Supporting User-Oriented Analysis for Multi-View Domain-Specific Visual Languages

Esther Guerra [a,*], Juan de Lara [b], Alessio Malizia [a], Paloma Díaz [a]

[a] *Computer Science Department, Universidad Carlos III de Madrid (Spain)*

[b] *Polytechnic School, Universidad Autónoma de Madrid (Spain)*

Abstract

The integration of usable and flexible analysis support in modelling environments is a key success factor in Model-Driven Development. In this paradigm, models are the core asset from which code is automatically generated, and thus ensuring model correctness is a fundamental quality control activity. For this purpose, a common approach is to transform the system models into formal semantic domains for verification. However, if the analysis results are not shown in a proper way to the end-user (e.g. in terms of the original language) they may become useless.

In this paper we present a novel DSVL called BaVeL that facilitates the flexible annotation of verification results obtained in semantic domains to different formats, including the context of the original language. BaVeL is used in combination with a consistency framework, providing support for all the verification life cycle: acquisition of additional input data, transformation of the system models into semantic domains, verification, and flexible annotation of analysis results.

The approach has been empirically validated by its implementation in the AToM^3 meta-modelling tool, and tested with several DSVLs. In this paper we present a case study for the analysis of a notation in the area of Digital Libraries, where the analysis is performed by transformations into Petri nets and a process algebra.

Key words: Domain-Specific Visual Languages, Consistency, Back-Annotation, Formal Methods, Model Transformation, Modelling Environments

* Corresponding author. Address: Escuela Politécnica Superior, Dep. Informática, Univ. Carlos III de Madrid, Av. Universidad 30 - 28911 Leganés (Madrid) Spain

Email addresses: eguerra@inf.uc3m.es (Esther Guerra), jdelara@uam.es (Juan de Lara), amalizia@inf.uc3m.es (Alessio Malizia), pdp@inf.uc3m.es (Paloma Díaz).

1 Introduction

Model-Driven Development [32] (MDD) seeks increasing quality and productivity in software development by considering models as the primary asset, from which the code of the application is generated. The idea is to capitalize the knowledge and expertise in a certain application domain by providing designers with a *Domain-Specific Visual Language* (DSVL) that describes the main concepts of the domain, as well as code generators for part or all the final application. Thus, designers work closer to the problem space as they use high-level, intuitive, powerful notations for the problem to be solved.

Moreover, in order to facilitate the design of complex systems, it is frequent splitting their specification into smaller parts that use the most appropriate notation to deal with a specific concern. We call *Multi-View DSVL* [17] (MV-DSVL) to a family of complementary, possibly overlapping DSVLs, each one of them used to describe the system from a different perspective. UML is a prominent example of this kind of languages, although it is general-purpose.

Since in MDD the code is automatically generated from the models, there is a clear need to verify their syntactic and semantic correctness. In this respect, a common approach is the use of hidden formal methods that provide analysis mechanisms for the system dynamic semantics. Formal methods [3,6] are techniques based on mathematics that help in the specification and verification of systems in order to obtain products with higher quality and fewer errors. One of their drawbacks is their difficulty, as they usually require experts in the given verification method. Thus, for them to be useful in practice, usable tools hiding their complexities are required. When using hidden formal methods, systems are specified with an intuitive notation (e.g. UML, DSVLs, etc.) and then translated into a formal semantic domain (e.g. Petri nets or logic) where the analysis of interest is performed [16,31–33]. Typical properties amenable to study are the absence of resource blocks, the impossibility of unwanted situations or the reachability of some system states.

In hidden formal methods, not only the input language should be intuitive to the user, but also the analysis results. If these are not properly shown to the users, e.g. in the notations they are experts in, they can be reluctant to the use of verification methods. As a consequence, the analysis results may be ignored or in the worst case misinterpreted, perhaps leading to wrong design decisions. The back-annotation aspect has been commonly neglected in the literature. At most one finds ad-hoc, hard-coded solutions for concrete applications, but not general ones. In this work we try to overcome this problem by proposing a means to facilitate the integration of hidden formal methods in modelling environments by providing a high-level mechanism for specifying how analysis results should be presented back to the end-user.

