Rapid Prototyping by Means of Meta-Modelling and Graph Grammars. An Example with Constraint Satisfaction Problems.

Iván Dotú and Juan de Lara*

Escuela Politécnica Superior, Ingeniería Informática
Universidad Autónoma de Madrid
Ivan.Dotu@ii.uam.es, Juan.Lara@ii.uam.es

Abstract. In this paper, we present our approach for rapid prototyping – by means of meta-modelling and graph grammars – in the multi-paradigm modelling tool AToM³. This is a tool which allows the graphical definition of formalisms by means of meta-modelling using high-level, graphical notations. As meta-models are stored as attributed, typed graphs, their manipulation (simulation, transformation into another formalism, optimisation and code generation) can be visually and formally expressed as graph transformation. In this way, computations become high-level models, expressed in the graph grammars formalism. As an example of these concepts, we show the automatic generation of a tool (by means of meta-modelling and graph grammars) to graphically define and solve Constraint Satisfaction Problems (CSP).

Keywords: Meta-Modelling, Graph Grammars, Rapid Prototyping, Automatic Code Generation, CSP.

1 Introduction

One of the problems of today’s software engineering practice is the need for higher levels of quality and productivity. A possible way to achieve such objectives is to increase the abstraction level during the development process. If from high-level models, we are able to generate most or all the code for the application we want to build, then we have reduced a significative amount of effort in coding, testing and maintenance. Meta-Modelling is a way to partially obtain such benefits as one can use graphical languages to describe the kind of models we want to deal with (that is, defining a formalism). A meta-model processor is able to generate a tool to process models in the described formalism. But usually the generated tools have very limited functionality (typically loading/saving models and checking their correctness) as we have only provided static information about the meta-model. Extra functionality is usually added by coding in low-level, textual languages.

* Partially sponsored by the Spanish Ministry of Science and Technology (TIC2002-01948)
In contrast to that, we propose to complement the static meta-models with high-level models of the computations, in the form of graph grammars. These are a natural notation to express manipulations on models, as we store models and meta-models as attributed, typed graphs. Graph grammars can be graphically (and formally) specified, thus they become high-level models, reducing the need for coding to a minimum. The advantage of this approach is that the necessary effort to create, test and maintain Domain-specific modelling tools is drastically reduced.

We show the implementation of these concepts in the multi-paradigm tool AToM³ [2] that one of the authors developed in collaboration with the MSDL lab at McGill University (Montreal). As an example, we show how to build a meta-model for a formalism to define Constraint Satisfaction Problems (CSP), the automatic generation of a tool to build this kind of models, and the definition of a graph grammar for their resolution.

2 Graph Grammars

Graph grammars [10] are a generalization of Chomsky grammars for graphs. They are composed of rules having graphs in their left and right hand sides (LHS and RHS). Rules are compared with an input graph (called host graph). If a matching is found between the LHS of a rule and a subgraph in the host graph, then the rule can be applied and the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied (both are expressed as Python code in AToM³).

Some graph rewriting systems have control mechanisms to decide which rule should be checked next. In AToM³ rules are ordered according to a priority, and are checked from higher (smaller numbers) to lower priority (bigger numbers). If a rule makes a match, after its application the system starts trying again the higher priority rule in the list. The graph grammar execution ends when no rule can be applied. In AToM³, graph grammars have also initial and final actions (expressed in Python) to be performed before and after the graph grammar execution.

In AToM³ we use models expressed in the graph grammars formalism to manipulate models (at any meta-level). The kind of model manipulations we are interested in are model execution (operational semantics, in this paper, we show an example of this kind of manipulation), optimisation (for example, reducing its complexity) and transformation into another (graphical or textual) formalism (denotational semantics). Using graph grammars to specify model manipulations have some advantages over using a textual programming language. Graph grammars are a natural, formal, visual, declarative and high-level representation of the computation. These become high-level models, which are more understandable, reusable and easier to maintain than low-level code. Moreover, the theoretical foundations of graph grammars may assist in proving correctness and convergence properties of the transformations. The main drawback is that
its use is constrained by efficiency as in the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node and edge types and attributes can greatly reduce the search space in the matching process.

