How Intelligent Functionality Implemented in Domain Specific Meta-Models Depends on Model Semantics

Peter Krall, pk@cortex-brainware.de

Cortex Brainware Consulting & Training GmbH, Kirchplatz 5, Pullach, D-82049 Germany,
http://www.cortex-brainware.de

Abstract. Model driven development may be based on different model semantics. This paper compares two alternatives.
The first of these alternatives conceives of models as representations of computational systems on different levels of abstraction. According to this approach, the development is a process of refinement and completion. An alternative approach uses domain specific meta-models to shift perspective from software design towards an analysis of the environment where the software is to be used. This approach combines concepts from logic programming and domain specific development. Model elements describe aspects of the environment directly, not indirectly by definition of their representation in a computational system. However, the use of models with declarative semantics is restricted to a representation of specific aspects of scenarios that vary within the domain. This fragmentary declarative information is evaluated by an intelligent functionality of a domain-specific meta-model for instantiation of a domain-specific application template.

Key words: Domain-specific modeling, semantics, artificial intelligence

1 Model Semantics

Model driven development may be seen as a paradigm that lifts the level of abstraction in programming [3], [6], [12]. An alternative approach aims at lifting the level of abstraction beyond programming [16]. According to the first approach, the development environment supports software engineering. The second approach generates software from declarative characterizations of facts from the environment where the software is to be used. The functionality for generating software from a model may therefore be based on evaluating either of two kinds of semantics for model elements: computational system oriented or analytic semantics.

1.1 Computational system oriented semantics

Computational system oriented semantics, in this context, means any kind of characterization of the computational systems that will act as an information processing system. This includes interpretation of model elements as instructions for a machine, either referring to a translation into a low-level machine language or using the constructions of an abstract state machine that accepts constructs of object oriented languages as instructions [1], [7]. It also includes interpretations of model elements as characterizations of the computational system that are not instructions in this system but may be evaluated by functionality in the model driven development process. Specification of the database type to be used exemplifies the latter type of model semantics: this information is not an instruction executed at runtime but nevertheless part of the computational system's characterization and a natural candidate for evaluation by a code-generator using it when generating instructions.

Computational semantic oriented techniques may use general purpose or domain specific languages. A domain specific language with computational semantics in this sense is *a language that enables the specification of software from a specific viewpoint* [6]

Abstracting from implementation details will lead to a more abstract view on the computational system but not change the perspective to something external to the computational system. Thus lifting the level of abstraction in computational system oriented semantics will yield computational system oriented semantics again, albeit on a higher level of abstraction. Technically the relation between semantics of the more abstract and more concrete view is a homomorphism.

1.2 Analytic semantics

Analytic model semantics correspond to interpretation of the model elements as representations of aspects of the environment where the solution is to be used. This is not a more abstract view of the computational system but a view on something external to the computational system. This view is expressed by means of the language defined in the meta-model, which will often be object oriented or based on other more general formal constructs than those of traditional formal languages [5], [9].

Analytic semantics may be introduced by defining a mapping from model elements into the set of syntactic formatives of a formal language - for example, a language used for serialization of an object oriented model. Even though the modeling language is defined as a type system in an object oriented meta-model this allows conceiving of analytic semantics based on the theory of semantics of formal languages [2],[8],[13],[15]. Following the concepts of classical logics, the formal structure of analytic semantics may be seen as a mapping from the model language into a lattice. Invariant aspects of the domain correspond to a filter in the lattice. Fragmentary analytic models contain complementary information that can neither be derived nor rejected on the basis of the domain-wide valid assertions. For example ‘all contracts cover liability’ may be part of the description of the domain ‘liability insurance’, the statements ‘all drivers own a license for at least 1 year’ versus ‘all drivers with valid license may be insured’ may be elements of two models, either characterizing the scenario ‘experienced driver insurance’ or ‘universal driver insurance’ in the domain.

1.3 Perspectives

The two types of semantics correspond to two perspectives for domain specific models: the first alternative observes that environments of a domain vary with respect to some aspects but may have a lot in common. Models are used to describe what is specific for a particular environment. The second alternative observes that there is variation within a software family while other aspects are invariant. Models describe the specifics of software solutions or parts thereof.

The two perspectives and the associated semantics are orthogonal, not mutually exclusive. This can be demonstrated by a simple thought-experiment, starting with the assumption of one language with purely declarative semantics and another language with purely computational system oriented semantics. A meta-model can then define model elements that contain expressions from one language with annotations from the other one, yielding model semantics with both aspects. There are thus models which have purely computational system oriented, purely analytic or integrated semantics.

