
  - and a slightly modified version of it: *add new random rule execution by priority* which chooses randomly from the available activations with the highest priority (*rule allocate-TaskToVariant*: 3; *rule createResource*: 2; other two has a priority one)

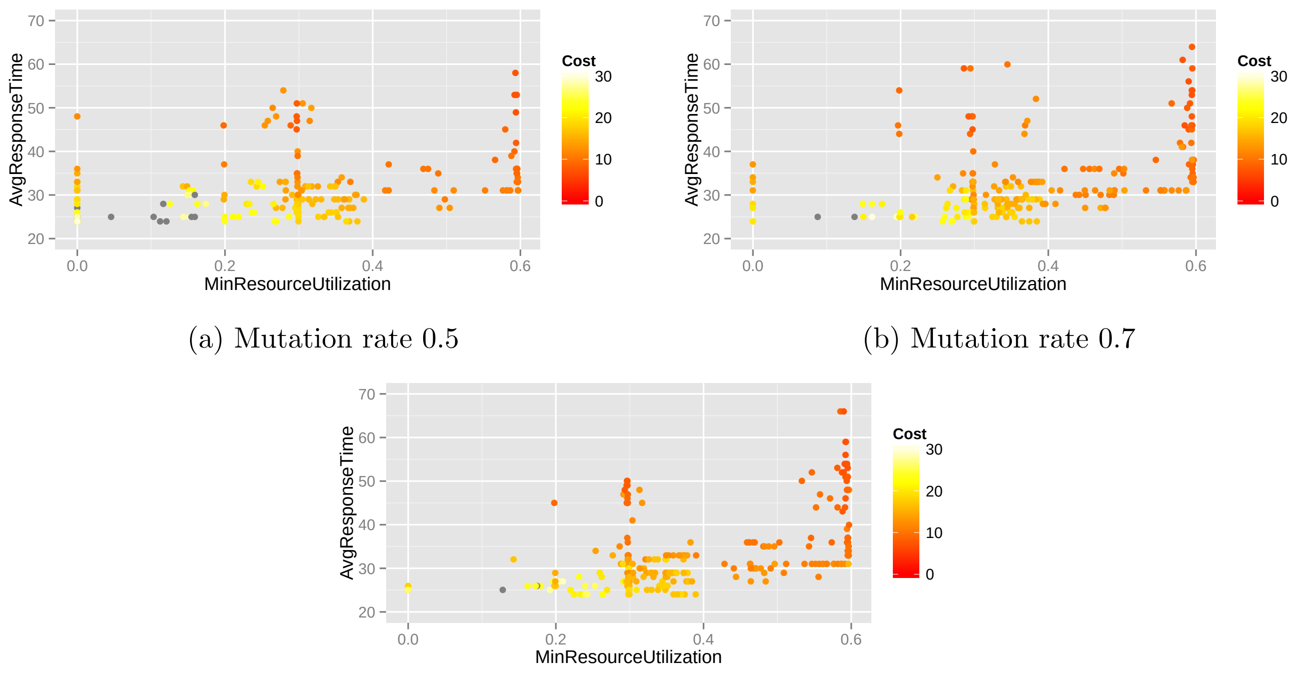
The aim of the measurements is to investigate the impact of the mutation rate (chance the choose a mutation operator instead of a crossover) to the Pareto front. The measurements were carried out on a desktop computer with 4 logical cores and 8 GB of RAM running Windows 7 operating system and Java 1.7.

8.2.2 Analysis of results

The scatter plots in Figure 8.2 show the objective values of Pareto fronts from 30 separate runs with different mutation rates. Grey points depict out of range cost values.

General observations:

- It is clear in the results of all three run configurations that the lowest response time (24-25 ms) can only be achieved with higher costs (>16) and lower resource utilization (<0.4).
- Cost and resource utilization have high correlation: higher resource utilization has lower costs as less resource instances are created during the exploration.