3 AToM³

AToM³ is a tool for computer aided multi-paradigm modelling (CAMPaM), which includes meta-modelling, multi-formalism and modelling at different abstraction levels. The main idea of the tool is “everything is a model”. Most of its components are indeed models (which can be modified by the user), such as the user interface, the type system, formalisms and model manipulations. Also, during implementation, the AToM³ kernel has been grown from a small initial kernel with code-generating capabilities.

![Fig. 1. Meta-... Modelling in AToM³.](image)

AToM³’s architecture is shown in Figure 1 (for more details about this architecture, see [2]), in which models (at any meta-level) are represented as white boxes, having on their upper-right corner an indication of the (meta-)³ model they were specified with. The AToM³ Kernel is responsible for loading, saving, creating and manipulating models, as well as for generating code for the defined formalisms (the code structure is shown in the upper-right corner in figure 1, labelled as “AToM³ (Meta-) Models’ structure”). Part of this code-generation task is performed using a graph grammar. The generated code must be loaded on top of the Kernel to allow the user building models in the defined formalism. One of the components of the generated files is a model of a part of the AToM³
user interface. This user interface model follows the “Buttons” formalism, and has its own meta-model. Initially, this model represents the necessary buttons to create the entities defined in the formalism’s meta-model, but can be modified to include for example, buttons to execute graph grammars on the current model.

Entity-Relationship and UML class diagrams (with constraints) are available at the meta-meta-level to define other formalisms and meta-formalisms. Constraints can be specified as OCL or Python expressions, and the designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be syntactic (such as editing an attribute or connecting two entities) or purely graphical (such as dragging, dropping or moving.) The Constraint Manager checks at run-time whether the constraints associated with the events are satisfied or not. In the latter case, the event is not performed (or its effects are undone, if it was a post-condition). When defining a meta-model, the formalism entities must be specified together with their relationships, attributes and their graphical appearance. This can be icon-like or arrow-like with optional icon decorations in the center, segments and extremes.

In AToM³, computations can be expressed either directly in Python, or as graph grammars. Another advantage of the latter option is that the user does not have to know the AToM³ internals to add new functionalities to the generated tool. Graph grammars are modelled graphically, possibly using several additional meta-models (besides the graph grammar meta-model.)

4 Example: Solving CSP with AToM³

In order to illustrate all the concepts explained above, we are presenting a simple example of the definition and resolution of a well-known formalism within artificial intelligence: Constraint Satisfaction Problems. These consist of a given set of variables, a finite and discrete domain for each variable, and a set of constraints. Each constraint is defined over some subset of the original set of variables and limits the combination of values that the variables in this subset can take. The goal is to find an assignment to the variables satisfying all the constraints [6].

Our discussion here is restricted to a particular kind of CSP where all the constraints are defined over exactly two variables. This is called binary CSP and it is not a restriction, since it is possible to convert CSP with n-ary constraints to another equivalent binary CSP [9].

4.1 The CSP Meta-Model

In this section we define the CSP meta-model in AToM³ by means of the Entity-Relationship meta-formalism. Following the description given above it is easy to discover the first entity: variables. As each variable has a finite and discrete domain, this is another entity in our meta-model. Both entities are related by means of a relationship (that we call static) which connects each variable to its assigned domain. Finally, we have modelled constraints as a relationship between two variables (as we are modelling binary CSP). These are provided
with a binary operator that is evaluated at run-time to check if the constraint holds.

![Fig. 2. The Meta-Model for CSP (left), and the generated tool (right).](image)

Figure 2 (left) shows the meta-model defined using AToM³, where we have added some extra entities and relationships to help in the resolution phase. The current_var entity and its relationship with variable (named `point` to) are just the representation of a pointer to the variable we are trying to assign a value during the model resolution. Similarly, the relationship named as current between variable and domain represents the list of possible values in a given moment during resolution.

