CGL: A Domain Specific Language For Constraint Generation

Marwa A. Elmenyawi, Mostafa E. A. Ibrahim
Benha Faculty of Engineering
Benha University
Benha, Egypt
Email: marwa.elmenyawi@bhit.bu.edu.eg
mustafa.ibrahim@bhit.bu.edu.eg

Cherif Salama, I. M. Hafez
Faculty of Engineering
Ain Shams University
Cairo, Egypt
Email: cherif.salama@eng.asu.edu.eg
ismail_hafez@eng.asu.edu.eg

***Abstract*—Integer linear programming solvers are used to solve a wide variety of problems emerging in diverse domains. However, automatically generating the integer linear equations that are used as input for the solvers remains a challenging task. This paper proposes a domain specific language called CGL that can be used to describe the equations in a concise manner. In addition to the proposed language implementation, the syntax and semantics of CGL are formally given. The paper demonstrates the usefulness of CGL using a motivating example.**

I. Introduction

Integer linear programming (ILP) solvers have evolved significantly and are considered extremely useful helper tools. ILP solvers are used to solve a wide variety of problems emerging in diverse domains. Examples include MINOS, CPLEX, etc. Nevertheless, generating the integer linear equations that are used as input for the ILP solvers remains a challenging task. Of course, it is always possible to generate the equations manually for relatively-small problem instances; however, this approach is impractical, non-scalable, and error-prone. The goal of this paper is to present a language that can be used to describe the equations in a concise manner. The language is simple enough to be used by beginners (without too much training) and yet powerful enough to generate all the needed constraints by a wide variety of applications. The language can also be easily extended to support more constraints whenever needed.

The proposed language, named Constraints Generation Language (CGL) is introduced in the paper as a Domain Specific Language (DSL). As such, its syntax and semantics are presented in details. As a generation language, the main usage of CGL is to be elaborated into standard ILP equations that can be accepted by existing ILP solvers. One important implication of CGL's conciseness and simplicity is that it can be integrated with other tools such that these tools can automatically generate CGL statements that will later be expanded into ILP statements.

A declarative programming approach, such as Constraint Programming [1] is considered to be a suitable way to describe the constraints in a natural, declarative and expressive way. A declarative, very expressive, DSL [2] , [3] is created to help the description of this system. DSLs are lightweight programming languages used to represent domain specific knowledge using some sort of syntax. DSLs let you concisely express the concepts of a particular domain. DSLs have a gain in terms of expressiveness and ease of use compared to general purpose languages for the specific domain as the syntax and semantic of these DSLs are designed explicitly to describe only the knowledge of a specific domain. Conversely they are usually less expressive than general-purpose programming languages out of their domain.

The rest of this paper is organized as follows: Section II introduces some basic concepts in order to help understanding the rest of the paper. Section III demonstrates the phases of designing the proposed language. Section IV illustrates the results of implementing the proposed DSL. Finally, Section V concludes this paper and suggests some future work.

II. Background and Related Work

In this section we provide the necessary background and informally introduce some important concepts. Namely we discuss DSLs, constraints modeling tools, and ILP solvers.

A. DSLs

DSLs are attracting a lot of attention in software engineering as well as in programming languages. Banking [4], robotics [5], [6], and telephony [7] are examples of the different application on which DSLs are used. Yet, DSLs do not have the same meaning and objectives depending on the community considered.

Deursen et al. [2] defined DSL as "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". Markus Vlter et al. [8] noted that "natural/suitable for the stake holders who specify that particular concern" are the specification that the abstractions and notations must follow. DSLs are usually small, and allow fast development of complete programs, which solve problems in the domain. As such, DSLs can be viewed as sets of general, all-encompassing solutions for problem domains. A DSL has the following characteristics [9] : 1) The domain is well-defined and central, 2) the notation is clear, 3) the informal and formal meaning are clear and can be implemented, and 4) finally the language size is small.

