Lightweight Domain-Specific Modeling and Model-Driven Development

Risto Pitkänen and Tommi Mikkonen
Institute of Software Systems, Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
{risto.pitkanen, tommi.mikkonen}@tut.fi

Abstract

Domain-specific modeling (DSM), especially when accompanied with powerful tools such as code generators, can significantly increase productivity in software development. On the other hand, DSM requires a high initial investment due to the effort needed for designing a domain-specific modeling language and implementing code generators. In this paper, a lightweight DSM approach that uses somewhat more generic languages and developer-guided transformations borrowed from model-driven development is discussed. It is concluded that the lightweight approach can be used as a bridge to full-blown DSM or in a context where sufficient economies of scale do not exist to justify the investment required by the latter approach.

1 Introduction

Domain-specific modeling (DSM) is commonly advocated as a means to raise the level of abstraction in software development and to achieve higher levels of productivity than with conventional languages. DSM languages borrow their vocabulary from the problem domain, letting developers concentrate on defining the problem instead of implementation-level details. Model transformations and code generators are then used for deriving an actual implentation in a computer-assisted fashion.

Model-driven development (MDD) is a related approach where abstract models are incrementally refined through model transformations, starting with a problem domain centric model, the final goal being the production of a solution-centric platform-specific model. Mainstream MDD research is, however, tightly coupled with OMG's Model Driven Architecture (MDA), where a generic modeling language, UML, is usually utilized instead of domain-specific languages.

In this paper we investigate the relationship between DSM and MDD, more specifically the use of *somewhat* domain-specific models in a context where a highly specific language does not (yet) exist, and the introduction of concepts and constructs related to the implementation domain using refinement and transformations. We refer to this approach as *lightweight hybrid DSM/MDD.* A simple mobile robot will be used as a running example.

The structure of the rest of this paper is as follows. In Section 2 we define the scope of the discussion more precisely and compare the lightweight approach to full-blown DSM. Section 3 introduces a running example and discusses a somewhat domain-specific modeling language for real-time control systems. In Section 4 it is shown how a somewhat domain-specific high-level model can be combined with an architecture, and eventually transformed to an implementation. Section 5 concludes the paper with some discussion.

2 Lightweight Domain-Specific Modeling

Domain-specific modeling languages are usually built around a tool chain that includes automatic code generation for a certain implementation platform. Provided that DSM languages, modeling tools, and code generation tools are well designed, significant increases in productivity can often be achieved. With MetaEdit+ [4], for instance, five- to ten-fold productivity increases compared to conventional coding have been reported [6].

The tradeoffs associated with the above approach are the high initial investment required for designing a domain-specific language, and the inflexibility with regard to the target platform. The latter disadvantage is of course partly due to the use of a code generator that normally only supports a certain target platform, but also to the fact that DSM languages tend to reflect a certain target architecture. In other words, DSM languages often include concepts that do not actually originate in the problem domain, but in the solution domain. In fact, the vagueness of the term *domain* is a known issue, and at least two different points of view have been identified: "domain as the real world" and "domain as a set of systems" [11].

For example, Tolvanen mentions [10] a MetaEdit+ based DSM language whose *"modelling concepts are directly based on the services and widgets that Series 60 phones offer for application development."* This reflects the "domain as a set of systems" view and is of course a valid and efficient approach when only Series 60 SmartPhones are targeted, but consider a setting where a company wants to produce the same application for both Series 60 and Microsoft Windows Smartphone platforms. In this situation, using a somewhat more generic specification language and perhaps adopting the "domain as the real world" view would make sense.

While we do not question the efficiency and usefulness of the full-blown DSM approach in many applications, in some cases a lighter, "modular" approach might be required, where slightly more generic specifications are refined in a systematic manner towards an implementation. Such an approach might have to do with lower levels of productivity increase than actual DSM, but it might be applicable in situations where DSM is not a feasible solution or defining a full-blown DSM language is simply not realistic due to the lack of experiences on the specific domain. Our proposal is based on a method framework called *specification-driven development* [8] and our experience with using a *somewhat* domain-specific specification language, DisCo [2, 1], for high-level specification in a context where architectural styles are utilized in the process of producing a more detailed design of a system [8].

