# Partial Model Completion in Model Driven Engineering using Constraint Logic Programming

Sagar Sen[1,2], Benoit Baudry[1], Doina Precup[2]

[1] IRISA, Campus de Beaulieu
35042 Rennes Cedex France
[2] School of Computer Science, McGill University,
Montreal, Quebec, Canada
{ssen,bbaudry}@irisa.fr , dprecup@cs.mcgill.ca

**Abstract.** In *Model Driven Engineering* a model is a graph of objects that conforms to a meta-model and a set of constraints. The meta-model and the constraints declaratively restrict models to a valid set. Models are used to represent the state and behaviour of software systems. They are specified in visual modelling environments or automatically synthesized for program testing. In such applications, a modeller is interested in specifying a partial model or a set of partial models which has a structure and associated properties that interests him/her. Completing a partial model manually can be an extremely tedious or an undecidable task since the modeller has to satisfy tightly-coupled and arbitrary constraints. We identify this to be a problem and present a methodology to solve (if a solution can be found within certain time bounds) it using *constraint logic programming*. We present a transformation from a partial model, its meta-model, and additional constraints to a constraint logic program. We solve/query the CLP to obtain value assignments for undefined properties in the partial model. We then complete the partial model using the value assignments for the rest of the properties.

## 1 Introduction

Model-driven engineering (MDE) [1] [2] [3] is an emerging trend that promises decrease in development time of complex software systems. For this, MDE makes extensive use of models. A model is specified in a modelling language which is described declaratively using a meta-model and a set of constraints. The Object Management Group (OMG) [4] standard for specifying a meta-model is Meta-object Facility (MOF) [5]. The constraints on the properties described in the meta-model are specified in a constraint language such as Object Constraint Language (OCL) [6].

Several tasks in MDE require building and manipulating models. Models are built in visual modelling environments [7] [8]. Models are automatically synthesized for testing [9] and verification [10]. The design space of models are explored for meeting requirements in embedded systems [11]. In all these cases, a modeller would like to specify a partial model that interests him/her. The task of completing a model manually could be extremely tedious since the modeller will have to satisfy a set of restrictions that are imposed by the meta-model and constraints (for instance, completing models in a visual modelling environment). We intend to automate this process.

In this paper we propose an approach where we use the idea that a partial model, its meta-model, and a set of constraints together result in a large set of constraints that can be represented as a *constraint logic program* (CLP). The CLP is solved/queried to obtain the missing elements in a partial model to give a complete model. We use the simple Hierarchical Finite State Machine (HFSM) modelling language to illustrate our approach. However, our approach is extensible to any modelling language. In Section 2 we describe meta-models, constraints on it and the resulting modelling language for specifying models. Section 3 presents the algorithm for transforming a partial model with its meta-model and constraints to a constraint logic program. In Section 4 we present an example based on the Hierarchical Finite State Machine modelling language. We conclude in Section 5.

## 2 Meta-models and Constraints of a Modelling Language

In this section we define the concept of a meta-model and constraints on it to specify a modelling language. We present the Hierarchical Finite State Machine (HFSM) modelling language with its meta-model and constraints as an illustrative example. The basic principles for specifying any modelling language follow the same set of procedures we use to discuss the HFSM modelling language.

### 2.1 Meta-models

A modelling language allows us to specify models. A model consists of objects and relationships between them. The *meta-model* of the modelling language specifies the types of all the objects and their possible inter-relationships. The type of an object is referred to as a *class*. The meta-model for the HFSM modelling language is presented in Figure 1 (a). The classes in the meta-model are **HFSM**, **Transition**, **AbstractState**, **State**, and **Composite**.

In this paper we use the Object Management Group (OMG) [4] standard Meta-object Facility (MOF) [5] for specifying a meta-model. MOF is a modelling language capable of specifying domain-specific modelling languages and its own meta-model. This property of MOF is known as *bootstrapping*. We use the visual language notation in MOF to specify the meta-model for the HFSM modelling language in Figure 1 (a).

Each class in the meta-model has *properties*. A property is either an *attribute* or a *reference*. An attribute is of primitive type which is either Integer, String, or Boolean. For instance, the attributes of the class **State** are isInitial and isFinal both of which are of primitive type Boolean. An example domain of values for the primitive attributes is given in Table 1. The String variable can be a finite set of strings or a regular expression that specifies a set of strings.

