A Domain Specific Language for Modeling Differential Constraints of Mobile Robots

Marco Guarnieri, Eros Magri, Davide Brugali, Luca Gherardi

***Abstract*— Kinematics and dynamics constraints of mobile robots can be modeled by means of differential equations. Simulation and sampling based path-planning algorithms need a model of these constraints in order to deal with non-holonomic mobile robots.**

Usually these models are hard-coded in the implementation of those algorithms and this makes hard their reuse. In order to design these algorithms in a modular and extensible way we have to explicitly represent the models of the robots and decouple them from algorithms implementation.

We propose DCML, a Domain Specific Language that can be used in order to describe differential models, and a tool that allows developers to automatically generate the code that implements the model. We also aim to show how this Model-Driven Engineering technique can be used with good results. As a demonstration of what can be done by means of our DSL, we present the differential model of an omnidirectional holonomic robot called BART, and we show how this model can be integrated in a framework for path planning.

I. Introduction

Differential equations are widely used for modeling kinematics and dynamics constraints of mobile robots, for example in simulation and sampling-based path planning algorithms. Differential models express relations between configuration variables. The possible states of a mobile robot are represented in the state space X and each state represents a particular configuration of the robot. For wheeled mobile robots each state $\vec{x} \in X$ is $\vec{x} = (x, y, \theta)$, where x and y represent the position of the robot in the plane and θ is its orientation. In the same way it is possible to define the action space U, which is the set of all the possible actions on all the possible states (an action is a response of the robot, which changes its current state, to an external input). Thus a differential model can be represented as $\dot{\vec{x}} = f(\vec{x}, \vec{u})$ where $\vec{x} \in X$ is the starting state, $\vec{u} \in U$ is the action applied to the model and f is a function, called state transition function, that defines the relation between state space and action space [1, Ch. 13]. The results are expressed in terms of velocities $\dot{\vec{x}}$ and the outcome of their integration represents the future states that satisfy the kinematics constrains.

These models are widely used in simulation algorithms (that, given the starting configuration and the action vector, compute the final configuration of the robot) and sampling-based motion planning algorithms (that sample collision free configurations that need to be compatible with the differential constraints). In order to compute final configurations it is necessary to solve the differential equations and this can be done by means of solvers, which use numerical approximation techniques. However solvers require that differential equations are implemented in the source code fulfilling specific interfaces, and implementing these equations is usually error prone and not trivial. Another problem is that differential models are usually hard-coded in the implementation of these algorithms, hence the algorithm implementation is hard-coupled to the specific robot.

In order to achieve higher flexibility, modularity, easier extensibility with respect to the current situation and to solve the problems presented before, a higher level representation of those models is needed. Domain specific languages (DSLs) provide this higher level representation. DSLs are simple formal languages, usually declarative, used to represent domain specific knowledge using some sort of syntax. DSLs let you describe easily a scenario in a specific domain.

The syntax and semantic of these DSLs are designed explicitly to describe only the knowledge of a specific domain and thus DSLs have a gain in terms of expressiveness and ease of use compared to general purpose languages for the specific domain. Conversely they are usually less expressive than general-purpose programming languages out of their domain. In this way they can also improve productivity and maintenance costs. More details on DSLs can be found in [2] and [3].

DSLs have also other advantages over general-purpose languages while expressing knowledge in the specific domain:

- being less expressive and complex than general purpose languages, DSLs can be used also by people that are not expert programmers;
- manual implementation of the model can require experience in computer programming and it is error prone while you can usually generate code from a DSL document in an in automated way;
- the model itself can be used as documentation;
- ease the communication between programmers and domain experts;
- ease the description of the scenario;
- can decouple the representation of the model from the technologies and interfaces used in the implementation.