This paper completes our previous work on analysis support for MV-DSVLs [16] with the definition of a general flexible mechanism for the back-annotation of verification results. For this purpose, we propose a novel DSVL called BaVeL (Back-annotation of Verifications Results Language) that allows the designer of the modelling environment to graphically specify the verification life cycle. This includes defining additional data needed for the verification (which may be requested to the user, read from files, etc.), model filtering to keep the relevant information, transformations into semantic domains, execution of verification methods, and mechanisms for the interpretation of results.

From the MV-DSVL specification (a meta-model) and the BaVeL model, a modelling environment is generated with a set of integrated analysis mechanisms that return the verification results in the most appropriate format, which in many occasions is in terms of the original MV-DSVL. In this way, the complexities of the verification method are hidden to the end-user and do not impose any additional cognitive overload. Altogether, the approach facilitates the definition of usable analysis techniques and produces rich modelling environments with user-oriented verification tools, where the formal methods are hidden behind the used MV-DSVL. Moreover, the framework is easily customizable to any source MV-DSVL or target semantic domain.

This approach is supported by the AToM^3 tool [21]. To the best of our knowledge, this is the only metaCASE tool integrating high-level mechanisms for the specification of the verification workflow. We illustrate its use by generating a modelling environment for VisMODLE [24], a MV-DSVL in the Digital Libraries domain. The environment relies on Petri nets [26] and a process algebra [23] as hidden formal methods in order to analize system properties such as deadlocks, reachability of states, invariants and refinement. The results obtained in the analysis are shown to the user in the VisMODLE notation.

Paper organization. We first give an overview of meta-modelling and model-to-model transformation, putting stress on graph transformation [8,9]. Next, Section 4 introduces our framework for the specification of MV-DSVLs, which includes the transformation of the system models into semantic domains. Section 5 presents BaVeL. Sections 6 and 7 illustrate the support of the framework using AToM^3 with a case study for the analysis of VisMODLE. Section 8 evaluates the usability of BaVeL and the generated tools. Next, Section 9 compares with related research. Finally, the paper ends with the conclusions.

2 Meta-Modelling by Example

Meta-modelling [32] is a common technique for describing DSVLs and generate modelling environments for them [21]. A meta-model is a model that describes

the DSVL syntax (i.e. all its valid models). It is usually built using class diagrams plus additional restrictions expressed in constraint languages such as OCL. We say that a model conforms to its meta-model or that it is a valid instance of it, when it is structurally correct (i.e. the model uses types defined in the meta-model, and respects the meta-model associations and cardinality constraints) and fulfils the additional restrictions. In the case of a DSVL made of several diagram types (i.e. a MV-DSVL), its definition is based on a single meta-model that defines and relates all the diagram types' concepts. This is for example the approach followed by UML. The different diagram types or *viewpoints* are projections or submodels of the complete meta-model, and the same element can belong to different viewpoints.

As an example, Fig. 1 shows the meta-model of VisMODLE [24], a MV-DSVL for the Visual MOdeling of Digital Library Environments that will be used as running example in this paper. VisMODLE tries to alleviate the lack of formal models for the design of user interfaces and interactions within Digital Library (DL) systems, a kind of systems that involves a wide audience, with prominent examples such as *Google books*, *IEEE Explore* or *ACM Portal*. Its meta-model defines five different viewpoints from which only two are emphasized in the figure with polygons. Note that they share some concepts, that is, there are elements that belong to the meta-models of both viewpoints. The five viewpoints are the following:

- *Collections*, which defines sequences of arbitrary media types (e.g. characters or images) corresponding to information contents interpreted as documents.
- *Structural*, which specifies the structure of the collections in the DL.
- *Services*, used to describe activities, tasks and operations defining the functionality of the DL. Services can be synchronous or asynchronous, depending on whether they can handle one message invocation at a time, or an arbitrary number of them.
- *Societal*, which defines the relationships between the DL actors and the services that operate on document collections. Actors represent users, hardware and software components that use or support DL services.
- *Behavioural*, used to define the behaviour of individual actors and services specified via state machines. The transitions of the state machines are fired whenever a certain event occurs, which corresponds to the arrival of a request to the actor or service. In the transition, an action can also be specified, which is the sending of a request to another actor or service.

The meta-model of a language usually describes its abstract syntax. In the case of DSVLs, a visualization must also be specified for their elements, which is called their concrete syntax. Meta-modelling tools use the specification of the abstract and concrete syntax in order to automatically generate a modelling environment.