DSL development generally involves the following phases [9] : Decision, Analysis, Design, and Implementation/Deployment. The decision phase is one in which the reasons for DSL development are weighed, with consideration of long-term goals along with economic and maintenance factors. The problem domain is defined at the domain analysis phase. The

978-1-4673-9971-5/15/$31.00 ©2015 IEEE

approaches of DSL design can be characterized by using two orthogonal dimensions. The first dimension is the relationship between the DSL and existing languages, while the second dimension is the formal nature of the design description. Following the design phase is the implementation/deployment phase, in which the most suitable implementation approach is chosen.

B. Constraints Modeling Tools

The modeling tools are divided into four categories as explained below [10].

Constraint Languages

Such languages are general programming languages and support a variety of solving techniques. There are different examples such as ECLiPSe [11] which is classified as logical language or object oriented languages like Comet [12]. The flexibility in modeling is the main advantage of such languages. A rich set of constraints can be provided allowing users to define their own constraints. The main drawback of such languages is that users must have enough programming skills as high level modeling is not supported by these languages.

Constraint Toolkits

Constraint toolkits are libraries in which constraint modeling and solving facilities for programming languages are provided. Examples of such toolkits are: CPLEX [13] and Localizer++ [14]. The main drawbacks of such toolkits are that the high-level modeling is not supported and the host language may impose some limitations. The advantage of constraint toolkits is that their users do not need to learn a new language. In addition, modelers can develop their libraries for specific domains by using supported data structures of the host languages.

Mathematical Modeling Languages

Mathematical modeling languages such as OPL [15] and AMPL [16] support high-level modeling and provide syntax close to mathematical expressions. As a result, using modeling languages is easy for non-programmers. They allow users to use high-level structures, such as records, mathematical notations, arrays and sets. Furthermore, they allow users to define the required constraints and functions and apply them into the model. However, current modeling languages are solver-dependent and usually force modelers to use a specific solving method. For example mathematical and constraint satisfaction techniques are used in OPL.

Specification Languages

Specification languages are very high level and support sets, relations, and functions. They cannot be used effectively for combinational and optimization problems due to the large gap between the conceptual model and the design model [17].

C. ILP Solvers

There are different examples of ILP solvers such as: CPLEX [13] and MINOS [18]. These are constraint solving toolkits suitable for ILP models. In this paper, we are interested in the CPLEX solver using CGL which was designed to facilitate the generation of CPLEX compatible constraints. Currently, CPLEX is used by more than 1300 companies, governmental organizations, and more than 1000 university researchers.

D. Motivating Example

C. Burguiere and C. Rochange [19], used the following equations 1-4 to describe the constraints needed to incorporate the branch predictor analysis into the Worst Case Execution Time (WCET) computation with the IPET method.

$$\forall b \in B \quad x_b = \sum_{\pi \in \Pi} x_b^{\pi} \tag{1}$$

$$\forall b \in B, \forall \pi \in \Pi_b \quad x_b^{\pi,00} = \sum_{p \in P_b^{\pi}} (x^{\pi,00}_{\overset{0}{p\rightarrow},b} + x^{\pi,01}_{\overset{0}{p\rightarrow},b}) + x^{\pi,00}_{start,b} \tag{2}$$

$$\forall b \in B, \forall d \in D \quad x_{b\overset{d}{\rightarrow}} \geq \sum_{\pi \in \Pi_b} \sum_{s \in S_b^{\pi}} \sum_{c \in C} x^{\pi,c}_{b\overset{d}{\rightarrow},s} \tag{3}$$

$$\forall b \in B, \forall \pi \in \Pi_b \quad x_b^{\pi} \leq \sum_{p \in P_b} \sum_{\pi' \in \Pi} x_b^{\pi'} where\, \pi = \pi'.d\, and\, p \overset{d}{\rightarrow} b \tag{4}$$

Each of these equations represents a relatively large set of ILP constraints that are hard to obtain manually. For example equation 1 expands to the following set of ILP constraints as shown in Equation 5a- 5d:

$$x_0 = x_0^{00} + x_0^{01} + x_0^{10} + x_0^{11} \tag{5a}$$
$$x_1 = x_1^{00} + x_1^{01} + x_1^{10} + x_1^{11} \tag{5b}$$
$$x_2 = x_2^{00} + x_2^{01} + x_2^{10} + x_2^{11} \tag{5c}$$
$$x_3 = x_3^{00} + x_3^{01} + x_3^{10} + x_3^{11} \tag{5d}$$

We designed CGL to allow this set of constraints to be automatically generated from a description that looks almost identical to equation 1.