(a) Mutation rate 0.5

(b) Mutation rate 0.7

(c) Mutation rate 0.9

Figure 8.2: Values of objectives in the Pareto front, plotting 30 runs

- It is interesting that the system cannot be configured with minimum resource utilization higher than 60%. It probably means that only a relatively small number of tokens reach one of the tasks and it is waiting most of the time leaving the resources unused, i.e. the rate of the incoming tokens is lower than the execution time of the task. There are three possible reasons for this: 1) the rate of tokens (requests) is inherently low, 2) the decision gateway directs most of the tokens in the other direction and 3) there is a bottleneck before this task.
- The same spike can be seen at minimum resource utilization 0.3. A possible explanation would be the same as the previous, but the particular task has twice as many resources.

Higher mutation rate increases the number of solutions with high resource utilization and low costs. The reason behind this is that cut and splice crossover can add two long trajectories (see Figure 6.4) resulting in many resource creation rules, thus reducing resource utilization and raising costs. Also two trajectories may have different resource type variants allocation and the child solution may have unused resource instances as they belong to unused variants. However, these trajectories are likely to remain in the Pareto front as they have high response time. On the other hand, mutation operators only add one activation to the trajectories which will be probably dropped if an unusable resource instance was created as minimum resource utilization 0 can be easily dominated by other solutions. Therefore, by raising mutation rate (and decreasing the chance of crossover) these trajectories are less likely to appear and better results can be achieved.

8.3 Speedup factor of the genetic algorithm

I also measured the speedup factor of the paralellization of the genetic algorithm that I introduced in subsection 7.1.5. The nature of the concept gives two things to point out: 1) at least two threads have to run and 2) the work of the master thread is sequential. Therefore, I measured only the average run time of the worker threads in one iteration, i.e. $T_n = average(\{T_i^{iteration}\})$, where i is the index of the iterations and n is the number of threads.

As the length of the solutions and population size can affect the speedup factor, I used three different case studies (two CPS problem - with problem size 4 and 6 - and one BPMN problem presented in section 2.1.2) with different population sizes (10, 15, 20). The tests were run 30 times until 50 iterations, with 1, 2, 3 and 4 worker threads on a desktop computer with 4 logical cores (2 core + HT) and 8 GB of RAM running Windows 7 operating system and Java 1.8.

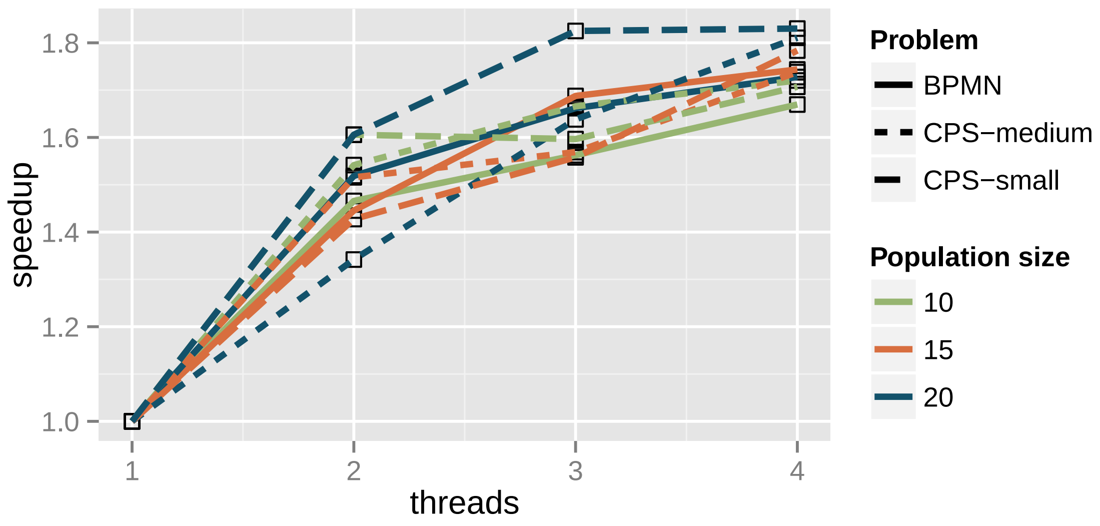


Figure 8.3: Speedup factor of different test cases

Figure 8.3 presents the speedup factor of the parallel genetic algorithm, with different number of worker threads. The line types denotes the case study, while different colours denotes different population size (10 - light blue, 15 - orange, 20 - dark blue). I made the following observations:

- First of all, the measured run times are very stochastic and it is possible that 30 runs was insufficient for the measurement. The results for two and three threads are somewhat differ from my assumptions, but for four threads it is almost identical.
- The speedup factor is far from the theoretical maximum n (the number of threads), but the following circumstances degrade the results: 1) the master thread is also running in the background and 2) hyper threading allows only a limited speed up. Therefore, even the results with two threads, which are relatively the best in approaching the theoretical maximum, could be better with a CPU with more cores.
- The speedup factor is only slightly dependent on the population size, which is clearly seen at four threads. The bigger the population size is the more work the worker threads have to do preventing idle time, thus they can be faster compared to the runs with one thread.
- The size of the problem (an thus the size of the solutions) also affects the speedup factor similarly to the previous point. The BPMN problem has the shortest solutions (9-11 long) and overall it has the smallest speedup factor. The size of the solutions for the two CPS problem failed to stabilize in 50 iterations and both reached the length between 40 and 50, thus they have similar speedup factor.

To sum up, the paralellization of the genetic algorithm has improved the performance and it seems to be more effective with bigger problems and population sizes, however more tests should be carried out with better CPU to confirm these results.

Chapter 9

Summary

In this report, we proposed to integrate constrained multi-objective optimization as a search strategy for rule-based design space exploration frameworks. In contrast to existing genetic approaches for design space exploration, in our approach, a genetic population consists of rule execution sequences from an initial model to design candidate, constraints are captured by model queries, objectives are calculated from models or rule sequences, while crossover and mutation operations are manipulating rule sequences.

A first key challenge in this setup is that traditional encoding of populations as fixed width bit vectors is unable to represent rule sequences of increasing depth, while it is very difficult or impossible to give a priori upper bounds for feasibility checks. Moreover, unlike in most application scenarios of genetic algorithms, crossover and mutation operations may derive non-executable application sequences as individuals which must be omitted from the population. As a consequence, we had to integrate multi-objective optimization techniques to a model-driven rule-based DSE framework as a mapping from a rule-based DSE to a genetic algorithm proved to be infeasible.

Our initial experiments demonstrated that multi-objective optimization is an effective strategy for solving rule-based design exploration problems. However, using a randomly synthesized initial population appears to be suboptimal choice in our context and further research is subject to future work.

This line of research has been carried out in close collaboration with researchers from BME-MIT and DIRO at Université de Montréal reported in [14]. The core theoretical novelty of adapting the global search technique of NSGA-II for multi-objective optimization techniques for rule-based design space exploration is a joint contribution. This report summarizes the entire line of research (where I continuously and actively participated in) and below I highlight my own contributions within this line.

- **Conceptual contributions**
  - I proposed to integrate global search techniques as local search based rule-base design space exploration by switching between trajectories and generating the initial population with an arbitrary local search technique.
  - I proposed to replace the built-in domain independent state coder of VIATRA-DSE with a domain specific one allowing activations to be encoded to the same ID.
  - I developed a concept of a parallel genetic exploration strategy for multi-objective rule-based design space exploration by executing the model transformation, fitness calculation and correction of the trajectories in worker threads.

- **Practical contributions**
  - I proposed an architecture and implementation with an easy to use API for multi-objective rule-based design space exploration built on top of VIATRA-DSE, which enables the overriding of all the genetic operators and parallel execution.
  - I developed an evaluation framework especially for this new exploration strategy which enables extensive and customizable monitoring of the execution internal behaviour and wrapped them with executable files which allows easy testing on several computers.
  - I prepared two case studies with domain specific state encoders for experimental evaluation.
  - I evaluated the effectiveness of the proposed techniques on two case studies (where the BPMN evaluation is my own contribution).

Future work In the future I would like to extend the approach with adaptiveness of the parameters, experiment with different genetic operators (like a guided local search as an initial population selector and guided crossover operations) and support hierarchical (prioritized) objectives with different selection operators. Furthermore, I would like to evaluate the feasibility and possibly implement other metaheuristic techniques like the ant colony optimization.

Bibliography

[1] Twan Basten, Emiel van Benthum, et al. Model-driven design-space exploration for embedded systems: The Octopus toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *LNCS*, pages 90–105. Springer, 2010.