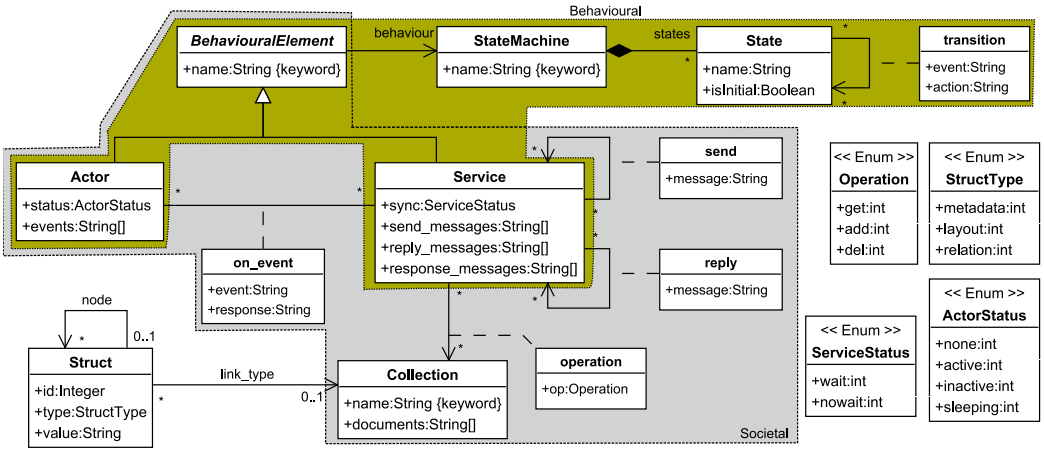
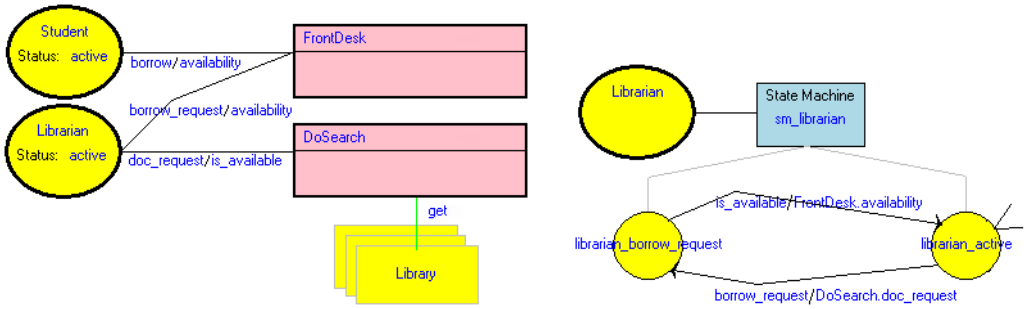


Fig. 1. The VisMODLE meta-model.

For instance, Fig. 2 shows two models in concrete syntax conforming to the VisMODLE societal and behavioural viewpoints, respectively. Both models belong to a simple DL example that will be used throughout the paper. The left model involves the actors `Student` and `Librarian` (represented as ellipses in concrete syntax), and the services `FrontDesk` and `DoSearch` (depicted as rectangles with two compartments). `FrontDesk` is responsible for managing communication between students and librarians, and `DoSearch` executes queries on a collection of documents named `Library` (shown as multiple rectangles). Services are connected to the actors that make use of them.



(a) Societal model (b) Behavioural model

Fig. 2. Some diagrams of the example DL.

Fig. 2 (b) shows the behavioural model for the actor `Librarian`. His initial state is `active`. When receiving a `borrow_request` event, he sends a `doc_request` message to service `DoSearch` and changes his state to `borrow_request`. Then, if he receives an `is_available` response, he becomes `active` again and sends the availability (documents contained in the collection matching the request) to service `FrontDesk`. The example DL has four behavioural models, one for each actor and service, although only this one is shown in the paper.

Finally, an appropriate modelling environment for a (MV-)DSVL should not only handle its abstract and concrete syntax, but should support *intra-* and

inter-diagram syntactic consistency as well. The former consists in checking that diagrams in isolation are consistent with their viewpoint meta-models. The latter must ensure that elements are coherent throughout the different diagrams, which includes mechanisms for change propagation. For example, in VisMODLE, the concept of *Service* is used in behavioural and societal diagrams; therefore, changing the name of a service in one diagram should be reflected in all the diagrams containing the service. Other inter-diagram syntactic checkings for VisMODLE include, e.g., that no state machine transition refers to non existing services, or that we cannot define a state machine for a non-existent actor.