DisCo is actually a formal specification language for reactive systems. Thus, if it is viewed

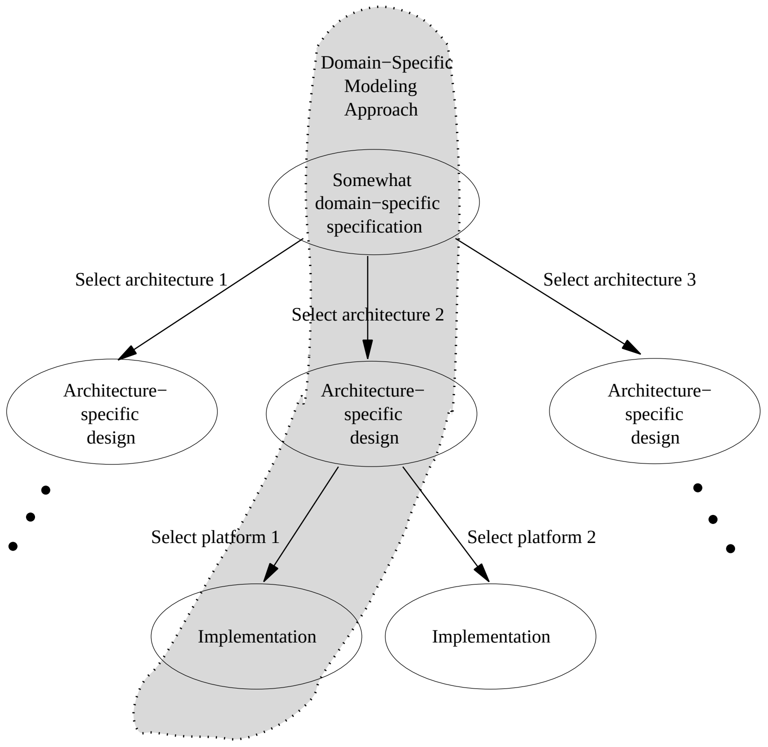
Domain−Specific Modeling Approach
Somewhat domain−specific specification
Select architecture 1
Select architecture 2
Select architecture 3
Architecture−specific design
Architecture−specific design
Architecture−specific design
Select platform 1
Select platform 2
Implementation
Implementation

Figure 1: Lightweight hybrid DSM/MDD compared with full-blown DSM.

as a domain-specific language, the problem domain is quite large: for instance, distributed enterprise business logic fits the definition of a reactive system, as does a mobile robot control application. The defining characteristic of a system whose modeling DisCo is suitable for is that it is in continuous interaction with its environment. Real-time properties may or may not be included. DisCo is not, however, as generic as UML, as the specification of a stereotypical transformational program such as a compiler using the language would not be feasible.

The proposed lightweight hybrid DSM/MDD approach and its relationship with full-blown DSM development are illustrated in Figure 1. In the lightweight approach, a somewhat domain-specific specification-level language is used for high-level modeling, and the resulting model is then transformed into an architecture-specific design model, ideally using some kind of a transformation tool. The idea is that the sort of transformations required for this step are less laborious to define than a full-blown code generation strategy for a DSM language because the transformation process is developer-guided. Furthermore, a more generic but still somewhat domain-specific language has a wider scope of applicability, and therefore it can be used in a context where defining a fully tailored language would be overkill.

In contrast, a full-blown DSM approach (whose scope covers the grey area in Figure 1) requires that architects and domain experts define a tailored specification language and specify how the different constructs are mapped into code, or even implement a code generator manually. Such a language-and-tool-chain is restricted to a particular combination of a problem domain, architecture, and implementation platform. The raise in productivity can indeed be high, but there are situations in which the approach is not applicable, for instance

- if only one or a small number of similar projects are planned, and the high initial investment required by DSM cannot be justified, or
- if precise enough estimates about performance issues cannot be done without first writing some sort of a specification, and if the results of this evaluation affect the choice of architecture and/or implementation technologies, or
- if several different implementation platforms are targeted.

In short, the lightweight hybrid DSM/MDD approach is perhaps more suitable than full-blown DSM for pilot type projects and environments where sufficient economies of scale do not exist. Additionally, the approach can be used in the initial phases, before a full-blown DSM toolchain has been implemented, to aid in defining the scope and concepts of a DSM language, and selecting the right architecture and implementation platform to commit to.