In this paper we present DCML, Differential Constraints Modeling Language, a Domain Specific Language that allows the description of such kind of models with a high level of abstraction from implementation details. DCML, in addition to the advantages presented above, provides devel-

M. Guarnieri, E. Magri, D. Brugali and L. Gherardi are with the Dept. of Information Technology and Mathematics, University of Bergamo, 24044 Dalmine, Italy 0guarnieri.marco0@gmail.com, erosmagri@gmail.com, brugali@unibg.it, luca.gherardi@unibg.it

opers with an automatic way to generate the code that implements the differential equations starting from the differential model, which describes the relations between state space and action space of a specific robot. This implementation can then be used by motion planning algorithms in order to do the simulation of the behavior of the robot itself. In order to develop DCML, we have followed a Model Driven Engineering (MDE) approach. MDE has already shown good results in robotics in terms of reusability and integration, as shown in [4] or [5], and thus we aim to demonstrate that this approach can be applied with good results also to the representation of differential models.

Section II presents some related work. Section III describes DCML more in detail, while Section IV shows how our language can be used for writing the differential model of an omnidirectional holonomic robot and how the generated code can be integrated in a framework for simulation and planning for mobile robots. Finally Section V presents our conclusions.

II. Related Work

Despite our research and this paper focus only on the use of differential models in sampling-based motion planning algorithms for the simulation of robots behaviour in response to specific actions, differential models are widely used also in other robotics fields. They can be used to describe several kinds of robots: [1] and [6] present differential models of some wheeled mobile robots under kinematics and dynamics constraints, while [7] shows a model of an hexapod robot. An extension of differential models, that can take into account also dynamics constraints, are phase-space models that consider also accelerations and can then be described as $\ddot{x} = f(\dot{x}, x, u)$. Each second order model can be converted in a first-order model, which is a differential model, using a *phase space*, that has more dimension than the *state space* of the second order model. In this way we can represent, using differential models, also dynamic constraints. In the same way a kth-order model can be expressed as a differential model using an adequate *phase space*. Thus differential models can be used for motion planning under kinematics and dynamics constraints, as shown in [1, Ch. 14].

A first way of defining differential models with an higher level of abstraction than hardcoded solutions is using Simulink [1]. It provides developers with a toolchain for defining, through block diagrams, differential models and generating from these diagrams C and C++ code that implements them. In our opinion this approach is not flexible enough because it does not allow the generation of code in other programming languages and also because it does not allow developers to customize the generated code, in terms of interface and optimization.

Another approach is using a dedicated Domain Specific Language. Literature presents, up to now, a few DSLs to describe differential equations. The MyFEM language, presented in [8], is a DSL for the definition of partial-differential equations using a subset of the Python language. It allows the generation of C++ code that implements the model defined in MyFem but it has not got an IDE. Scalation [9] is an embedded DSL defined over the Scala programming language, and it has a package that allows the representation of systems of differential equations. These approaches use subsets of existing programming languages to define the DSLs. This has some advantages, such as less learning time, but it has also the big drawback that the resulting DSL is too close to the general purpose language and thus it has less abstraction than a dedicated DSL and requires too much effort to be used by users that are not expert programmers. Another drawback of both approaches is that the syntax used to express differential equations is too far from the mathematical formalism because it is tied to the syntax of native programming languages.

Other approaches, such as the one in [10] that defines a specification language for partial differential equations on a union of rectangles, or [11] that defines differential equations using the arrow notation, are, in our opinion, too complex and difficult in order to be used as an effective aid to developers. A common disadvantage of all these approaches is that they are tied to work only with a fixed set of numerical solvers.

Given the fact that existing solutions for representing differential equations are too complex or do not offer enough flexibility in the code generation phase we decided to create a new DSL for representing differential models. DCML offers two advantages with respect to existing solutions. Firstly the syntax used to describe differential equations is close to the mathematical one, and secondly DCML is not tied to work with a fixed set of differential equations solvers.

III. Differential Constraints Modeling Language

DCML allows users to describe constraints that affect mobile robots by means of differential models. In this way users can focus on the description of the differential equations.