III. CGL Design

This section describes the (CGL) design. CGL is a DSL for describing constraints using high-level constructs, which are not typically accepted by ILP solvers such as quantifiers, summations, and guards (conditions). Figure 1 illustrates the process to validate and generate the CPLEX constraints from the proposed DSL. The process is divided into three phases. First, the parsing phase is responsible for validating the CGL constraints from a syntactic perspective and constructing an Abstract Syntax Tree (AST) as an intermediate representation of the high-level constraints.

Second, the translation phase is responsible for generating an elaborated AST. The elaborated AST can be seen as an intermediate representation of the final constraints to be generated. After the translation high level constructs such as quantifiers and summations do not exist. Finally, the last phase is a pretty-printer phase responsible for generating the final ILP constraints in a printable format that can be fed to CPLEX for solving. Building the proposed CGL using the three phases process is advantageous in terms of extensibility and flexibility. For instance decoupling parsing from translation allows the following:

- The validation of the CGL input first without performing the translation.

Fig. 1. CGL Design Phases

- The development of several translators, which target several programming languages and/or numerical solvers, without modifying the parser.

A. CGL Syntax

In this section, we present the syntax of the proposed CGL. The syntax is given as Extended Backus-Naur Form (EBNF) grammar rules in Fig. 2.

CGL syntax

The first production, line 1 of Fig. 2, involves the constraint non-terminal. It is constituted of three parts: The first part is an optional part which indicates whether the constraint includes a quantifier or not. The second part is mandatory and represents the relational operation (lhs, relop, and rhs). Finally, the third part is an optional part that shows the condition. As a result, the constraint can take four forms.

The production at line 2 indicates that the constraint may have one or more comma-separated quantifiers. Each quantifier must begin with the forall keyword and the range of the quantifier is specified in the third production. The fourth production indicates that one or more conditions of different types may be applied on the constraint and the relation between conditions is either a disjunction or a conjunction. The types of the conditions are specified in the fifth production. The first alternative of the condition is the shifting condition which means shifts ID2 to the left by one bit (the old leftmost bit is therefore discarded) and puts the value of ID2 as the rightmost bit and this will be related to ID1 according to RELOP. The other alternative is the leading condition which means ID3 is the successor of ID1 using the value of ID2. The lhs can be a number or a string as indicated in rule 6. A string can take any form that is specified in rule 9 such as x_b or $x_b \xrightarrow{3}$. Rhs may be an rhsexp or the sum of multiple rhsexp. Each rhs expression, represented by the non-terminal rhsexp, is made up of zero or more summation followed by exp as in rule 10. There are two alternative for exp. The first one is a string and the other is a sum of multiple strings.

B. CGL Semantics

In this section, we present the semantics of the proposed CGL. The main goal of the translation phase is to get rid of the high level constructs of CGL to generate constraints that would be compatible with typical ILP solvers. The main constructs that we need to expand away are quantifiers, summations, and conditions. The elaboration semantics are defined using the big-step operational semantics listed in Fig. 3.