Describing the state of a class of objects with only primitive attributes is not sufficient in many cases. Modelling many real-world systems elicits the need to model complex relationships such as modelling that an object contains another set of objects or an object is related to another finite set of objects. This set of related objects is constrained by a *multiplicity*. A reference property of a class allows its objects to be related to a set of objects. For instance, an object of class **HFSM** contains **Transition** objects.

**Table 1.** Domains for Primitive Datatypes

| Type | Domain |
|---|---|
| Boolean | $\{true\}, \{false\}$ |
| Integer | $\{MinInt, .., -2\}, \{-1\}, \{0\}, \{1\}, \{2, ..MaxInt\}$ |
| String | $\{null\}, \{""\}, \{". + "\}$ |

The multiplicity constraint for the relationship is shown at the relationship ends in Figure 1 (a). One **HFSM** object can contain * or arbitrary number of **Transition** objects. The reference names for either class are given at opposite relationship ends. The reference hfsmTransitions is a property of class **Transition** while reference transitions is a property of class **HFSM**.

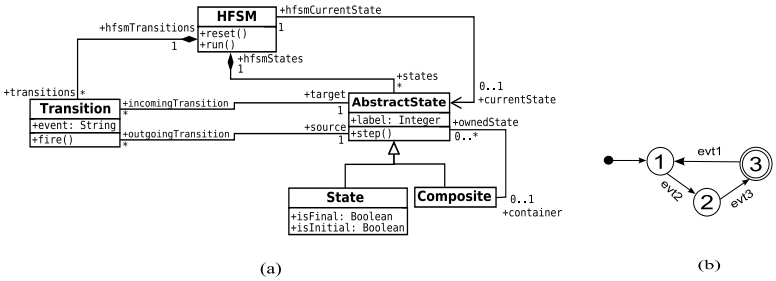


**Fig. 1.** (a) The Hierarchical Finite State Machine Meta-model (b) A Hierarchical Finite State Machine model in its Concrete Visual Syntax

A special type of relationship is that of *containment*. For instance, in Figure 1 (a) the **HFSM** class is the container for both **AbstractState** and **Transition** objects. The distinguishing visual syntax in comparison to other types of relationships is black rhombus at the container class end of the relationship.

Objects can inherit properties from other classes. The attributes and references of a class called a *super class* are inherited by derived classes. For instance, the classes **State** and **Composite** inherit properties from **AbstractState**. The visual language syntax to represent an inheritance relationship is distinguished by a triangle at the super class end of the relationship.

The visual language of MOF to specify a meta-model is expressive enough to specify a network of objects along with the domain of each of its properties. The multiplicity constraints impose constraints on the number of possible relationships between objects. However, most complex constraints that describe relationships between properties can easily and concisely be specified using a textual language for modelling constraints. In the following section we present examples of such constraints and talk about a language to specify them.

In our implementation the input meta-model for a modelling language is read in from a XML file containing the graph with the representation of the classes and the relationships between them. The cardinality constraints are generated and stored separately as Prolog clauses.

### 2.2 Constraints on Meta-models

Classes and the domain of its properties in a meta-model specify basic domain and multiplicity constraints on a model. However, some constraints that cannot be expressed directly within the meta-model are better specified in a textual language. For example, it is not possible to use multiplicities to express that a HFSM must contain only one initial state. This has to be defined in an external constraint that is attached to the meta-model.

There exist a number of choices for languages when it comes to expressing constraints on models. The OMG standard for specifying constraints on MOF models is the Object Constraint Language (OCL) [6]. The constraint that there must be only one initial state in a HFSM model is expressed in OCL as:-

$$context\ State\ inv:$$
$$State.allInstances() \rightarrow select(s|s.isInitial = True) \rightarrow size() = 1$$

The OCL is a high-level and complex constraints specification language for constraints. The evolving complexity in the syntax of OCL on the one hand makes it hard for one to specify its formal semantics on the other. We consider only a subset of OCL for manually specifying Prolog constraints on models.