A simple model, taken from [1], that can be used to describe the constraints of a differential drive, a mobile robot with two independent wheels, is presented in 1.

$$\begin{aligned} \dot{x} &= \frac{r}{2}(u_l + u_r)cos\theta \\ \dot{y} &= \frac{r}{2}(u_l + u_r)sin\theta \\ \dot{\theta} &= \frac{r}{L}(u_r - u_l) \end{aligned} \tag{1}$$

The state vector (x, y, θ) represents the cartesian position of the robot while the action vector $u = (u_l, u_r)$ represents angular velocities of the wheels, r is the radius of each wheel while L is the distance between the two wheels.

Listing 1 shows the DCML document representing the differential drive presented above, and it will be used to describe how our language works. It describes the model of the differential drive presented above. A document written

[1] Simulink - www.mathworks.com/products/simulink/

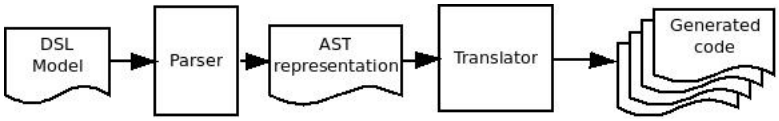

Fig. 1. Validation and Code generation process

by means of our DSL can describe several models and for each model the user can specify:

- The action space: after the keyword **ACTION** the user can specify all the actions. In the differential drive example we have two actions u_l and u_r.
- The configuration space: after the keyword **CONFIG** the user can specify the dimensions of each configuration. In the example each configuration can be expressed in terms of x, y, θ.
- After the keyword **PARAM** the user can define the parameters of the model. In our example they are r and L.
- The state transition function of the model can be expressed by means of differential equations in an understandable way.
- After the keyword **VAR** the user can define some temporary variables that can be used to ease the definition of differential constraints.
- After the keyword **CONST** the user can define some constant values, different from the predefined ones, such as π and e.
- After the keyword **PACKAGE** the user can define the package in which the source code will be created. In the example we have decided that we want to create the source code in the package *robotics.models*.
- If the model definition isn't expressive enough, further comments can be added with a JavaDoc style notation.

```
1  BEGIN DifferentialDrive
2   PACKAGE : robotics.models;
3   ACTION : u_l, u_r;
4   PARAM : L, r;
5   CONFIG : x, y, theta;
6
7   d(x) = r / 2 * (u_l + u_r) * cos(theta);
8   d(y) = r / 2 * (u_l + u_r) * sin(theta);
9   d(theta) = (r / L) * (u_r - u_l);
10 END;
```

Listing 1. Differential Drive model

While actions, configurations and differential equations are mandatory, the other elements are useful only for describing more complex models (see Section IV). We will describe, now, the structure of the grammar of our DSL, that is shown in Listing 2. A grammar has four main components, [12]:

1) a set Σ of terminals, which are the basic symbols that form valid instructions of our language;
2) a set V of non-terminals, that are syntactic variables that represent set of strings;
3) a non-terminal $s \in V$ that acts as start symbol;
4) a set P of productions, that define how terminals and non-terminals can be combined in order to generate valid strings.

For our grammar the set Σ is equal to {*"**", *"*\"*, *"BEGIN"*, *"END"*, *";"*, *"PACKAGE"*, *":"*, *"ACTION"*, *"PARAM"*, *"CONST"*, *"CONFIG"*, *"VAR"*, *","*, *"+"*, *"-"*, *"*"*, *"\"*, *"("*, *")"*, *"d("*, *ID, PCKG_ID, NUM, COMMENT*} where ID represents an alphanumerical identifier, NUM is a numeric literal and $PCKG\_ID$ is a package identifier. The set V is composed by {*modelList, model, package, actions, params, constants, configurations, variables, varList, constList, varDef, constDef, assignments, assignment, var, expr, term, factor, paramList*} and the start symbol is $modelList$.