Small step semantics and big step semantics are two different approaches for defining the meaning of programs [20]. There are two differences between them. The first is the start of the evaluation rule while the second is the conduction of the proofs. In big step, the result is obtained in one step where in small step the result is obtained after doing several steps. The main mathematical tool used in operational semantics is induction [20]. The induction rules have the following general form:

$$\frac{Antecedents}{Consequent} \quad Conditions$$

A rule has a number of premises (nominator) and one conclusion (denominator). Moreover, a number of conditions (written to the right of the rule) may be contained in the rule and these conditions should be satisfied whenever the rule is applied. Rules with an empty set of premises are called axioms and inference rules that characterize semantic behavior of the language constructs [21]. A relation between terms M and values V is defined as follows: M $\Downarrow$ V It can be read as M is the CGL construct while v is the final result that is obtained at the end of translating M.

In this paper, we implement a simple operational semantics directed language as illustrated in Fig. 3. Any constraint is translated by one of the six rules from 1.1 to 1.4.2 in Fig. 3. The rhs term of the constraint is elaborated, using rule from 10 to 11.2 according to its form, to get an expanded list of the constraint meanwhile there is no substitution for lhs and relop to get the first form of constraint. Rule 1.2.1 represents how to get rid of one quantifier in any constraint. Getting rid of a quantifier means replacing each ID, appearing in quantifier, in lhs and substituted rhs with ID range indicated in quantifier. Multi quantifiers are supported in CGL and applied over rhs and lhs as seen in rule 1.2.2 as applied in rule 1.2.1 but it applied many times. In rule 1.3, the aim is to get rid of the condition construct. There are different types of conditions as shown in Fig. 2 lines 4 and 5. Any condition is applied to substituted rhs term to get new rhs terms which satisfy these conditions. The two rules 1.4.1 and 1.4.2 are the merged forms of all preferred rules i.e. the constraint contains condition and quantifier construct. In rule 1.4.1, one quantifier and condition are applied over the rhs and lhs as in the way explained in rule 1.2.1 and 1.3. Moreover, multi quantifiers are applied in rhs and lhs as seen in rule 1.4.2. Getting rid of summation construct in rhs term is reached in two steps. First, rhs is compensated by rhs' and also rhsexp is replaced by rhsexp' as illustrated in rule 10. Second, each exp's ID, matched summation ID, is substituted with summation ID range as shown in rule 11.1. While removing multi-summation i shown in rule 11.2 like rule 11.1 but it is applied many times according to number of summation.

IV. Implementations and Results

Our proposed CGL has several applications in computer science. The lex and yacc tools were used for realizing the CGL parser. The CGL translator is written and debugged using

1	constraint	::=	[quantifier] lhs relop rhs [WHERE condition]
2	quantifier	::=	FORALL range | FORALL range , quantifier
3	range	::=	ID BELONG integer TO integer
4	condition	::=	condlist | condlist (AND | OR) condition
5	condlist	::=	ID relop ID SHIFT ID | ID LEADS ID WITH ID
6	lhs	::=	string | integer
7	string	::=	ID _ repstaff ^ repstaff
8	repstaff	::=	staff | staff COMMA repstaff
9	staff	::=	ID | integer | ID LEADS integer | ID LEADS ID
10	rhs	::=	rhsexp | rhsexp + rhs
11	rhsexp	::=	{SUM range} exp
12	exp	::=	string | string + exp
13	relop	::=	$>$ | $\geq$ | $<$ | $\leq$ | = | !=