Figure 2 (right) shows a model built with AToM³ once the files generated by the previous meta-model are loaded in the tool. Note how variables have been represented by means of circles with the name on top and the value inside. Domains have been represented as rectangles, with the list of possible values inside. Constraints appear as arrows with the constraint condition besides (in the figure, all constraints are labelled as "!=", that is, the pairs of variables (0, 1), (0, 2), (1, 3) and (2, 3) must have different values). The meaning of this specific CSP model is explained in more detail in section 4.3. It can be noted that a part of the user interface (the column of buttons to the left) has changed with respect to the left window. We have modified the user interface model, adding two buttons ("init" and "simulate") to solve the model. The resolution has been modelled using graph grammars and is explained in the next section.

### 4.2 Modelling the Backtracking Algorithm with Graph Grammars

CSPs can be solved using a number of methods, among them generate-and-test, backtracking, backjumping, forward checking and arc-consistency [6]. Here, we use the backtracking algorithm to solve CSPs. This method consists on performing a sequential instantiation of the model variables. When all variables
relevant to a constraint are instantiated, its consistency is checked. If the partial instantiation is consistent the search continues, if not, backtracking eliminates a subspace from the cartesian product of all variable domains. The algorithm is essentially a depth-first search in the space of potential CSP solutions.

In order to implement this method, we first need to initialize the model we want to solve, preparing it for the backtracking method. This is issued by two graph grammars. The first one (“initialisation”) has only one rule that creates the current_var entity (the pointer to the current variable) and relates it with the first variable to be instantiated. The second one (“curDomains”) has one rule too which assigns a current domain to each variable (by means of the current relationship), duplicating the static domain of the variable. Once these grammars have been applied, the CSP model is ready to be solved with our backtracking algorithm modelled by means of a graph grammar (called “simulation”). This is composed of three rules which we present ordered by priority (higher to lower):

- The first rule (Backtracking) is applied if the current_var entity points to a variable with an empty current domain. This means that we need to backtrack to the last variable and copy its static domain to its empty current domain. In case the rule is considering the first variable, it can not backtrack (there is no previous variable) so execution finishes without finding any solution. The rule is shown in Figure 3. Note how nodes and connections on both LHS and RHS are labelled with numbers. If a number appears in the LHS but not in the RHS, the node or connection is deleted when the rule is applied. If the number appears in the RHS but not in the LHS, then the node or connection is created when the rule is applied. Finally, nodes and connections which are maintained appear with the same number. Nodes and connections in the LHS must be provided with the attribute values that will make a match with nodes and connections in the host graph. In AToM³ we can specify that any value of these attributes will make a match (shown as ⟨ANY⟩ in the figure), or we can set a specific value.

  In attributes of nodes and connections in the RHS of a rule, we can either maintain the value (and this is shown as ⟨COPIED⟩), give a specific value, or calculate a new one by means of Python code. This code can use other node and connection attributes. This latter option is shown as ⟨SPECIFIED⟩ in the figure. For example, we have copied the values list of node 6 into the values list of node 4 in the RHS.

- Rule number two (NoAssign) looks for the current variable with a non empty current domain, and tries to assign it the first value from its current domain. As this value is not consistent with the constraints, the rule eliminates it from its current domain. We do not need to check whether the current domain is empty, because if it were empty, rule number 1 would have been applied. The rule is shown in Figure 4. Function consistent checks whether a value from a variable is consistent with the constraints of other variables that have already been instantiated.

- Rule number three (Assign) is similar to the previous one but in this case the value is consistent with the constraints, therefore it is assigned to the
variable and removed from its current domain. The pointer to current.var advances to the next variable (the one returned by function next.var in the ACTION section) unless the current variable is already the last one, which means the algorithm has found a solution.

With this three rules we have easily and graphically modelled the backtracking algorithm. Though it is a simple method we can implement more efficient ones changing the last.var, next.var and consistent functions. For example,
other heuristics have been proposed in the literature to guide the search, such as performing variable and value ordering. In our example, both orderings are lexicographic, but they can easily be changed. To define a new variable ordering we just need to redefine the next_variable function. Value ordering can be achieved by changing the way the grammar chooses values from current domains.