The grammar is expressed using the Extended Backus-Naur Form (EBNF) [13] that describes each production in the form $A \rightarrow f(V_1, \ldots, V_n, \alpha_1, \ldots, \alpha_m)$ where $A, V_1, \ldots, V_n \in V$, $\alpha_1, \ldots, \alpha_m \in \Sigma$ and f is a function that concatenates symbols using regular expressions. A production means that the non-terminal on the left hand side can be replaced by the regular expression on the right hand side of the $\rightarrow$ operator.

```
1  modelList -> model(model)*
2  model -> ["/**" COMMENT "*/"] BEGIN ID [package]
       actions [params] [constants] configurations
       [variables] assignments END";"
3  package -> PACKAGE ":" PCKG_ID";"
4  actions -> ACTION ":" varList ";"
5  params -> PARAM ":" varList ";"
6  constants -> CONST ":" constList ";"
7  configurations -> CONFIG ":" varList ";"
8  variables -> VAR ":" varList ";"
9  varList -> varDef ("," varDef)*
10 constList -> constDef ("," constDef)*
11 varDef -> ID
12 constDef -> ID "=" ("+" | "-")NUM
13 assignments -> assignment (assignment)*
14 assignment -> var "=" expr";"
15 var -> ID | "d(" ID ")"
16 expr -> term ( ("+" | "-")term )*
17 term -> factor ( ("*" | "/")factor )*
18 factor -> NUM | "(" expr ")" | var["("paramList")"]
19 paramList -> expr("," expr)*
```

Listing 2. DSL Grammar

The first production (row 1) involves the $modelList$ terminal, and means that a document of our DSL must contain at least one model. The second production describes the syntax of each model, it must be enclosed between a *"BEGIN"* instruction and an *"END"* instruction and the ID must be unique in the document. Symbols enclosed between square brackets are optional. In the differential drive example the ID is *DifferentialDrive*.

The productions at rows 4,5,7,8 define that, after the specific keywords, a list of variable declarations is needed. These lists represent, respectively, actions, parameters, configurations, and temporary variables. Each variable list, represented

```
1  package robotics.models;
2
3  public class DifferentialDrive implements IFirstOrderModel {
4
5          private double L, r, u_l, u_r;
6
7          public void setAction(double[] actions) {
8                  if (actions.length != 2)
9                          throw new IllegalArgumentException(``Actions must have size 2.'');
10                 u_l = actions[0];
11                 u_r = actions[1];
12         }
13         public void setParameters(double[] parameters) {
14                 if (parameters.length != 2)
15                         throw new IllegalArgumentException(``Parameters must have size 2.'');
16                 L = parameters[0];
17                 r = parameters[1];
18         }
19         public void computeDerivatives(double t, double[] y, double[] yDot) throws DerivativeException{
20           yDot[0] = r / 2 * (u_l + u_r) * java.lang.Math.cos(y[2]);
21                 yDot[1] = r / 2 * (u_l + u_r) * java.lang.Math.sin(y[2]);
22                 yDot[2] = (r / L) * (u_r − u_l);
23         }
24 }
```

Listing 3. Differential Drive implementation

by the non-terminal $varList$, is made up of one or more variable declarations (row 9), each one consisting in an ID, as shown in the production at row 11, that must be unique in the model. In a similar way the production that has as head the non-terminal $constants$ (row 6) defines that, after the keyword *"CONST"*, a list of constant declarations $constList$ (row 10) is needed. Each constant declaration is made up of an ID, unique in the model, and a numeric literal, as shown in the production 12.

The non-terminal $assignments$ can be replaced by a list of differential equations. Each equation is defined as $var = expr$, where var is a non-terminal that represents, as shown in production 15, either a differential variable of the first order, or an already defined identifier. $expr$ represents an algebraic expression composed by predefined functions, such as sin or cos, parenthesized expressions, numeric literals, predefined constants, such as π, or instances of the var non-terminal and also the usual mathematical operators $+, -, *, /$.