In addition to syntactic correctness, the environment should provide support for *dynamic semantics* consistency in order to check that the behaviour expressed by the combination of the different diagrams is the intended one. For example, in the case of VisMODLE one may wonder whether an actor or service can reach a deadlock state, if the DL system can get blocked, or analysing if a service can receive an unbounded number of requests that could lead to overflow. These kinds of analysis can be achieved either defining an operational semantics for the different diagram types, or translating the models into a semantic domain for further analysis and then showing back the results in the original notation. In this paper we follow the second approach and rely on model-to-model transformation to implement the translation.

3 Model-to-Model Transformation

Model-to-model transformation consists in translating a source model conformant to a meta-model, into a target model conformant to a possibly different meta-model. Many specialized languages for model transformation exist, ranging from textual [1,29] to visual [8,29,30]; declarative [8,30] to imperative through hybrid [1,29]; and semi-formal [1,29] to formal [8,30].

In this paper, we use graph transformation [8,9] (and more specifically triple graph transformation) as model transformation language for translating DSVL models into semantic domains for analysis. Graph transformation is a declarative, visual and formal technique for graph manipulation. Note that, as models and meta-models can be represented as attributed typed graphs [8], they are also suitable to be manipulated by graph transformation. We use it because its visual nature makes rules intuitive, as it is possible to use the concrete syntax of the DSVL in the rules. Moreover, its formal basis makes possible to prove interesting properties of the transformations themselves, e.g. confluence, rule conflicts and termination (partially) [8].

Graph transformation systems are made of rules with a left and a right hand

side (LHS and RHS) graphs. The LHS expresses pre-conditions for a rule to be applied, whereas the RHS contains the rule's post-conditions. In order to apply a rule to a *host graph*, a morphism (an occurrence or match) of the LHS has to be found in it. Then, the rule is applied by substituting the match by the rule's RHS. Such rule application is called direct derivation. In addition, rules can be equipped with application conditions that restrict their applicability. For instance, a Negative Application Condition (NAC) is a graph that must not be found in the host graph for the rule to be applied. The application of a graph grammar to a host graph consists in a non-deterministic application of its rules until they are no longer applicable.

Graph transformation is useful for in-place transformations, such as animation or model refactorings. However, for model-to-model transformation, it is desirable to separate the source and target models and keep mappings relating their elements. Triple graphs [16,30] can be used for this purpose. They are made of three separate graphs called source, target and correspondence. The nodes in the latter can have morphisms to the nodes and edges in the other two graphs, thus pointing out relations between the source and target graphs's elements. In [16] we demonstrated that we can use the theory of graph transformation developed in [8] to manipulate triple graphs. In this way, similarly to graph grammars for single graphs, Triple Graph Transformation Systems (TGTSs) [16,30] allow rewriting triple graphs. That is, TGTS rules have triple graphs in their LHS, RHS and NACs.

For example, Fig. 3 shows a TGTS rule on top whose components are triple graphs made of a VisMODLE state machine (upper part), a Petri net (lower part) and a correspondence graph in between. The numbers indicate which elements are the same in the rule's components. Thus, the rule creates a Petri net place related to a state if the state is not related to any place (checked by the NAC). In the figure, the rule is applied to the left triple graph below by identifying the state in the LHS with state `student_borrow` in the triple graph, which is shown in a shaded region. The state in the LHS cannot be matched to state `student_active` because this is already connected to a place through a correspondence element, and this situation is forbidden by the NAC.

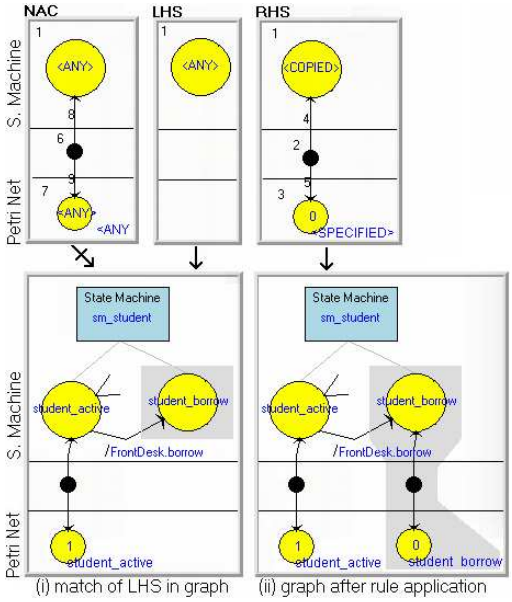


Fig. 3. Application of TGTS rule.