Fig. 2. CGL Grammar

1	$\frac{rhs \Downarrow rhs'}{lhs\, relop\, rhs \Downarrow lhs\, relop\, rhs'}$	1.1
2	$\frac{rhs \Downarrow rhs'}{forall\, ID\, belong\, int1\, to\, int2\, lhs\, relop\, rhs \Downarrow \{rel : rel \in lhs\,[ID \longmapsto int]\, relop\, rhs'\,[ID \longmapsto int]\, where\, int1 \leq int \leq int2\}}$	1.2.1
3	$\frac{quantifier\, lhs\, relop\, rhs \Downarrow cs}{forall\, ID\, belong\, int1\, to\, int2\,, quantifier\, lhs\, relop\, rhs \Downarrow \{rel : rel \in cs\,[ID \longmapsto int]\, where\, int1 \leq int \leq int2\}}$	1.2.2
4	$\frac{rhs\, where\, condition \Downarrow rhs'}{lhs\, relop\, rhs\, where\, condition \Downarrow lhs\, relop\, rhs'}$	1.3
5	$\frac{rhs \Downarrow rhs'}{forall\, ID\, belong\, int1\, to\, int2\, lhs\, relop\, rhs\, where\, condition \Downarrow \{rel : rel \in lhs\,[ID \longmapsto int]\, relop\, rhs'\,[ID \longmapsto int]\}}$	
	$where\, int1 \leq int \leq int2\, and\, condtion$	1.4.1
6	$\frac{quantifier\, lhs\, relop\, rhs \Downarrow cs}{forall\, ID\, belong\, int1\, to\, int2\,, quantifier\, lhs\, relop\, rhs\, where\, condition \Downarrow \{rel : rel \in cs\,[ID \longmapsto int]\}}$	
	$where\, int1 \leq int \leq int2\, and\, condtion$	1.4.2
7	$\frac{rhs \Downarrow rhs'\,, rhsexp \Downarrow rhsexp'}{rhsexp\, +\, rhs \Downarrow rhsexp'\, +\, rhs'}$	10
8	$[sum\, ID\, belong\, int1\, to\, int2]\, exp \Downarrow \Sigma\{rel : rel \in exp\,[ID \longmapsto int]\, where\, int1 \leq int \leq int2\}$	11.1
9	$\frac{\{sum\, range\} \Downarrow e1}{[sum\, range]\,\{sum\, range\}\, exp \Downarrow \Sigma\{rel : rel \in e1\,[ID \longmapsto int]\, where\, int1 \leq int \leq int2\}}$	11.2

Fig. 3. CGL Semantics

the C language. Our implementation of CGL is open source and can be freely downloaded [22].

The CGL is tested with different inputs to ensure that the output is compliant with the ILP format. Equation 6 is an example of constraint that contains multiple summations. The result of removing summation construct from rhs, as indicated in rule 11.2 of Fig. 3, and substituting it by rhs' in constraint rule as illustred in rule 11.2 is indicated in Equ. 7. An example of getting rid of a quantifier as in rule 1.2.1 is shown in Equ. 9 and the input in Equ.8. While removing multiple quantifiers as in rule 1.2.2 is indicated in Equ. 11 and input is shown in Equ.10. The output of applying condition over constraint as illustrated by rule 1.3 is shown in Equ.13 while the input is shown in Equ. 12. Finally, an example that includes removing condition, quantifier and summation of the constraint is shown in Equ. 15 and the input in indicated in Equ. 14.

$$v = sum(x\, belong\, 0\, to\, 3)\, sum(y\, belong\, 4\, to\, 8)\, [v\_x, y] \quad (6)$$

$$\begin{aligned} v = [&v\_0,4 + v\_0,5 + v\_0,6 + v\_0,7 \\ &+ v\_0,8 + v\_1,4 + v\_1,5 + v\_1,6 \\ &+ v\_1,7 + v\_1,8 + v\_2,4 + v\_2,5 \\ &+ v\_2,6 + v\_2,7 + v\_2,8 + v\_3,4 \\ &+ v\_3,5 + v\_3,6 + v\_3,7 + v\_3,8] \end{aligned} \quad (7)$$

$$forall(b\, belong\, 0\, to\, 3)\, x\_b = sum(pi\, belong\, 5\, to\, 7)[x\_b \wedge pi] \quad (8)$$

$$\begin{aligned} x\_0 &= [x\_0 \wedge 5 \ + \ x\_0 \wedge 6 + x\_0 \wedge 7] \\ x\_1 &= [x\_1 \wedge 5 \ + \ x\_1 \wedge 6 + x\_1 \wedge 7] \\ x\_2 &= [x\_2 \wedge 5 \ + x\_2 \wedge 6 + x\_2 \wedge 7] \\ x\_3 &= [x\_3 \wedge 5 \ + x\_3 \wedge 6 + x\_3 \wedge 7] \end{aligned} \tag{9}$$