[2] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, volume 2855 of *LNCS*, pages 290–305. Springer, 2003.

[3] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(4):39, 2011.

[4] Joachim Denil, Maris Jukšs, Clark Verbrugge, and Hans Vangheluwe. Search-based model optimization using model transformations. Technical report, McGill University, Canada, 2014.

[5] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. A model-driven framework for guided design space exploration. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Lawrence, Kansas, USA, 11/2011 2011. IEEE Computer Society, IEEE Computer Society.

[6] I. Galvao Lourenco da Silva, E. Zambon, A. Rensink, L. Wevers, and M. Akşit. Knowledge-based graph exploration analysis. In *Fourth International Symposium on Applications of Graph Transformation with Industrial Relevance, AGTIVE 2011, Budapest, Hungary*, LNCS 7233, pages 105–120. Springer, October 2011.

[7] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick fix generation for DSMLs. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011*. IEEE Computer Society, 09/2011 2011.

[8] Ákos Horváth and Dániel Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 11:385–408, 2012 2012.

[9] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[10] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and OmarBen Omar. Search-based model transformation by example. *Software and Systems Modeling*, 11(2):209–226, 2012.

[11] Marwa Shousha, Lionel C. Briand, and Yvan Labiche. A uml/spt model analysis methodology for concurrent systems based on genetic algorithms. In *MoDELS*, pages 475–489, 2008.

[12] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions Evolutionary Computation*, 6(2):182–197, 2002.

[13] András Szabolcs Nagy and Miklós Földényi. Highly efficient design space exploration with model-driven techniques. `https://tdk.bme.hu/VIK/ViewPaper/Nagyhatekonysagu-tervezesi-ter-bejaras`, 2013. TDK.

[14] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 289–300, New York, NY, USA, 2014. ACM.

[15] András Szabolcs Nagy. Multi-objective optimization in rule-based design space exploration. `https://tdk.bme.hu/VIK/DownloadPaper/Tobbcelu-optimalizacios-modszerek-szabaly`, 2014. TDK.

[16] Cyber-physical systems. `http://cyberphysicalsystems.org/`, October 2014.

[17] Business Process Model and Notation. `http://www.bpmn.org/`, October 2014.

[18] Object Management Group. `http://www.omg.org/`, October 2014.

[19] Unified Modeling Language. `http://www.uml.org/`, October 2014.

[20] Visual Paradigm for UML. `http://www.visual-paradigm.com/`, October 2014.

[21] Papyrus. `https://www.eclipse.org/papyrus/`, October 2014.

[22] Eclipse Modeling Framework. `https://www.eclipse.org/modeling/emf/`, October 2014.

[23] Resource Description Framework. `http://www.w3.org/RDF/`, October 2014.

[24] Web Ontology Language. `http://www.w3.org/2001/sw/wiki/OWL`, October 2014.

[25] Protégé. `http://protege.stanford.edu/`, October 2014.

[26] Object Constraint Language. `http://www.omg.org/spec/OCL/`, October 2014.

[27] SPARQL Protocol and RDF Query Language. `http://sparql.org/`, October 2014.

[28] The World Wide Web Consortium (W3C). `http://www.w3.org/`, October 2014.

[29] EMF query. `https://www.eclipse.org/incquery/`, October 2014.

[30] EMF-IncQuery. `https://www.eclipse.org/incquery/`, October 2014.

[31] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, pages 1–38, 08/2014 2014.

[32] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[33] Arend Rensink. Isomorphism checking in GROOVE. *Electronic Communications of the EASST*, 1, 2007.

[34] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, San Francisco, CA, USA, October 20-24 2003. Springer.

[35] Stefan Edelkamp, Shahid Jabbar, and Alberto Lluch-Lafuente. Heuristic search for the analysis of graph transition systems. In *Proceedings of the Third international conference on Graph Transformations*, volume 4178 of *LNCS*, pages 414–429. Springer-Verlag, 2006.

[36] Luciano Baresi, Vahid Rafe, Adel T. Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *ENTCS*, 213, 2008.