For further details, see [8,9] for the graph transformation theory, and [16,30] for triple graph grammars and transformation systems.

4 A Consistency Framework for MV-DSVLs

Our framework for the consistency of MV-DSVLs is based on meta-modelling for the specification of the syntax of the different system viewpoints and intra-diagram consistency, on TGTSs for the syntactic and dynamic inter-diagram consistency, and on BaVeL for the specification of the verification workflow. The latter includes visual patterns for the back-annotation of analysis results into the source notation. Fig. 4 shows the overall organization of the approach. The left part shows the process of specifying a modelling environment for a MV-DSVL, whereas the right part shows the use of the modelling environment that results from such specification. Next, we explain the steps in this framework, corresponding to the different numbers shown in the figure.

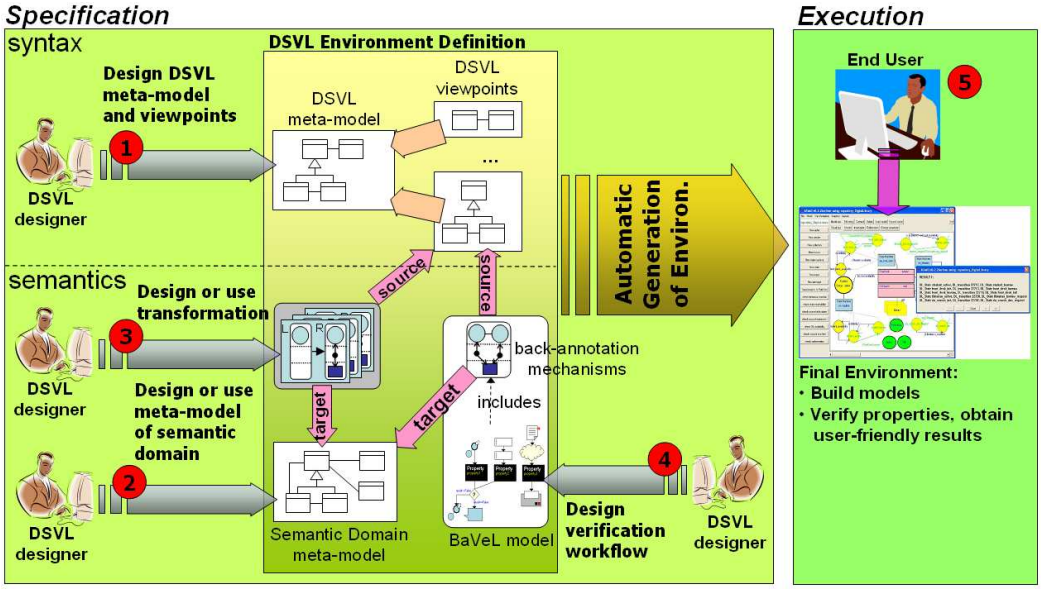


Fig. 4. Scheme of the proposed consistency framework.

Step 1. First, the DSVL defines the complete MV-DSVL syntax with a single meta-model that contains all language elements. Diagram types (DSVL viewpoints) are defined as possibly overlapping submodels or projections of the complete meta-model (see e.g. Fig. 1 for the case of the VisMODLE meta-model). In this step a concrete syntax should also be provided.

Step 2. Once the syntax is defined, the DSVL designer specifies the properties of interest to be verified in the notation. In general, these can be classified as security ("something bad never happens") and liveness properties ("something good eventually happens") [25]. The former represent

system invariants, while the latter are requirements that may not hold continuously though the system should guarantee their eventual realization. In the case of VisMODLE, absence of system deadlocks, or buffer overflows are example of security properties. Reachability of states or delivery of a given request are examples of liveness properties. In order to verify such properties, an appropriate formalism or semantic domain has to be selected. For this purpose, a meta-model of the semantic domain has to be built, or reused if already exists. Note that several semantic domains may be used if all properties cannot be verified with the same formalism. For example, next section will show the use of Petri nets an a process algebra as semantic domains for VisMODLE.