3 Example: High-Level Model of Digibot

We will illustrate lightweight hybrid DSM/MDD using a simple mobile robot as an example. The mobile robot is based on Tutebot, a simple robot introduced by Jones and Flynn [3]. While the original Tutebot was implemented using analogue electronics, our version is digital and therefore called Digibot. A more verbose discussion on the Digibot model can be found in [8], where also animation and model-checking are used for validating and verifying the specification.

Like Tutebot, Digibot wanders around a room, traveling straight ahead until it encounters a wall or some other obstacle. It has two sensor switches attached to the front bumper, one on the left and one on the right. Depending on which side touches the obstacle, Digibot shall back up in an arc towards the left or the right before starting to travel straight ahead again. Thus, the robot will travel a route that resembles that in Figure 2.

3.1 DisCo Specification of Digibot

In a full-blown domain-specific development approach a tailored modeling language with concepts related to timers, sensors, actuators, motors, etc. would probably be utilized for specifying Digibot. Ideally, a code generator would then produce a complete implementation for a certain hardware platform, such as an Atmel AVR based controller card.

In contrast, we use DisCo as a specification language that is biased towards reactive real-time systems, but does not prescribe a single implementation platform or even a general architecture – there are modeling constructs for concepts such as deadlines, minimum separation, and atomic event, but no architecture-specific constructs. The specifications can be implemented using hardware as well as software.

DisCo specifications are arranged in *layers* that are superimposed on each other. Each layer typically specifies an *aspect* of the problem, and rules of superposition guarantee that safety

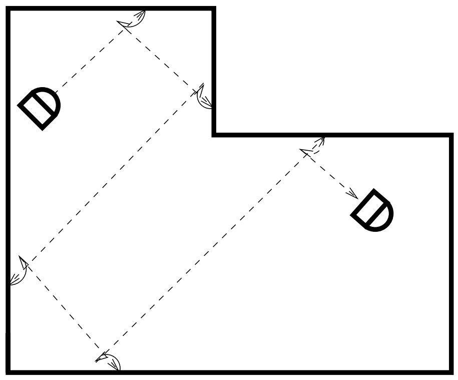

Figure 2: Route of Digibot.

```
layer abstract_robot is
  constant MIN_STOPPED: time := 0.5;

  class Robot(1) is
    mode: (STOPPED, MOVING);
    allowed_to_move_at: time := 0.0;
  end;

  action move(r: Robot) is
  when r.mode'STOPPED ∧ now ≥ r.allowed_to_move_at do
    r.mode := MOVING();
  end;

  action stop(r : Robot) is
  when r.mode'MOVING do
    r.mode := STOPPED() ||
    r.allowed_to_move_at := now + MIN_STOPPED;
  end;
end abstract_robot;
```

Figure 3: Layer abstract_robot.

properties (of the form *"something bad never happens"*) are preserved by construction. Layers can introduce *classes*, *actions*, and some other constructs, and they can refine classes and actions introduced in previous layers.

Digibot is specified in DisCo using three layers. The first layer called abstract_robot, depicted in Figure 3, specifies an abstract robot as a device that can only move or stop. The layer defines a constant time interval MIN_STOPPED that specifies the minimum duration of inactivity between periods of movement. This anticipates the capability to change direction that is to be added later on; i.e. the robot must not change the direction of rotation of its electric motors too quickly to avoid causing damage to them. The literal constant '1' in parentheses after the class name Robot indicates that there must be exactly one instance of Robot. State machine mode inside the class specifies that the robot can either be stopped or moving, and the time-valued variable allowed_to_move_at is used for implementing a minimum separation between distinct periods of movement.