Models with purely computational system oriented semantics address technical tasks that can be accomplished without any knowledge about the meaning of the computational system’s states or processes in external world.

Purely analytic modeling approach aims at obtaining an information processing system by feeding information about the environment into a machine that will generate software for a computational system matching the requirements of the scenario. This strategy does not aim at seamless integration of analysis, design and programming but at strict separation. It does not only abstract from implementation detail but avoids reference to software design as well.

Conversely, object oriented philosophy aims at seamless integration of analysis, design and programming. This implies a different usage of models driving development, compared with the purely analytic approach: in the former case the model will ideally support the work of analyst-programmers who build an information processing system, understanding what it is good for and how it works, whereas in the other case the model ideally is built by domain experts without any knowledge about the software system that will be generated from the model.

1.4 The application template

Often families of software systems are variants of a common core or fixed part. The invariant part itself may take various forms: *The fixed part captures the architectural patterns that make up the domain and exposes extension points that enable it to be used in a variety of solutions. ... this fixed part may be called a framework, a platform, an interpreter or an Application Programming Interface* [3]. The family of applications may be conceived as instances of a template. The template in this notion corresponds to the totality of constructs defining the fixed part while every instantiation corresponds to a possible use of the extension points.

Corresponding to the various forms the application template may take, there also are various alternative mechanisms for the instantiation of an application template, for example by generating code for specializations of domain-wide usable base classes from the model, embedding the generated classes into a non-generated framework or generating partial class code.

The usage of application templates that are instantiated for particular purposes is the common base of the two approaches to domain specific development discussed here.

2 Functionality for template instantiation

Since the two concepts under discussion differ with respect to the semantics of the models used, a different kind of functionality has to be implemented for evaluating the information represented in the models.

2.1 Functionality for design models

According to the design oriented approach the mapping between analytic and design aspects is done by the modeler. Model elements do not directly represent things in the external world but rather their representation in the computational system. The mechanisms implemented in the meta-model will not generate a type system from declarative information about the external world but support the modelers' work on the type system designed by them for the purpose.

There is a wide variety of tasks that may be supported by tools in software development. Examples include coordinated production of variants of several classes for instantiation of general or domain-specific design patterns. Another class of issues concerns the generation of particular functionality for tasks like communication between processes or persistence. More powerful mechanisms than the universally applicable concepts of inheritance or generics can be implemented for specific domains using domain-specific languages and generic functionality [3],[6].

2.2 Functionality for analytic models

Analytic semantics of model elements means their semantics in the external world. The origins of the idea to shift perspective from software design to the analytic description of the environment are based on universal meta-models, based on the observation that formulas of formal languages may be given a declarative as well as a procedural interpretation [10],[11],[14]. However, since nothing except the applicability of the rules of logics may be presupposed for the whole world, procedural semantics in the universal meta-model reduces to drawing logical conclusions. This yields the necessity for explicit representation of all non-tautological information, which is not always feasible. Despite the elegance of the concepts of logic programming the majority of information processing systems is still built by programming, not by feeding logical inference machines with sufficiently comprehensive information (Figure 1). Domain specific

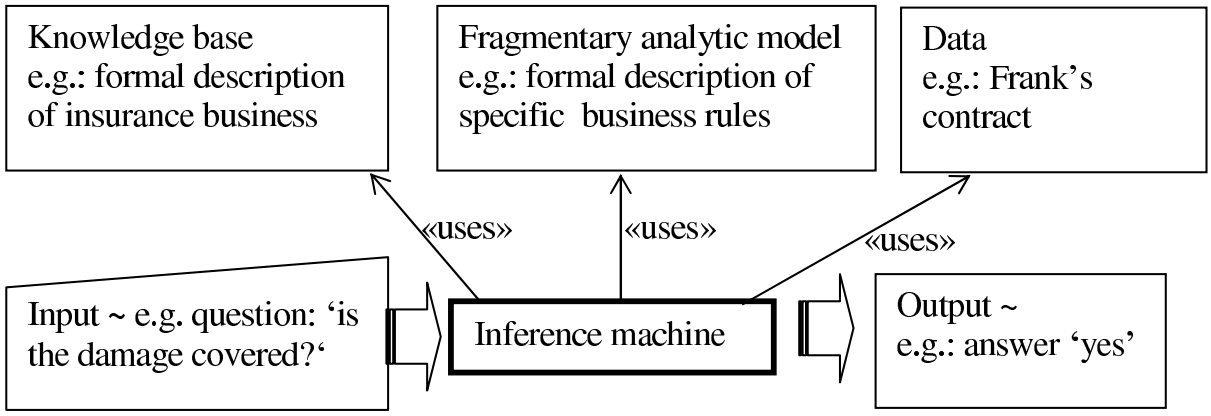

Fig. 1. The logical inference machine uses information from a domain-wide knowledge base, a fragmentary analytic model and additional data. In the example the knowledge base describes basic facts about insurance business, the analytic model describes specific business rules and the data describes the state of Frank's contract. From this information the system derives the answer to the question whether a reported damage is covered. The information must allow the answer to be obtained by pure logic reasoning.