We use the constraint logic programming language called ECLiPSe [12] (based on Prolog) to specify constraints on model properties. Predicates in ECLiPSe equips a search algorithm with results from automatic symbolic analysis and domain reduction in a model to prune its search space. Consider the model of a HFSM in Figure 1 (b). The ECLiPSe predicate that constrains the number of initial states is:

$$ModelisInitial = [HFSMStateObject1isInitial,$$
$$HFSMStateObject2isInitial,$$
$$HFSMStateObject3isInitial],$$
$$ModelisInitial :: [0..1],$$
$$ic\_global : occurrences(1, ModelisInitial, 1),$$

The *ModelisInitial* list contains unique identifiers for *isInitial* attributes of three state objects $StateObject1$, $StateObject2$, and $StateObject3$. The domain of all these identifiers in the list *ModelisInitial* is binary. The predicate *occurences* in the *ic_global* library of ECLiPSe states that the number of occurrences of the value 1 in the list is limited to 1. This means that only one of the variables in the list is assigned a value of 1. The choice of 1 for a variable automatically guides the compiler to perform domain reduction and reduce the domain of the rest of the variables in the list. The dynamically assigned domain for the other variables will be 0 and not $0, 1$. This feature drastically reduces the search space for the constraint satisfaction algorithm.

It is interesting to note that predicates in OCL are specified at the meta-model level of the modelling language while the constraints in ECLiPSe are generated for each model. In the next section we present how we generate ECLiPSe predicates for each model automatically.

In our implementation Prolog constraints are generated for a meta-model and stored in a prolog pl file. The constraints operate of lists of variables in the partial model.

## 3 Transformation: Partial Model, Meta-model, and Constraints to CLP

In this section we describe an algorithm to transform a partial model in a modelling language (meta-model and constraints) to a CLP. Keeping the structure static in the partial model we query the CLP to obtain value assignments for the properties in the model. The query process invokes a back-tracking algorithm to assign values to the variables such that all related predicates are satisfied. In this case the algorithm returns a *Yes*. If the back-tracking method fails to find a set of value assignments that can complete the model the algorithm returns a *No*. We start with describing a partial model in 3.1. In Section 3.3 we present the variables and functions that will be used in the algorithm. The algorithm itself is presented in Section 3.4.

### 3.1 Partial Model

A *partial model* is a set of objects described in a modelling language. The properties of the objects are either partially specified or they are not specified at all. What is specified is the domain of possible assignments for each and every property. This is derived from the meta-model that specifies the modelling language for the partial model objects.

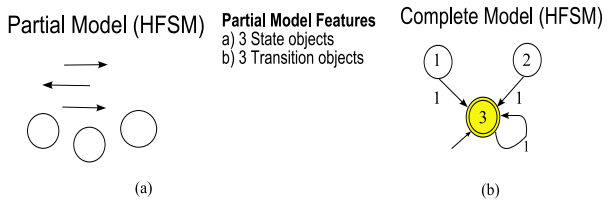


**Fig. 2.** (a) A Partial Model in HFSM (b) A Complete Model

In Figure 2 (a) we present a partial model that has 3 state objects and 3 transition objects in the HFSM language. Note that the labels, events, and other properties of the objects are not specified at all.

The domain of each property in the model is shown in Figure 3. We need to find a way to assign appropriate values for every property in the model so that it satisfies the structural requirements imposed by the meta-model and also additional constraints specified textually as CLP predicates.