In ATOM³, there are several execution modes for graph grammars: continuous, step by step, and animated. In the latter case, the user can assign a delay to each rule (that can be changed at run-time by the rules). The user can also choose what to do in case of multiple matchings of the LHS of a rule in the host graph: manually select one of them, let the system choose a random one or execute all matchings which do not overlap. In our example, the execution of these graph grammars have been assigned to two buttons in the user interface of the CSP formalism. The init button, runs both initialisation grammars sequentially in continuous mode. The simulate button runs the backtracking algorithm grammar step by step.

4.3 Solving a Quasigroup

A quasigroup is an ordered pair \((Q, \cdot)\), where \(Q\) is a set and \(\cdot\) is a binary operation on \(Q\) such that the equations \(a \cdot x = b\) and \(y \cdot a = b\) are uniquely solvable for every pair of elements \(a, b\) in \(Q\) [5]. The order \(n\) of the quasigroup is the cardinality of the set \(Q\). A quasigroup can be seen as an \(n \times n\) multiplication table defining a Latin Square, which must be filled with colours in such a way that colours in each row and column must be different.

As an example, we solve the \(2 \times 2\) quasigroup that was shown in Figure 2. The problem is kept simple on purpose for space constraints. The first thing to do is to initialize the model by clicking on the init button. This triggers the execution of the initialization graph grammars, which leads to the model depicted in the upper left corner in Figure 6. This figure shows how the current_var entity has been created and points to the first variable to be assigned (named “0”), and a current domain has been created for each variable. During execution, these domains will contain the possible values to be assigned to the variables.

Next, the user has to click on the simulate button to solve the model. Some resolution steps are shown in Figure 6. In the first step the grammar finds a current variable (upper-left variable) with a consistent value in its current domain, assigns it and changes the reference of the current_var pointer to the next variable (labelled as “1”). In the second step, the grammar tries to assign the first value to the variable pointed by the current_var entity, but it is not consistent therefore, rule no Assign is triggered. The execution lasts for six steps, the algorithm finishes with the solution that can be found in the variables in the lower right model of Figure 6.

5 Related work

There are some similar, well-known tools in the graph grammars community, such as GenGed [1]. In ATOM³ there is no structural difference between the
generated editors (which could be used to generate other ones) and the editor which generated them. Additionally, almost everything in AToM³ has been defined by a model and thus the user can change it. Other commercial tools, such as DoME [7] or MetaEdit+ [8] use a textual, low-level language for the definition of the model manipulations. In opposition, in our approach the user can define transformations as models in the graph grammars formalism. Among other advantages (besides the ones stated in sections 1 and 2), this frees the user with the necessity of having to know many details of the internals of the tool.

It is worth mentioning that, to the knowledge of the authors, using graph grammars this way to solve CSPs represents a novel approach. Other approaches to solve CSPs using grammars are Constraint Handling Rules, but their goal is to represent simplifications and resolution of constraints with grammar rules [4].

6 Conclusions and Future work

In this paper we have shown the power of the combined use of Meta-Modelling and Graph Transformation for rapid prototyping. With Meta-Modelling, it is possible to describe using a high-level notation the kind of models one would like to deal with. By means of graph grammars it is possible to define the computations to be performed on the models (simulation, transformation into another formalism, code generation and optimisation) on a formal basis, but also
graphically. As an example of these concepts, we have presented the generation of a tool to define and solve Constraint Satisfaction Problems. This generation has been achieved using graphical, high-level notations reducing the necessity of coding to a minimum. The advantage of this approach is a drastic reduction in development time (including testing and maintenance). If more complex functionalities were needed, they can be implemented by adding Python code.

A side-effect of our approach to model computations with graph grammars is in education. Graph grammars are a natural and didactic way to express model manipulations [3], and one gets insight into the computation to be performed. Graph grammar models become an executable specification of the computation to be performed. In ATOM [3] one can trace its execution step by step, or by means of animation. In the case of the example presented in this paper, their use allows us to graphically trace the steps in the assignments of variables, without having to code complex graphical routines. We have also used graph grammars with educational purposes to show the simulation of discrete-event models [3]. This idea can be extended to other areas, for example to show the animation of algorithms on data structures such as lists, stacks, vectors (and of course graphs); as well as for visualizing the (simulated) execution of parallel algorithms in networks.

References