In order to validate and generate the code that implements the models expressed using our DSL we have defined the process shown in Figure 1. It can be divided in two phases. In the first one, the parsing phase, the document is validated. The parser checks that the document is correct, both from a syntactic point of view (it must respect the syntactic rules) and also from a semantic point of view (e.g. the parser checks that the document does not contains undeclared variables, non unique identifiers or function invocations with a wrong number of parameters). In this phase the parser builds, starting from the document, the Abstract Syntax Tree (AST) that is an intermediate representation of the model. The AST is a tree representation of the syntactic structure of the model enriched with some useful semantic information elaborated during the parsing phase.

Taking the AST as input, we can start the second phase, i.e. the translation phase, which creates the code that implements the differential model by means of a general purpose programming language. This can be done by simply visiting the AST, because it is a tree structure bearing all the information needed for the translation.

The decision of generating an intermediate representation by means of ASTs, instead of performing directly the translation during the parsing phase has some advantages:

- allows the validation of the model without performing the translation;
- by decoupling the translation form the parsing phase we can develop and use several translators, which target several programming languages and/or numerical solvers, without modifying the parser. This is possible because the parsing phase is completely separated by details regarding the generation of the code.

Despite this solution is a bit less efficient than performing the translation during the parsing phase, it has great advantages in terms of extensibility and flexibility. The Java code generated from the differential drive example is shown in Listing 3. This code is written to be compatible with numerical solvers provided by the *Apache Commons Math* library[2]. The generated class implements the interface *IFirstOrderModel*, which extends the interface *FirstOrderDifferentialEquation* defined in the *Apache Math* library. It defines three methods: 1) *computeDerivatives*, called by the solver, contains the definition of the state transition function (the state $\vec{x}$ is mapped on the array y, while the velocities $\dot{\vec{x}}$ are mapped on $yDot$), 2) *setParameter* that can be used to set the parameters of a specific robot, 3) *setAction* that can be used to set the actions. The methods *setParameter* and *setAction* are called by the simulator.

Using our DSL users can focus only on modeling the state transition function of the robot. The code can be

[2]Apache Commons Math - http://commons.apache.org/math/

automatically generated by translators optimized accordingly to both the destination programming language and the model interface. In this way the details related to the model implementation (e.g. the numerical solver used to solve the equations) can be completely hidden to the user and the task of creating optimized code can be delegated to the writer of the translator. Our approach allows developers to define new translators in order to generate code optimized accordingly to real-time and computational requirements.

The grammar of our DSL was defined by using AntLR3[3]. We used it also for the definition of the semantic actions and for building the AST tree. AntLR is a parser generator that reduces the time and effort needed to build and maintain language processing tools.

One of the problems of MDE is that developers, in order to use MDE techniques, usually need tools that support them in the management and development of models. Thus, in order to create such a tool we have decided to use the Xtext framework[4], which provides a simple way for creating textual DSLs and to automatically generate a full-featured Eclipse Text Editor from the grammar. The grammar implemented in the Xtext editor is the same used for the parser, without semantic actions.

We choose to implement the parser separately from the editor for two reasons. First, thanks to AntLR we can have better control on the definition of the syntax and the semantic of our language and on the AST creation phase than using Xtext. Second, by using AntLR we have created a tool that can be used also in a stand-alone way, or can be integrated in others IDEs.

In this way we can use Xtext in order to integrate our DSL in the Eclipse IDE, that is by now one of the de facto standards in terms of IDEs and has several plugins related to model driven engineering. This integration gives to developers useful features such as auto-completion and syntax highlighting, while expressing the grammar using AntLR give us the power of expressing complex semantic rules.

In order to integrate the parser in the editor we have added a button that allows the invocation of the parser, which takes the model as input and validate it. Then, in the case that the model is correct, the Java translator is invoked. It takes the AST produced by the parser and translate it into the Java class that implements the differential equations of the model and fulfills the interface for differential models.