| Property | Domain |
|---|---|
| HFSM.StateObject1.isInitial | {True, False} |
| HFSM.StateObject1.isFinal | {True, False} |
| HFSM.StateObject1.label | {1,2,3,...,N}, where N is positive integer |
| HFSM.StateObject2.isInitial | {True, False} |
| HFSM.StateObject2.isFinal | {True, False} |
| HFSM.StateObject2.label | {1,2,3,...,N}, where N is positive integer |
| HFSM.StateObject3.isInitial | {True, False} |
| HFSM.StateObject3.isFinal | {True, False} |
| HFSM.StateObject3.label | {1,2,3,...,N}, where N is positive integer |
| HFSM.TransitionObject1.event | {1,2,3,4,5} |
| HFSM.TransitionObject2.event | {1,2,3,4,5} |
| HFSM.TransitionObject3.event | {1,2,3,4,5} |
| HFSM.StateObject1.incomingTransition | {HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target} |
| HFSM.StateObject1.outgoingTransition | {HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source} |
| HFSM.StateObject1.hfsmStates | {HFSM.states} |
| HFSM.StateObject2.incomingTransition | {HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target} |
| HFSM.StateObject2.outgoingTransition | {HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source} |
| HFSM.StateObject2.hfsmStates | {HFSM.states} |
| HFSM.StateObject3.incomingTransition | {HFSM.TransitionObject1.target,HFSM.TransitionObject2.target,HFSM.TransitionObject3.target} |
| HFSM.StateObject3.outgoingTransition | {HFSM.TransitionObject1.source,HFSM.TransitionObject2.source,HFSM.TransitionObject3.source} |
| HFSM.StateObject3.hfsmStates | {HFSM.states} |
| HFSM.states | {HFSM.StateObject1.hfsmStates,HFSM.StateObject2.hfsmStates,HFSM.StateObject3.hfsmStates} |
| HFSM.currentState | {HFSM.StateObject1.hfsmCurrentState, HFSM.StateObject2.hfsmCurrentState, HFSM.StateObject3.hfsmCurrentState } |
| HFSM.transitions | {HFSM.TransitionObject1.hfsmTransitions, HFSM.TransitionObject2.hfsmTransitions, HFSM.TransitionObject3.hfsmTransitions} |
| HFSM.TransitionObject1.target | {HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition} |
| HFSM.TransitionObject1.source | {HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition} |
| HFSM.TransitionObject1.hfsmTransitions | {HFSM.transitions} |
| HFSM.TransitionObject2.target | {HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition} |
| HFSM.TransitionObject2.source | {HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition} |
| HFSM.TransitionObject2.hfsmTransitions | {HFSM.transitions} |
| HFSM.TransitionObject3.target | {HFSM.StateObject1.incomingTransition,HFSM.StateObject2.incomingTransition,HFSM.StateObject3.incomingTransition} |
| HFSM.TransitionObject3.source | {HFSM.StateObject1.outgoingTransition,HFSM.StateObject2.outgoingTransition,HFSM.StateObject3.outgoingTransition} |
| HFSM.TransitionObject3.hfsmTransitions | {HFSM.transitions} |

**Fig. 3.** Domain of Properties in the Partial Model

Which value is taken by a property will be decided when we transform the partial model to a CLP and solve it. This transformation process and solution is discussed in the next sections.

### 3.2 CLP as a Common Language

Requirements in MDE come from many sources. In this paper we focus on three sources of requirements.

1. The meta-model is a diagram specified in a meta-modelling language such as MOF
2. The constraints are textually specified in OCL or directly in CLP
3. The partial model is specified either diagrammatically or textually as an instance of the meta-model.

There also could be many other sources of requirements such as pre-conditions and post-conditions from programs that manipulate models and an objective function that is soft constraint on the properties of a model. These requirements are specified in different languages that are either graphical or textual and are usually expressed in the most convenient specification language such as OCL or MOF. Combining the declarative specifications from different sources into one *common language* is necessary for analysis on a common platform. The following section presents an algorithm that syntactically transforms constraints from various sources (the three that we consider) into a CLP.

### 3.3 Preliminaries

The input to the transformation consists of a partial model, the meta-model and the constraints specification of a modelling language. In Table 2 we list out symbols and their descriptions we use for describing the algorithm in the next section.

| Variable | Description |
|---|---|
| $M$ | Partial model |
| $M.properties$ | Set of all properties(attributes and references) in $M$ |
| $Model$ | List of CLP variables for all properties in $M$ |
| $Model\_property$ | List of CLP variables for a relationship from a reference called *property* |
| $property.lowerBound$ | Lower bound on multiplicity of a relationship from a reference called *property* |
| $property.upperBound$ | Upper bound on multiplicity of a relationship from a reference called *property* |
| $property\_to\_reference$ | A 0/1 variable for a relationship from a *property* to one of its *reference*s |
| $reference\_to\_property$ | A 0/1 variable for a relationship from a *reference* to its *property* |
| $C$ | Set of non-meta-model constraint predicates |
| $Model\_constraint$ | List of variables that are constrained by the constraint *constraint* |
| $constraint.predicate$ | Name of the CLP predicate for a constraint identifier |
| $constraint.parameters$ | String of additional comma separated parameters for a predicate |