meta-models make substitution of programming possible without a comprehensive knowledge base because restriction to particular domains allows adopting an intentional stance [4]. The ability of software engineers with domain expertise to produce variants of a known application template from fragmentary analytic information illustrates this principle: This is accomplished by finding some template for programmatic solution capturing some kind of situation and using the information about the environment to instantiate the template for this particular case. For example, human software developers may be able to adapt a scheme for insurance contract administration programs to cope with the fact that legal persons merge or split, despite the unavailability of a complete theory of insurance business and a formalized framework that would allow them to characterize persons as entities that exist in time and may share their identity for a part of the lifetime. This type of domain specific intelligence for intentional interpretation of fragmentary information may be implemented in functionality defined in domain-specific meta-models (Figure2). The domain specific intelligence for evaluation of model information will work in a way that is very similar to that of human software engineers with domain expertise: it implements a programmatic solution for instantation of an application template from highly

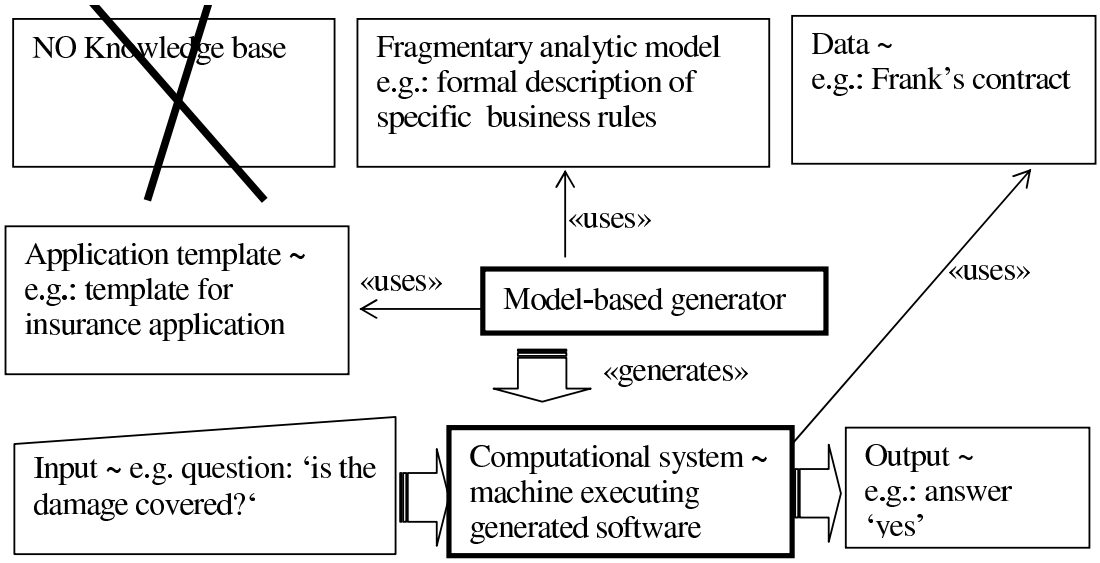

Fig. 2. The generator uses a domain-specific application template and a fragmentary analytic model to generate software for a computational system. The system executing the generated software uses data to generate output as answer to given input. There is no comprehensive knowledge base. The correctness of behavior must be guaranteed by the application template and the programmatic rule for instantiation of this template from fragmentary analytic models.

fragmentary analytic information about the environment. There is no such thing as a knowledge base that would be completed to a comprehensive formal theory of the environmental scenario. The meta-model defining this kind of intelligent functionality must be domain-specific since the intentional interpretation is based on non-tautological presuppositions concerning the use that shall be made of the information.

2.3 Perspectives

The two types of functionality for evaluation of model information correspond to the perspectives mentioned in the paragraph on semantics. The characterization of facts in the environment that defines the analytic view is completed by functionality for instantiation of an application template that matches the requirements in the specific environment. The characterization of specifics of a solution variant within a family that corresponds to the design perspective is completed by functionality for instantiation of an application template with the specific characteristics defined in the model.