Actions move and stop specify the behavioral part of the layer. The guard of action move specifies that, when stopped, the robot may begin moving when the minimum duration of inactivity has passed. The change of state is implemented using a simple assignment statement in the action body. Action stop is enabled whenever the robot is moving. The mode of execution is nondeterministic: any action whose guard evaluates to true in the current state can be exexuted. In addition to the change of state, the action body stores to the variable allowed_to_move_at the earliest

```
layer directional_robot is
  import abstract_robot;

  extend Robot by
    extend MOVING by
      dir : ( FORWARD, BACKUP_LEFT,
              BACKUP_RIGHT);
    end;
  end;

  refined forward(r : Robot) of move(r) is
  when ... do
    ...
    r.mode'MOVING.dir := FORWARD();
  end;

  refined backup_left(r : Robot) of move(r) is
  when ... do
    ...
    r.mode'MOVING.dir := BACKUP_LEFT();
  end;

  refined backup_right(r : Robot) of move(r) is
  when ... do
    ...
    r.mode'MOVING.dir := BACKUP_RIGHT();
  end;
end directional_robot;
```

Figure 4: Layer directional_robot.

instant of time at which the robot is again allowed to move.

Layer directional_robot (Figure 4) adds the notion of direction of travel to the abstract specification. When moving, Digibot is assumed to be heading forward, backing up towards the left, or backing up towards the right. This is specified by extending state MOVING by three sub-states that indicate whether the robot is moving forward, backing up towards the left, or backing up towards the right. In addition, three refinements of action move are given. They correspond to the three different directions of movement, and activate the appropriate sub-state. Note that the base action move ensures that each of these versions must respect the real-time constraint discussed above. After the refinements, the pure move action will cease to exist.

This far, Digibot is just a nondeterministic entity that can execute any sequence of stopping and moving actions as long as the sole real-time constraint is honored. To add the actual control part to the robot, layer robot_with_sensors is given in Figure 5. The time constants BACKUP_DL and STOP_BACKUP_DL define maximum time intervals for starting to back up after running into and obstacle, and stopping the backup phase. These constants are used in actions. Class Robot is extend with a state machine indicating backup mode, i.e. whether Digibot is about to back up in either direction. The contact actions modeling the closing of sensor switches are refinements of stop, i.e. sensor switch contact triggers an immediate stop. The actions modeling backing up are enabled in the corresponding backup mode, and they reset the backup_mode state machine to state NO in addition to removing the deadline for starting the backup phase and setting a deadline for stopping the backup phase. The specification also needs an ordinary stop action that brings the robot to a halt after backing up. The guard of action forward is refined to require that the robot is not currently in either of the backup modes.

Digibot is of course a very simple and small example, and does not demonstrate the full power of DisCo as a modeling language. For instance, timed automata could well be used for specifying the same system while still keeping it manageable. The advantages of layerwise refinement and adding one aspect at a time start to show in larger systems and systems that include unbounded ranges and structures or dynamic creation of objects.

```
layer robot_with_sensors is
  import directional_robot;

  constant BACKUP_DL: time := 1.0;
  constant STOP_BACKUP_DL: time := 3.0;

  extend Robot by
    backup_mode: (NO, LEFT, RIGHT);
    halt_dl : time;
    backup_dl: time;
  end;

  refined contact_left (r : Robot) of stop(r ) is
  when ... r.mode'MOVING.dir'FORWARD do
    ...
    r.backup_mode := LEFT() ||
    r.backup_dl @ BACKUP_DL;
  end;

  refined contact_right(r : Robot) of stop(r ) is
  when ... r.mode'MOVING.dir'FORWARD do
    ...
    r.backup_mode := RIGHT() ||
    r.backup_dl @ BACKUP_DL;
  end;

  refined backup_left(r : Robot) of
          backup_left(r ) is
  when ... r.backup_mode'LEFT do
    ...
    r.backup_mode := NO() ||
    r.backup_dl @ ||
    r.halt_dl @ STOP_BACKUP_DL;
  end;
  refined backup_right(r : Robot) of
          backup_right(r) is
  when ... r.backup_mode'RIGHT do
    ...
    r.backup_mode := NO() ||
    r.backup_dl @ ||
    r.halt_dl @ STOP_BACKUP_DL;
  end;

  refined stop(r : Robot) of stop(r ) is
  when ... now ≥ r.halt_dl do
    ...
    r.halt_dl @;
  end;

  refined forward(r : Robot) of
          forward(r ) is
  when ... r.backup_mode'NO do
    ...
  end;
end robot_with_sensors;
```

Figure 5: Layer robot_with_sensors.