IV. Case Study

Using DCML we can describe models of simple robots, such as the differential drive described in Section III, or models of more complex robots like BART.

BART is an omnidirectional holonomic wheeled robot, developed by the Software for Experimental Robotics Lab (SERL) at the University of Bergamo. It is made up of two steering blocks and two free wheels. Each block is a differential drive and the rotational joint is not on the axis of its wheels. The mechanical structure of the robot is presented in Figure 2. BART is slightly similar to the robots presented in [14] and [15].

As a case of study we will show how the kinematic model of BART can be expressed by means of our DSL. While

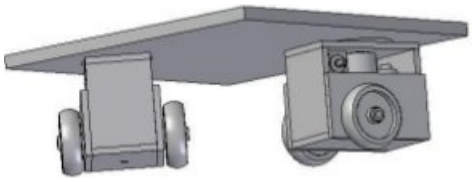

Fig. 2. BART robot

modeling the kinematics of BART robot defining the rigid bodies that made it up can be quite difficult, modeling the general kinematics of the robot can be done quite easily using differential models, and thus in our DSL, as shown in Listing 4.

We choose to use as configuration space of the BART robot the variables x, y and $theta$ (cartesian position and the orientation of the center of the robot), the variables x_front, y_front and phi_front (cartesian position and orientation of the joint that connects the base of the robot to the frontal differential drive, related to the absolute reference system), and the variables x_rear, y_rear and phi_rear (position and orientation of the rear steering block).

The parameters of the model are $tt\_wheel\_half\_axis$, that represents the half length between the wheels in one of the differential drives, tt_wheel_radius, that is the radius of each wheel of the steering blocks, and tt_steer_offset, that is the distance between the steering axis and the axis of the wheels of the robot. We introduce a variable k that represents the ratio between tt_steer_offset and $tt\_wheel\_half\_axis$ to simplify the writing of the differential equations. The actions accepted by the robot are the values $left\_f\_s$ and $right\_f\_s$, that represent the angular speeds of left and right wheels of the frontal steering block, and the values $left\_r\_s$ and $right\_r\_s$, that represent the angular speeds of left and right wheels of the rear differential drive.

The differential model of BART can be divided in three parts. The first one, that involves x_front, y_front and phi_front variables, expresses the differential equations needed to compute the position of the joint of the front steering block. It is an extension of the differential model for a standard differential drive that considers also the fact that the rotational joint is not on the axis of the differential drive. The second part, involving variables x_rear, y_rear and phi_rear, is quite similar to the first one but it is related to the rear steering block. The last part of the model, involving variables x, y and $theta$, computes the position of the center of the BART robot. This part is not made up of differential equations because these values can be computed with algebraic equations from the values of the two steering blocks.

We developed a Java framework that implements some well known algorithms for sampling-based path planning.

[3]ANother Tool for Language Recognition - `http://www.antlr.org/`

[4]Xtext - `http://www.xtext.org/`

```
BEGIN Bart
 PACKAGE : robotics.models;
 ACTION : left_f_s, right_f_s, left_r_s, right_r_s;
 PARAM : tt_wheel_half_axis, tt_wheel_radius, tt_steer_offset;
 CONFIG : x, y, theta, x_front, y_front, phi_front, x_rear, y_rear, phi_rear;
 VAR : k;

 k = tt_steer_offset / tt_wheel_half_axis;

 d(x_front) = (tt_wheel_radius /2) * (((cos(phi_front) - k * sin(phi_front)) * right_f_s) + ((cos(phi_front
      ) + k * sin(phi_front)) * left_f_s));
 d(y_front) = (tt_wheel_radius /2) * (((sin(phi_front) + k * cos(phi_front)) * right_f_s) + ((sin(phi_front)
       - k * cos(phi_front)) * left_f_s));
 d(phi_front) = (tt_wheel_radius / (2 * tt_wheel_half_axis)) * (right_f_s - left_f_s);

 d(x_rear) = (tt_wheel_radius /2) * (((cos(phi_rear) - k * sin(phi_rear)) * right_r_s) + ((cos(phi_rear) + k
      * sin(phi_rear)) * left_r_s));
 d(y_rear) = (tt_wheel_radius /2) * (((sin(phi_rear) + k * cos(phi_rear)) * right_r_s) + ((sin(phi_rear) - k
      * cos(phi_rear)) * left_r_s));
 d(phi_rear) = (tt_wheel_radius / (2 * tt_wheel_half_axis)) * (right_r_s - left_r_s);

 x = (x_front + x_rear)/2;
 y = (y_front + y_rear)/2;
 theta = atan2((y_front-y_rear),(x_front-x_rear)) + (pi / 4);
END;
```