For models representing both, analytic and design aspects, the corresponding functionalities for evaluation of model information may also be combined, though it may be remarked that this will yield an increase in complexity.

3 Example: Chess variant playgrounds

This section illustrates the principles of application template instantiation from analytic models. The target domain is: 'playgrounds for chess variants'. Elements of the domain are games played on a chess-board. These games differ with respect to sets of piece types, rules for moving the pieces and initial position. The application to be generated for every game from the domain

shall allow two players to play the game on virtual computer boards presenting the current situation on two windows, allowing entering moves, checking input for conformity with rules and updating the position.

3.1 Model element type system

The meta-model itself has been constructed as an object oriented application. Every model consists of instances of three model element classes:

- Class Graph (The content of a Graph is a collection of pairs of points, where every point is a pair of two non-negative integers x < 8, y < 8. Graphs may be constructed from other graphs by use of set-operations.
- Class PieceType (A piece type has a label and a set of references to Graph-objects. The references to Graph-objects semantically define what moves are allowed for the pieces of the respective type, i.e.: it may move along the edges in any of the associated Graph-objects.
- Class InitialPosition (Characterizes the initial position on the board.

The simplest way to describe the semantics of this object-oriented meta-model is by reference to a formal language that is used by the models for serialization. Assuming that chess is a sufficiently well-known domain, this language and its semantics will only be indicated by an example:

```
Graph HAdjacency { <<0,0>,<1,0>>,<<1,0>,<0,0>>,<<1,0>,<2,0>>,.... }
Graph VAdjacency { <<0,0>,<0,1>>,<<0,1>,<0,0>>,<<1,0>,<1,1>>,.... }
PieceType Rook { Uses(HAdjacency ,Iterate=True,Jump=false,MayTake=True
               , Uses(VAdjacency ,Iterate=True,Jump=false,MayTake=True)}
```

The meaning of the model elements that serialize in form of the above formulae is: There is a collection of edges connecting fields on the board named ‘HAdjacency’, containing edges from <0,0> to <1,0>, <1,0> to <2,0> etc. Rooks may move along horizontal or vertical adjacencies.

3.2 Application template

The application template in the example consists of some non-generated partial classes. The core class is ‘Board’. The board contains a two-dimensional matrix of Enum-values, representing the current position:

```
partial class Board { Enum [][] fields; }
```

Next, Board declares a delegate ‘PossibleMove’. Every PossibleMove instance will represent one rule for an allowed transition.

```
partial class Board { delegate bool PossibleMove(int x, int y, int xn, int yn
          ,Enum piece, Enum[][] fields); }
```

Board also has a method for checking rule conformity of a move entered by a player. This method uses reflection for retrieving all properties of PossibleMove type and, for each of these properties, invokes the delegate that is the property’s value. A move is accepted if it is accepted according to the rule represented by any PossibleMove property.

```
bool AllowsMove(int x,int y, int xn,int y,Enum piece) {
foreach (PropertyInfo property in GetType().GetProperties()) {
   if (property.PropertyType == typeof(PossibleMove)) {
     PossibleMove move = (PossibleMove)(property.GetValue(this, null));
     if (move(x,y,xn,yn,piece,fields)){return true;}}
} return false /* no applicable rule found */;}
```

3.3 Template instantiation

Template instantiation is done by a code-generating functionality implemented in the model element classes. For the rule conformity check this code will be generated by PieceType objects, using the Graph-objects they refer to. Every PieceType will generate a PossibleMove property of Board for every edge in every referenced Graph. A typical rule property generated by 'Rook' for standard chess will look like this:

```
partial class Board { public PossibleMove Move4RookFrom1_1To1_3 { get {
 return delegate(int x, int y,int xn,int yn,Enum piece,Enum[][] fields {
  return (x== 1)&&(y==1)&&(xn==1)&&(yn==3)&&((PieceType)piece==PieceType.Rook)
      && ((PieceType)fields[x][y]==PieceType.Rook)
      && ((PieceType)fields[1][2]==PieceType.Empty); }; }}}
```

Invocation of this delegate will confirm rule-conformity for moving a rook from field <1,1> to <1,3>, provided there is a rook on <1,1> initially and the path is not blocked at <1,2>.

The presented fragment of a domain specific meta-model for instantiation of application templates from analytic models allows some observations:

- The semantics of the analytic models is a purely declarative characterization of the rules for games.
- The relation between model elements and generated elements of the solution does not preserve any structure in the sense of abstraction-refinement.
- There is no explicit declarative model of games in general or use-cases for a playground. Principal aspects of the functionality are captured by the application template.