**Table 2.** Symbols we use in the model to CLP transformation algorithm

The transformation algorithm uses some handy functions which we list and describe in Table 3.

| Function | Description |
|---|---|
| $isAttribute(property)$ | Returns 1 if the property is an attribute, otherwise 0 |
| $isReference(property)$ | Returns 1 if the property is an reference, otherwise 0 |
| $domainSet(property)$ | Returns domain set of a property |
| $relatedVariables(constraint)$ | Returns a set of variables that are constrained by *constraint* |

**Table 3.** Functions we use in the model to CLP transformation algorithm

The CLP we generate uses predicate libraries in ECLiPSe. The *ic* library contains predicates for hybrid integer/real interval arithmetic constraints. It also includes a solver for these predicates. The *ic_global* library contains predicates for specifying various global constraints over intervals. The standard CLP predicates synthesized in the code by the algorithm are described in the book [12]. In ECLiPSe each predicate is separated by a ',' indicating that it is a logical and of the predicates.

### 3.4 Algorithm

We describe the algorithm to transform a partial model to a CLP in six phases. In Phase 1 we synthesize CLP code to import library predicates *ic* and *ic_global*. We then synthe-

size code to start the CLP procedure *solveModel*. In Phase 2 the algorithm synthesizes the list, *Model*, of all the model properties in the partial model.

In Phase 3 the algorithm generates code to assign a domain of values to the attributes and relationships. The function *domainSet* returns a set of possible values for each attribute. For each reference variable, *property*, in the model the *domainSet* returns a set of possible related references in the model. A binary relationship variable $property\_to\_reference$ is created and has the integer domain $[0..1]$. If a relationship variable $property\_to\_reference$ is set to 1 by the back-tracking algorithm then its corresponding relationship $reference\_to\_property$ is also set to 1 by the constraint $property\_to\_reference = reference\_to\_property$. A list $Model\_property$ stores all relationship variables from the property. The multiplicity constraints on a relationship are inserted in the CLP using the *sum* predicate on the list $Model\_property$. The number of possible relationships should be between $property.lowerBound$ and $property.upperBound$

Now that constraints on domains of attributes and basic multiplicity are already included we go on to Phase 4 of the algorithm. In this phase we create a dependent variables list $Model\_constraint$ for each *constraint*. In Phase 5 we insert the CLP predicate $constraint.predicate$ for the *constraint* with parameters $Model\_constraint$ and additional parameters $constraint.parameters$, if necessary.

In the final Phase 6 we insert the predicate $labeling(Model)$ which performs back-tracking search on the CLP variables. To execute the synthesized CLP we make a query : $-solveModel(Model)$ in ECLiPSe after compiling the program. The result of the query is the set of value assignments that satisfies the constraints.

The pseudo code for the transformation algorithm is presented as follows :-

Phase 1: Import predicate libraries and start the CLP procedure

1: **print** ": $-lib(ic)$"
2: **print** ": $-lib(ic\_global)$"
3: **print** "$solveModel(Model) : -$"

Phase 2: Synthesize CLP variables for all model properties

4: $printString \leftarrow$"$Model = [$"
5: **for** *property* in $M.properties$ **do**
6: **if** $isAttribute(property)$ **then**
7: $printString \leftarrow printString + property$+","
8: **else if** $isReference(property)$ **then**
9: **for** *reference* in $domainSet(property)$ **do**
10: $printString \leftarrow printString + property$+"_to_"+$reference$+","
11: **end for**
12: **end if**
13: **end for**
14: **print** $printString.rstrip(',')$ + "];" {rstrip strips *printString* of the last ,}