[37] Osmar Marchi dos Santos, Fernando Luís Dotti, and Leila Ribeiro. Verifying object-based graph grammars. *ENTCS*, 109:125–136, 2004.

[38] Luciano Baresi and Paola Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *Graph Transformations*, volume 4178 of *LNCS*, pages 306–320. 2006.

[39] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2004. 10.1007/978-3-540-25959-6_40.

[40] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. ICGT 2002*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.

[41] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS*, pages 197–211, 2006.

[42] Tripti Saxena and Gabor Karsai. MDE-based approach for generalizing design space exploration. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 46–60. Springer, 2010.

[43] Twan Basten, Martijn Hendriks, Nikola Trčka, Lou Somers, Marc Geilen, Yang Yang, Georgeta Igna, Sebastian de Smet, Marc Voorhoeve, Wil van der Aalst, et al. Model-driven design-space exploration for software-intensive embedded systems. In *Model-Based Design of Adaptive Embedded Systems*, pages 189–244. Springer, 2013.

[44] Eunsuk Kang, Ethan K. Jackson, and Wolfram Schulte. An approach for effective design space exploration. In *Monterey Workshop*, pages 33–54, 2010.

[45] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Reliability-driven deployment optimization for embedded systems. *Journal of Systems and Software*, 84(5):835–846, 2011.

[46] Indika Meedeniya, Aldeida Aleti, and Lars Grunske. Architecture-driven reliability optimization with uncertain model parameters. *Journal of Systems and Software*, 85(10):2340 – 2355, 2012.

[47] B. Schatz, F. Holzl, and T. Lundkvist. Design-space exploration through constraint-based model-transformation. In *Engineering of Computer Based Systems (ECBS)*, pages 173 –182, March 2010.

[48] Christian Blum and Xiaodong Li. Swarm intelligence in optimization, 2008.

[49] Marco Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

[50] J. Kennedy and R. Eberhart. Particle swarm optimization. *Neural Networks, 1995. Proceedings., IEEE International Conference on (Volume:4)*, 1995.

[51] DT Pham and Michele Castellani. The bees algorithm: Modelling foraging behaviour to solve continuous optimization problems. *Proceedings of the Institution of Mechanical Engineers*, 2009.

[52] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Adv. Eng. Softw.*, 69:46–61, March 2014.

[53] Yuanyuan Zhang. *Multi-Objective Search-based Requirements Selection and Optimisation.* Phd. thesis, King's College London, UK, 2010.

[54] H. Jain and K. Deb. An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *Evolutionary Computation, IEEE Transactions on*, 2013.

[55] Hajer Saada, Marianne Huchard, Clémentine Nebut, and Houari A. Sahraoui. Recovering model transformation traces using multi-objective optimization. In *ASE*, pages 688–693, 2013.

[56] Ramin Etemaadi and Michel RV Chaudron. A model-based tool for automated quality-driven design of system architectures. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications (ECMFA'12)*, pages 2–5, 2012.

[57] Marouane Kessentini, Philip Langer, and Manuel Wimmer. Searching models, modeling search: On the synergies of sbse and mde. In *CMSBSE@ICSE*, pages 51–54, 2013.

[58] Horia Calborean, Ralf Jahr, Theo Ungerer, and Lucian Vintan. A comparison of multi-objective algorithms for the automatic design space exploration of a superscalar system. In Loan Dumitrache, editor, *Advances in Intelligent Control Systems and Computer Science*, volume 187 of *Advances in Intelligent Systems and Computing*, pages 489–502. Springer Berlin Heidelberg, 2013.

[59] Ralf Jahr, Horia Calborean, Lucian Vintan, and Theo Ungerer. Boosting design space explorations with existing or automatically learned knowledge. In JensB. Schmitt, editor, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 7201 of *Lecture Notes in Computer Science*, pages 221–235. Springer Berlin Heidelberg, 2012.

[60] Cristiana Bolchini, Pier Luca Lanzi, and Antonio Miele. A multi-objective genetic algorithm framework for design space exploration of reliable fpga-based systems. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[61] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[62] Ákos Dudás and Sándor Juhász. Blocking and non-blocking concurrent hash tables in multi-core systems. *WSEAS Transactions on Computers*, 2013.