3.4 The computational system oriented alternative

The same application template could also be combined with domain-specific models for the variant parts of the software defining a computational system. This will mean to define the template parameters, i.e. the properties of the possible-move delegate type, directly. The domain-specific language will hide the technical aspects and allow to define rules by constraints describing relations between parameters passed to the rule candidate. Unlike the analytic model, the computational system oriented model can refer to these parameters because checking rule-conformity by methods with a particular signature is part of the software design, whereas it is not part of the analysis of the game rules as such.

The difference between design model and analytic model may be underlined by varying the application template: Games could also be defined by formal grammars for notations of allowed sequences of moves – corresponding to treatment of 'chess' as a game defined by the sequences of strings denoting a regular chess game in standard notation. The change of the application

template would not have any impact on the analytic model's language or declarative semantics. The algorithm for generating template parameters will change of course but not the modeling language, which is why the description of rules in the analytic model must not depend on usage of either the original or the new method for checking rule conformity of moves. In contrast the different template will yield a change of the domain towards definition of a formal grammar for the approach working with models of the computational system.

4 Conclusion

The type of intelligence implemented in meta-models for model-driven development depends on the semantics of models used for driving development.

Functionality based on computational system oriented semantics can assist software engineers in completion and refinement of software, instantiation of design patterns, or adaptation to platforms. This functionality lifts the level of abstraction in programming but does not allow substitution of programming by analysis.

Functionality based on evaluation of analytic semantics may lift the level of abstraction beyond programming by generating a computational system from a description of the environment where the computational system is supposed to provide means for processing information. Domain specific meta-models may provide this functionality by evaluating fragmentary analytic models and instantiating application templates from the fragmentary information. This makes substitution of programming by analytic modeling feasible for domains where universal meta-models would require unmanageable heavyweight models.

References

1. Boerger, E. & Schulte, W.: *A Programmer Friendly Modular Definition of the Semantics of Java.* In: Alves-Foss, J (ed) *Formal Syntax and Semantics of Java.* Springer LNCS 1523, 1998.
2. Chang C.C, Keisler H.J. *Model Theory.* North Holland 1973.
3. Cook S., Gareth J., Kent S., Wills A.C., *Domain-Specific Development with Visual Studio DSL Tools.* ISBN 978-0-321-39820-8. Addison Wesley 2007
4. Dennett, D.: *The intentional stance.* MIT press, ISBN 978-0-262-04093-8, 1987
5. Feilkas, M. *How to represent Models, Languages and Transformations.* In Gray, J, Tolvanen, J.-P., Sprinkle, J: 6^{th} OOPSLA-Workshop on Domain-Specific Modeling (p.169-176). Jväskylä 2006. ISBN 951-39-2631-1.
6. Greenfield J. Short K., Cook S., Kent S. Crup J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* ISBN: 0471202843 Wiley, 2004
7. Gurevich, Y. *Evolving Algebras 1993: Lipari Guide* in Boerger, E. (ed) *Specification and Validation Methods* 231-243. Oxford University Press, 1995
8. Harrison, M. *Introduction to Formal Language Theory*, Addison Wesley, 1978.
9. Krall, P.: *Model integration in domains requiring user-model interaction and continuous adaptation of meta-model.* In Gray, J, Tolvanen, J.-P., Sprinkle, J: 6^{th} OOPSLA-Workshop on Domain-Specific Modeling (p.177-184). Jväskylä 2006. ISBN 951-39-2631-1. `http://www.dsmforum.org/events/DSM06/papers.html`
10. Lloyd, J.W.: *Foundations of Logic Programming.* Springer 1984, 1987. ISBN:3-540-18199-7
11. McCarthy, J. *Programs with common sense. Mechanization of thought processes,* Vol. I. London: Her Majesty's Stationery Office, 1959.
12. Architectur Board ORMSC; Miller, Joaquin & Mukerji, Jishmu (eds): *Model Driven Architecture.* http://www.omg.org/docs/ormsc/01-07-01.pdf, 2001.
13. Montague, R. *Universal Grammar.* Theoria 36: 373-398. 1970.
14. Newell, A., Shaw, J. C. and Simon, H.A. *Empirical Explorations of the Logic Theory Machine. A case Study in Heuristic.* Proceedings of the Western Joint Computer Conference, published by the Institute of Radio Engineers, New York, 1957, pp. 218-230
15. Rasiova H., Sikorsky R. *The Mathematics of Metamathematics.*Polish Scientific Publishers 1963.
16. Tolvanen, J.-P. *Domain-Specific Modeling for Full Code Generation.* Software Developer's Journal 2006. `http://en.sdjournal.org/products/articleInfo/86`