Step 3. Next, a model-to-model transformation has to be provided to transform the MV-DSVL (i.e. from one of its diagrams or a combination of them) into the semantic domain. If several semantic domains are present, a transformation must be given for each one of them. We specify the transformation by means of a TGTS that creates the target model as well as correspondence elements (mappings) relating the source and target models. In the case of VisMODLE, one of the semantic domains (process algebra) is a textual formalism. However such transformation can be built by providing a meta-model for its abstract syntax.

Step 4. This is the last step in the specification of the environment. Here, for each transformation, a model is given specifying the verification workflow for the properties to be verified in the semantic domain. For this purpose we have designed a DSVL called BaVeL. The language allows selecting input data, filtering the system models, performing the verification by calling external or internal analysis tools, and selecting the output format of the verification. The latter includes back-annotation visual patterns specifying how the verification results should be reflected in the source model. As the back-annotation process is one of the main contributions of this paper, we dedicate Section 5 to present the BaVeL notation.

Step 5. Once the previous steps have been performed, a customized modelling environment for the MV-DSVL is automatically generated. Such environment allows the final user to build models conforming to the different diagram types, as well as verifying the properties that the designer made available. Syntactic and static semantics consistency between diagrams is achieved by means of TGTSs automatically generated from the meta-model information. These build a *repository* made of the gluing of all the system views [16] and propagate changes to the other diagrams if necessary. This behaviour, illustrated in Fig. 5, is an implementation of the model-view-controller pattern: when the user modifies or creates a view (step 1), the changes are propagated to the repository (step 2) and from there to the other views (steps 3a and 3b). Note that this syntactic consistency mechanism is

hidden to the end user of the final environment. For the case of VisMODLE, the user builds models like those in Fig. 2 and the tool internally constructs a repository like the one in Fig. 13.

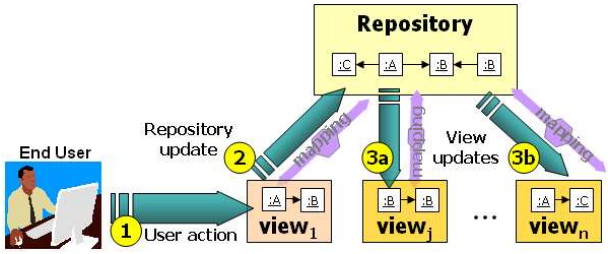


Fig. 5. Working scheme of generated environment: Syntactic consistency.

The verification of the dynamic semantics includes transformations into semantic domains. This process is transparent to the end user who just selects the property to analyze, and the verification results are returned as defined by the DSVL designer through a BaVeL specification in step 4. Such working scheme is shown in Fig. 6. In this way, performing an analysis includes internally executing the associated TGTS into a semantic domain (step 2), invoking the analysis algorithm (step 3), and then performing the back-annotation (step 4) using the BaVeL information and the traces left by the transformation.

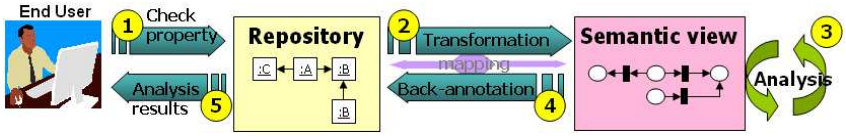


Fig. 6. Working scheme of generated environment: Dynamic semantics consistency.

5 BaVeL: Back-annotation of Verification Results Language

In the transformation into a semantic domain, a set of mappings is created relating elements in the source and target models. As triple graph grammars [30] allow bidirectional transformation, one could think of back-annotating the results obtained in the semantic domain by executing the inverse of the transformation system. However, this does not work in general because it restricts the approach to 1-to-1 annotations, that is, at most one element of the source model is annotated for each element of the target model. In other words, one is limited by the mappings created by the transformation into the semantic domain. On the contrary, sometimes, a result in the semantic domain must be reflected in several source elements and not only in the one that is related through a correspondence element. Some other times, the result derived from the analysis is not an element of the semantic model, but can be a boolean

value, a number or a matrix. In all these cases, the back-annotation cannot be achieved by simply following the mappings given by the correspondence graph. Finally, a semantic analysis can return several solutions that should be reflected in the source model one at a time, and not simultaneously.

For all these reasons, we use uni-directional TGTSs to perform the model transformation, and provide a separate high-level description of the different properties to be analyzed together with their respective back-annotation mechanisms. For this purpose we have defined a DSVL called BaVeL that allows to graphically specify different ways of back-annotating the verification results. Moreover, it includes primitives to configure the whole verification process: from input gathering to showing the analysis results.