4 Adding an Architecture to Digibot

In our approach, combining a specification and an architecture results in a *design*. A design model is a solution domain oriented description of the structure and processing of a particular system. To be complete, it has to take into account both behavioral and structural aspects. Application-specific data and behavior is imported from the specification, while a suitable architectural style is used as the source of more generic structural and behavioral elements.

We shall discuss two different architectural styles that can be applied to the Digibot specification. They are a hardware-based style, and an interrupt-driven software based style.

4.1 Hardware architecture for Digibot

An architectural style for hardware implementation of DisCo specifications (further discussed in [5]) is best described in terms of certain elements. The elements are: combinatorial block for computing the guard of an action, scheduler, sequential block corresponding to an object, and sequential block corresponding to a subsystem. A generic architecture resulting from such components is depicted in Figure 6.

The operating principle of a system based on the architecture is the following: the combinatorial blocks $G_{A_i}, i \in [1, n]$ constantly evaluate the value of each action guard. The results are input to a synchronously operating scheduler that selects one enabled action per each clock cycle

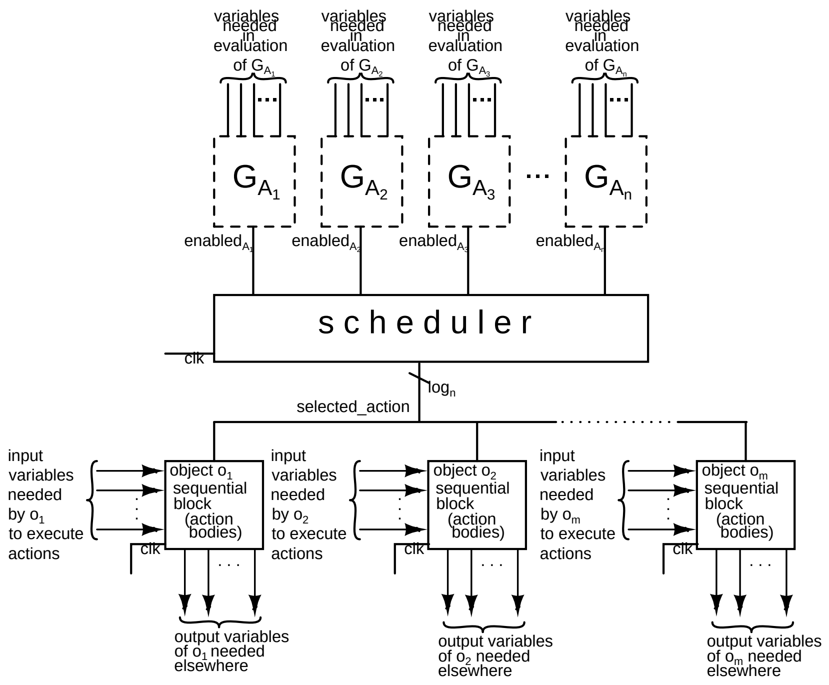
variables needed in evaluation of G_{A_1}
variables needed in evaluation of G_{A_2}
variables needed in evaluation of G_{A_3}
variables needed in evaluation of G_{A_n}
G_{A_1}
G_{A_2}
G_{A_3}
G_{A_n}
$enabled_{A_1}$
$enabled_{A_2}$
$enabled_{A_3}$
$enabled_{A_n}$
scheduler
clk
log_n
selected_action
input variables needed by o_1 to execute actions
object o_1 sequential block (action bodies)
input variables needed by o_2 to execute actions
object o_2 sequential block (action bodies)
input variables needed by o_m to execute actions
object o_m sequential block (action bodies)
clk
output variables of o_1 needed elsewhere
output variables of o_2 needed elsewhere
output variables of o_m needed elsewhere

Figure 6: Hardware style.

according to some scheduling strategy. The selected action is then communicated to the object blocks using a bus that can be implemented e.g. using $log\ n$ one-bit signals. Each synchronously operating object block then executes its share of the action body, and the cycle is then repeated from the beginning.

Each guard of a basic action maps to a combinatorial network (shown in the top part of Figure 6) that takes as its input signals corresponding to all the variables that the action guard depends on, and outputs a single bit indicating whether the action is currently enabled or not. This mapping is usually straightforward, as it in effect means the transformation of a logical formula to a logic network.