Listing 4. BART model

All these algorithms depend on the model of the robot under simulation and thus, the differential model is, for all of them, an input parameter. Using DCML we can modify the implementation of the differential model without changing directly any line of source code. We simply have to modify the DCML document and regenerating from it the new code. In order to do this we have developed a DCML to Java translator that, taken as input the AST of the model, creates the class that implements the model itself.

V. Conclusions

In this paper we have shown how, using a Domain Specific Language, it is possible to describe the differential model of a mobile robot. We have also shown how an intermediate representation of the model by means of an AST is useful in order to use translators that can generate optimized code for any target platform. In this way the model is independent from the actual implementation. We have also presented the tool suite that can be used in order to define and generate implementations of DCML models.

This work shows that MDE can be used with good results in specific areas, such as the representation of differential models, in which the representation of the knowledge can be formalized in a defined model. The integration of the generated models in the path planning framework demonstrates that the technique can have useful application also in a real environment.

Appendix

The DCML Eclipse Tool and some example can be found at *http://robotics.unibg.it/software/dcml/*.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

References

[1] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[2] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35, 2000.

[3] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4), 2005.

[4] C. Schlegel, T. Haßler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*. IEEE, 2009.

[5] Christian Schlegel, Andreas Steck, Davide Brugali, and Alois Knoll. Design abstraction and processes in robotics: from code-driven to model-driven engineering. In *Proceedings of the Second international conference on Simulation, modeling, and programming for autonomous robots*, SIMPAR'10. Springer-Verlag, 2010.

[6] J.P. Laumond, S. Sekhavat, and F. Lamiraux. Guidelines in nonholonomic motion planning for mobile robots. *Robot motion planning and control*, 1998.

[7] E. Szádeczky-Kardoss and B. Kiss. Extension of the rapidly exploring random tree algorithm with key configurations for nonholonomic motion planning. In *Mechatronics, 2006 IEEE International Conference on*. IEEE, 2006.

[8] J. Riehl. Implementing the myfem embedded domain-specific language. In *Proceedings of the Second International Workshop on Domain-Specific Program Development*, DSPD'08, 2008.

[9] John A. Miller, Jun Han, and Maria Hybinette. Using domain specific language for modeling and simulation: Scalation as a case study. In *Winter Simulation Conference*, 2010.

[10] M.H. Hohn. A little language for modularizing numerical pde solvers. *Software: Practice and Experience*, 34(9), 2004.

[11] H. Liu and P. Hudak. An ode to arrows. *Practical Aspects of Declarative Languages*, 2010.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[13] N. Wirth. Extended backus-naur form (ebnf), 1996. *ISO/IEC*, 14977, 1996.

[14] Fuhua Han, Takaaki Yamada, Keigo Watanabe, Kazuo Kiguchi, and Kiyotaka Izumi. Construction of an omnidirectional mobile robot platform based on active dual-wheel caster mechanisms and development of a control simulator. *J. Intell. Robotics Syst.*, 29, 2000.

[15] Takaaki Yamada, Keigo Watanabe, Kazuo Kiguchi, and Kiyotaka Izumi. Dynamic model and control for a holonomic omnidirectional mobile robot. *Auton. Robots*, 11, 2001.