Phase 3: Assign property domains and multiplicity constraints

15: **for** *property* in $M.properties$ **do**
16: **if** $isAttribute(property)$ **then**
17: **print** $property$+"::"+"["+$domainSet(property)$+"],"

```
18:    end if
19:    if isReference(property) then
20:       printString ← "Model_property = ["
21:       for reference in domainSet(property) do
22:          printString ← printString + property+"_to_"+reference+","
23:       end for
24:    end if
25:    print  printString.rstrip(',')+"];"
26:    print  "Model_"+property+":: [0..1]"
27:    print  "sum(Model_"+property+") >="+ property.lowerBound+","
28:    print  "sum(Model_"+property+") =<"+property.upperBound+","
29: end for
30: for reference in domainSet(property) do
31:    print  property+"_to_"+reference = reference+"_to_"+property+","
32: end for
    Phase 4: Create dependent variable lists
33: for constraint in C do
34:    printString ← "Model_constraint=["
35:    for variable in relatedVariables(constraint) do
36:       printString ← printString + variable+","
37:    end for
38:    print  printString.rstrip(',')
39: end for
    Phase 5: Impose constraints on lists
40: for constraint in C do
41:    print constraint.predicate+"("+constraint.parameters+","+Model_constraint+"),"
42: end for
    Phase 6: Solve Model
43: print  "labeling(Model)."
```

## 4 An Example

We consider the partial model shown in Figure 2 (a) as our example. The simple HFSM has three states and three transition objects. The algorithm presented in Section 3.4 takes the partial model and the meta-model and constraints of the HFSM modelling language as input. The output is a CLP program. We do not present the structure of the CLP program due to space limitations.

When the CLP is queried or solved the *labeling* predicate invokes a backtracking solver that selects values from the domain of each property such that the conjunction of all the constraints is satisfied. Executing this predicate returns a *Yes* if the model is satisfied else it returns a *No*. If a *Yes* is returned then the compiler also prints a set of valid assignments for the variables. Assigning these values to the partial model results in a complete model. In the simplest case, when no property of the partial model is given the result is the complete model shown in Figure 2 (b). However, when certain values for properties are already specified the synthesized CLP program has new domains for

these properties. These domains have only one value restricting the solver to choose just one assignment.

## 5 Conclusion

The idea of completing a partially specified model can be extended in many ways. For example, when we want to test a program it is important that we test it with input models that lead to covering most of its execution paths. Specific execution paths can be reached when some partial information about reaching it is already available. Such partial information can be specified in the form of a partial model with value assignments to some properties. The task of obtaining values of other properties is not directly consequential to identifying an error in a program but is necessary to produce a valid input model. Our method to complete such a partial model can be extended to large scale generation of test models to perform *coverage-based* testing of programs that manipulate models.

We solve a CLP using back-tracking as a means to achieve constraint satisfaction. At the moment we donot select different results based on back-tracking. The difference in the result is due to the knowledge already given in the partial model. Nevertheless, simply replacing a constraint satisfaction algorithm such as back-tracking with a constraint optimization algorithm opens many possibilities. One such possibility is to synthesize a model that optimizes an objective by meeting soft constraints or requirements including hard constraints specified in the modelling language. In the domain of software engineering such a framework can be used to synthesize customized software.

## References


1. Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2) (2006) 25–31
2. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-driven development using uml 2.0: Promises and pitfalls,. In: IEEE Computer Society Press. (2006)
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap,. In: FOSE '07: 2007 Future of Software Engineering. (2007)
4. OMG: OMG Home page. http://www.omg.org (2007)
5. OMG: MOF 2.0 Core Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04 (2005)
6. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03- 01-07 (2007)
7. de Lara Jaramillo, J., Vangheluwe, H., Moreno., M.A.: Using meta-modelling and graph grammars to create modelling environments. Electronic Notes in Theoretical Computer Science with editors Paolo Bottoni and Mark Minas **72** (2003) 15
8. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer (2001) 44–51
9. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon., Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proceedings of ISSRE'06, Raleigh, NC, USA (2006)
10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)

11. Neema, S., Szitpanovits, J., Karsai, G.: Constraint-based design space exploration and model synthesis. In: Proceedings of EMSOFT 2003, Lecture Notes in Computer Science. Number 2855 (2003) 290–305
12. Apt, K.R., Wallace, M.G.: Constraint Logic Programming with ECLiPSe. Cambridge University Press (2007)