[63] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. *Principles of Distributed Computing*, 1994.

[64] Maged M. Michael and Michael L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 1997.

[65] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. Towards guided trajectory exploration of graph transformation systems. *Electronic Communications of the EASST, Petri Nets and Graph Transformations 2010*, 40, 08/2011 2011.

Appendix A

BPMN case study

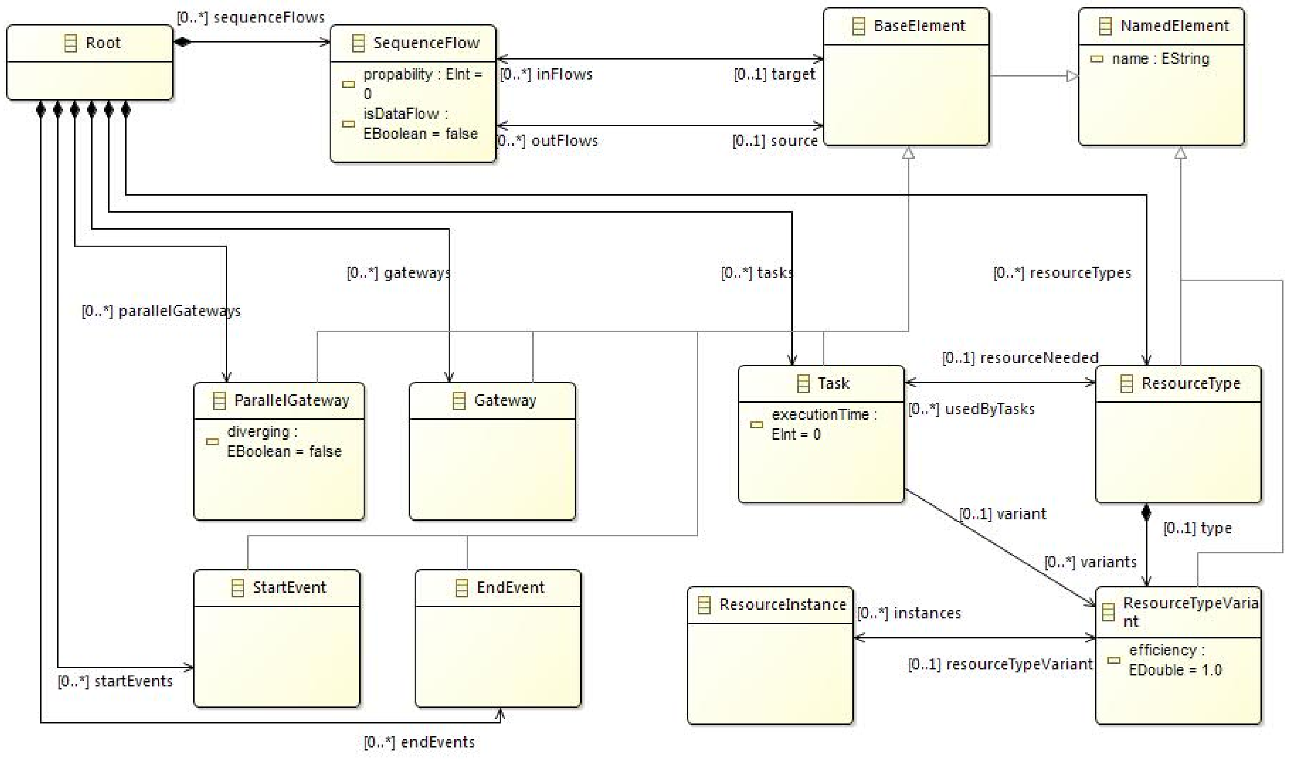


Figure A.1: Metamodel

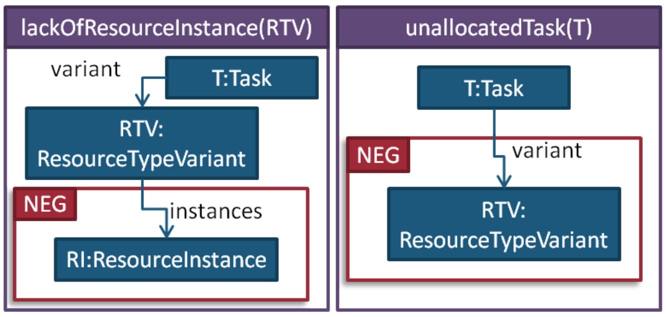


Figure A.2: Constraints

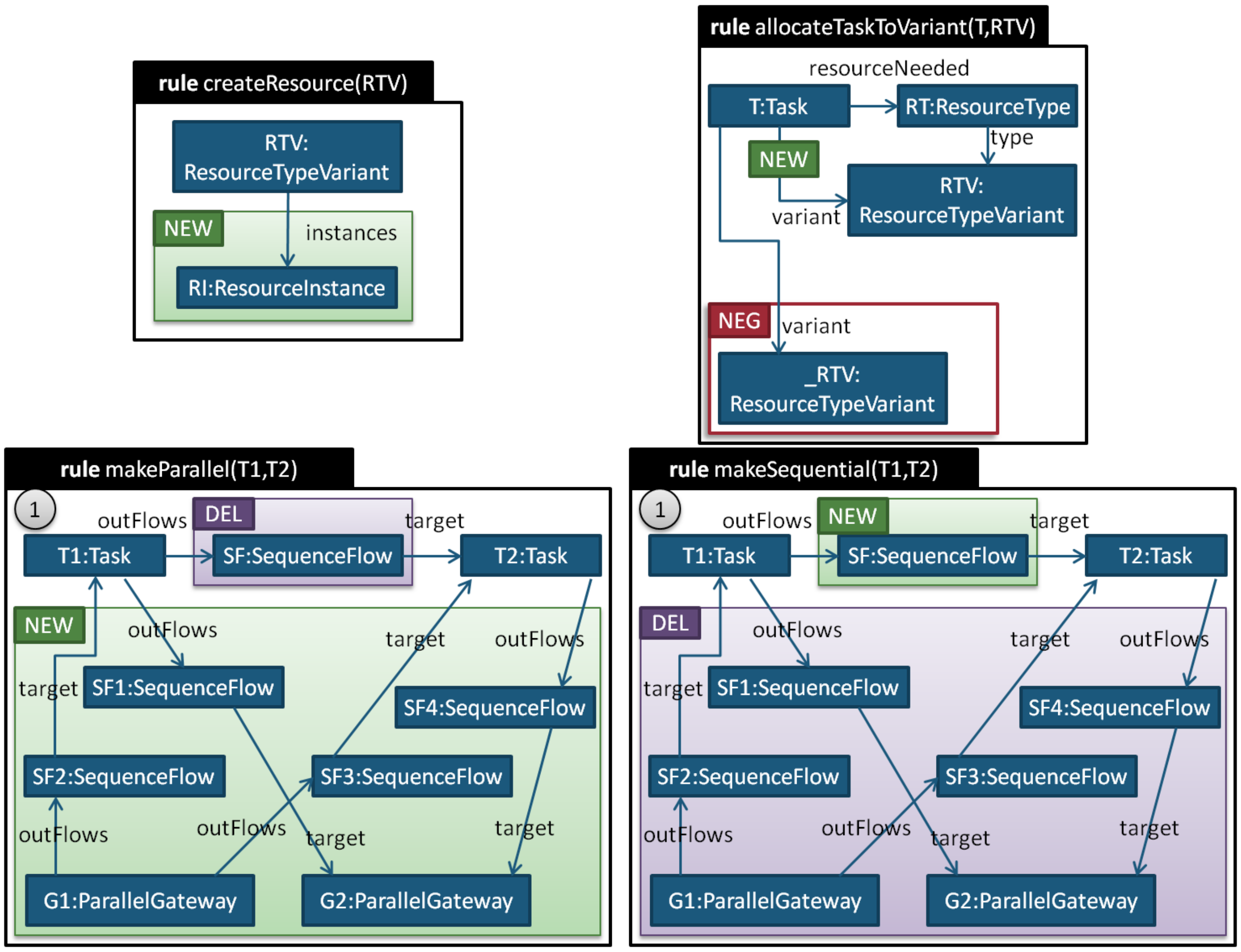


Figure A.3: Transformation Rules