$$\begin{aligned} forall(s\, belong\, 0\, to\, 2), forall(d\, belong\, 0\, to\, 1) m\_s \rightarrow d \\ = sum(pi\, belong\, 0\, to\, 3)[m\_s \rightarrow d \wedge pi] \end{aligned} \tag{10}$$

$$\begin{aligned} m\_0 \rightarrow 0 &= [m\_0 \rightarrow 0 \wedge 0 + m\_0 \rightarrow 0 \wedge 1 \\ &\quad + m\_0 \rightarrow 0 \wedge 2 + m\_0 \rightarrow 0 \wedge 3] \\ m\_1 \rightarrow 0 &= [m\_1 \rightarrow 0 \wedge 0 + m\_1 \rightarrow 0 \wedge 1 \\ &\quad + m\_1 \rightarrow 0 \wedge 2 + m\_1 \rightarrow 0 \wedge 3] \\ m\_2 \rightarrow 0 &= [m\_2 \rightarrow 0 \wedge 0 + m\_2 \rightarrow 0 \wedge 1 \\ &\quad + m\_2 \rightarrow 0 \wedge 2 + m\_2 \rightarrow 0 \wedge 3] \\ m\_0 \rightarrow 1 &= [m\_0 \rightarrow 1 \wedge 0 + m\_0 \rightarrow 1 \wedge 1 \\ &\quad + m\_0 \rightarrow 1 \wedge 2 + m\_0 \rightarrow 1 \wedge 3] \\ m\_1 \rightarrow 1 &= [m\_1 \rightarrow 1 \wedge 0 + m\_1 \rightarrow 1 \wedge 1 \\ &\quad + m\_1 \rightarrow 1 \wedge 2 + m\_1 \rightarrow 1 \wedge 3] \\ m\_2 \rightarrow 1 &= [m\_2 \rightarrow 1 \wedge 0 + m\_2 \rightarrow 1 \wedge 1 \\ &\quad + m\_2 \rightarrow 1 \wedge 2 + m\_2 \rightarrow 1 \wedge 3] \end{aligned} \tag{11}$$

$$\begin{aligned} v = sum(x\, belong\, 0\, to\, 7) sum(y\, belong\, 3\, to\, 6) \\ sum(d\, belong\, 0\, to\, 1)[v\_y \wedge x, d] where\, y \ = x\, shift\, d \end{aligned} \tag{12}$$

$$v = [v\_3 \wedge 2, 0 + v\_4 \wedge 2, 1 + v\_5 \wedge 3, 0 + v\_6 \wedge 3, 1] \tag{13}$$

$$\begin{aligned} forall(b\, belong\, 0\, to\, 2), forall(pi\, belong\, 0\, to\, 2) x\_b \wedge pi = \\ sum(p\, belong\, 2\, to\, 3) sum(pic\, belong\, 0\, to\, 2) \\ sum(d\, belong\, 0\, to\, 1)[x\_p \wedge pic, d] \\ where\, pi = pic\, shift\, d\, and\, p\, leads\, to\, pic\, with\, d \end{aligned} \tag{14}$$

$$\begin{aligned} x\_0 \wedge 0 &= [x\_2 \wedge 0, 0 + x\_3 \wedge 0, 0] \\ x\_1 \wedge 0 &= [x\_2 \wedge 0, 0 + x\_3 \wedge 0, 0] \\ x\_2 \wedge 0 &= [x\_2 \wedge 0, 0 + x\_3 \wedge 0, 0] \\ x\_0 \wedge 1 &= [x\_2 \wedge 0, 1 + x\_3 \wedge 0, 1 + x\_3 \wedge 2, 0] \\ x\_1 \wedge 1 &= [x\_2 \wedge 0, 1 + x\_3 \wedge 0, 1 + x\_3 \wedge 2, 0] \\ x\_2 \wedge 1 &= [x\_2 \wedge 0, 1 + x\_3 \wedge 0, 1 + x\_3 \wedge 2, 0] \\ x\_0 \wedge 2 &= [x\_2 \wedge 1, 0 + x\_3 \wedge 1, 0] \\ x\_1 \wedge 2 &= [x\_2 \wedge 1, 0 + x\_3 \wedge 1, 0] \\ x\_2 \wedge 2 &= [x\_2 \wedge 1, 0 + x\_3 \wedge 1, 0] \end{aligned} \tag{15}$$