The outputs of the combinatorial networks are input to a scheduler component that picks the next action to be executed. The scheduler operates synchronously, triggered by a an edge of a clock signal. The scheduler might also have additional inputs that are used in the computation (for example, some of the signals representing variables of the system), and it can utilize many different kinds of scheduling strategies. Finally, the scheduled action and all needed input variables are communicated to the sequential blocks that correspond to objects that synchronously execute action bodies. Each object block is responsible for executing those parts of the body that result in state changes in its own variables. The object blocks must also output the values of those variables that are needed elsewhere.

Transforming a DisCo specification to such a hardware-based design can either be carried out manually or with the aid of an experimental compiler [7]. The approach is discussed further in [9], [5], and [7].

4.2 Interrupt-Driven Architecture for Digibot

The *interrupt-driven* style is a software-based architectural style suitable for the implementation of real-time control applications. It is widely used for small-scale embedded systems, and especially useful if there is no operating system that supports processes or threads. The architectural style is very simple and can be described as follows: *1)* There is a main loop that is often empty or performs some background tasks. *2)* Interrupts are generated by timers and input/output events. *3)* When an interrupt occurs, control is transferred to an interrupt handler for that particular interrupt. *4)* The actual application functionality resides in the interrupt handlers.

The interrupt-driven style can be used for implementing relatively simple DisCo specifications. It is particularly useful for specifications that contain real-time aspects, and where actions are effectively triggered by the passing of time or by events that are conceptually controlled by the environment. Examples of events of the latter kind are inputs coming from sensors.

When applying the interrupt-driven style to a DisCo specification, one first needs to identify the variables, events, and timing constraints that are used to trigger interrupts, and determine which actions are executed on which interrupt. An action may model an external event that generates an interrupt; for example the closing of a sensor switch. Potentially nondeterministic timing constraints often map into deterministic timer interrupts: for instance, if action A sets both a minimum separation and a deadline for action B, this can be realized by setting a timer that expires somewhere in between these time instances and causes an interrupt. Actions that are not interrupt-triggered can be placed in the background loop of the program, their guards implemented using conditional statements. However, often in control-oriented specifications such actions do not exist.

5 Conclusion

Lightweight hybrid DSM/MDD is an approach where a somewhat domain-specific specification languages are used in conjunction with transformation techniques and tools that enable computer-assisted implementation of specifications. Compared to a full-blown DSM approach based e.g. on a tool such as MetaEdit+ [4], the lightweight method requires a smaller initial investment and offers flexibility with regard to choosing an architecture and a target platform. On the other hand, the process of deriving an implementation based on a specification requires more developer intervention, and thus the productivity increase is probably lower.

Lightweight hybrid DSM/MDD can be used as a bridge from generic methods and tools to a domain-specific workflow, as it allows incremental development of the modeling languages, model compilers and transformators that are required for a full-blown DSM toolchain. A specialized DSM language and automatic code generators can be based on experience and partial transformators that have been developed when applying the lightweight approach.

References

[1] DisCo WWW site. At `http://disco.cs.tut.fi` on the World Wide Web.

[2] H.-M. Järvinen and R. Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[3] J. L. Jones and A. M. Flynn. *Mobile Robots: Inspiration and Implementation*. A K Peters Ltd, 1993.

[4] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *CAiSE ;96: Proceedings of the 8th International Conference on Advances Information System Engineering*, pages 1–21, London, UK, 1996. Springer-Verlag.

[5] H. Klapuri. *Hardware-Software Codesign with Action Systems*. PhD thesis, Tampere University of Technology, 2002.

[6] The MetaCase website. http://www.metacase.com.

[7] J. Nykänen, H. Klapuri, and J. Takala. Mapping action systems to hardware descriptions. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03), Las Vegas, Nevada, USA*, pages 1407–1412. CSREA Press, June 2003.

[8] R. Pitkänen. *Tools and Techniques for Specification-Driven Software Development*. PhD thesis, Tampere University of Technology, 2006.

[9] R. Pitkänen and H. Klapuri. Incremental cospecification using objects and joint actions. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2961–2967, Las Vegas, Nevada, USA, June 1999. CSREA Press.

[10] J.-P. Tolvanen. Making model-based code generation work - practical examples. *Embedded Systems Europe*, pages 38–41, March 2005.

[11] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.