V. Conclusion

In this paper we presented CGL, a DSL for automatic constraint generation from simple high-level equations. Formal syntax, expansion semantics, and implementation details are provided. CGL can be used as illustrated in the motivating example to generate ILP constraints needed to take branch prediction into account when estimating the worst case execution time of a program. CGL can also be used in other domains as well without any modifications. However, CGL provides a proof of concept and can certainly be extended to support more types of constraints and high-level constructs. To the best of our knowledge this paper is the first effort to provide a formal account of such a language.

Another advantage of CGL, is that it can easily be extended to support several ILP solvers. This extension does not require any modifications to the CGL syntax or semantics; instead, it only requires minor tweaks to the pretty-printing step. If such extension is implemented, CGL users should be able to write their high level constraints only once (in CGL) and send the corresponding ILP equations to any ILP solver of their choice without having to manually re-write them.

References

[1] P. V. B. F. Rossi and T, Eds., *Handbook of constraint programming*. Elsevier Science, 2006.

[2] A. V. Deursen and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, p. 35(6):2636, 2000.

[3] S. I. Salgueiro Pedro, Diaz Daniel and A. Salvador, "Using constraints for intrusion detection: The nemode system," *Springer*, 2011.

[4] H. M. Lennart Augustsson and G. Sittampalam, "Paradise: a two-stage dsl embedded in haskell," *ACM SIGPLAN international conference on Functional programming*, pp. 225–228, 2008.

[5] N. H. Nordmann, A. and S. Wrede, "A survey on domain-specific languages in robotics," *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

[6] A. Nordmann, S. Wrede, and J. J. Steil, "Modeling of movement control architectures based on motion primitives using domain-specific languages," in *Int. Conf. Robotics and Automation*. IEEE, 2015.

[7] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry, "Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages," in *ECOOP Workshop on Domain-Specific Program Development*, Nantes, France, July 2006.

[8] M. Vlter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[9] R. Jha, A. Samuel, A. Pawar, and M. Kiruthika, "A domain-specific language for discrete mathematics," *CoRR*, vol. abs/1310.3473, 2013.

[10] N. J. Mohamad Kordafahar and R. Rafeh, "On the optimization of cplex models," *International Research Journal of Applied and Basic Sciences*, 2013.

[11] K. R. Apt and M. Wallace, *Constraint Logic Programming using Eclipse*. Cambridge Univ. Press, 2006.

[12] P. Van Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005.

[13] IBM ILOG, CPLEX 12.1 user manual, 2010.

[14] L. Michel and P. V. Hentenryck, "Localizer++: An open library for local search," Tech. Rep., 2001.

[15] IBM ILOG, OPL Language Reference Manual, 2005.

[16] IBM ILOG, AMPL Version 12.2 Users Guide, 2006.

[17] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace, "The design of the zinc modelling language," *Constraints*, vol. 13, no. 3, pp. 229–267, 2008.

[18] Tomlab Optimization Inc, Users Guide for TOMLAB /MINOS, 2008.

[19] C. Burguiere and C. Rochange, "On the complexity of modeling dynamic branch predictors when computing worst-case execution time," in *Proceedings of the ERCIM/DECOS Workshop On Dependable Embedded Systems*, 2007.

[20] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[21] M. Mernik and M. repinek, "Prolog and automatic language implementation systems," 2005.

[22] *CGL, Constraint Generation Language*, last access on: 8-11-2015. [Online]. Available: http://www.bu.edu.eg/staff/marwaelmenyawi5