The BaVeL meta-model is shown in Fig. 7. In this notation, a property to be verified is specified by giving its *name*, its *description*, and an *action* consisting on a call to the specific, probably external, analysis method and tool. Each property can be assigned a high-level description of the previous data input process necessary for the analysis (abstract class *Input*), as well as the subsequent output mechanisms used to reflect each specific analysis result using the most appropriate representation (abstract class *Output*).

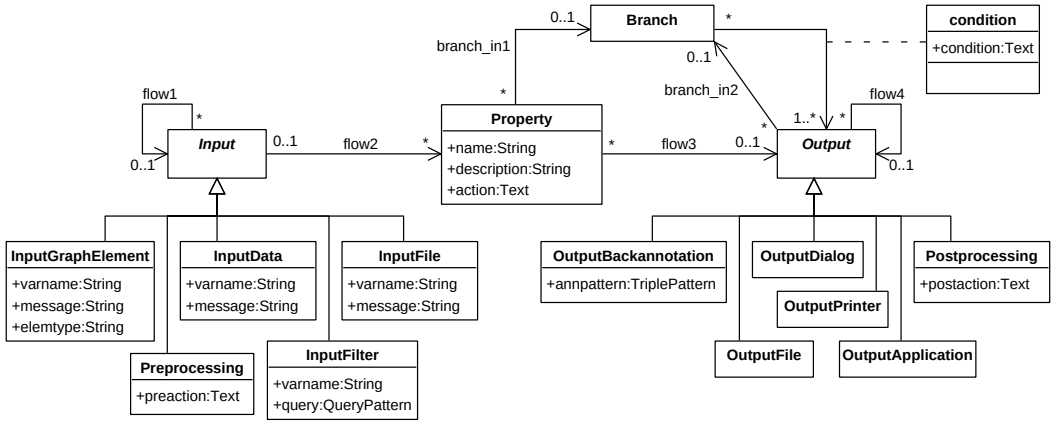


Fig. 7. BaVeL meta-model.

We have identified four typical input types that can be combined in a sequential way to specify composite data input processes. The identified types are the interactive selection of a model element of certain type (class *InputGraphElement*) or an input file (*InputFile*) by the user, the input of a numerical or textual value (*InputData*), and the filtering of the model under study by a graph query pattern that extracts relevant information or reduces the parts of the model to be analyzed (*InputFilter*). A graph query pattern [16] is a declarative graphical query language, useful to extract information from a base model. In the extraction process, a set of mappings between the base and the resulting model is created that enables their synchronization. Briefly, a query pattern is made of a meta-model with the elements to be obtained

as result of the query, and optionally a set of positive and negative graph restrictions on these meta-model elements. We use this query notation to define filters on models due to its declarative visual nature, which uses the DSVL concrete syntax, although other languages could be used for the same purpose. An example is shown in Fig. 16(a).

Each input data is stored in the specified variable name, so that its content is available to be used in the property action as well as in subsequent filter's query patterns. In addition, the designer can specify a message to request the data to the user. Finally, a class named *Preprocessing* allows defining additional data requests or manipulations by using procedural code, which is given in its attribute *preaction*. Such code will be executed before the property action.

In order to provide the verification results obtained by a property to the final user, BaVeL defines five general output mechanisms. In particular, results can be printed out (class *OutputPrinter*), stored in a file selected by the user (*OutputFile*), shown in a dialog window (*OutputDialog*), expressed in terms of the original notation (*OutputBackannotation*), or in case the result is a file, it can be loaded in the default application for the file type, which for example allows showing a counterexample returned by a model checker (*OutputApplication*). Note that it is possible to define several sequential output mechanisms to provide different representations for the same results. In addition, class *Branch* allows selecting different annotation mechanisms depending on certain conditions of the result, following the style of decision branching in flowcharts. Other more specific output mechanisms, different from the presented ones, can be procedurally specified by using class *Postprocessing*.

The *OutputBackannotation* mechanism mentioned before includes the definition of a Triple Graphical Pattern (TGP) that states how a result in the semantic domain must be shown in the original model. A TGP consists of a triple graph called *positive* plus a number of application conditions made of a premise and a set of consequence triple graphs. The application of a TGP to a triple host graph results in all the subgraphs of the triple graph that fulfil the pattern. For this purpose, first all occurrences of the positive graph are sought in the host graph. Then, for each application condition whose premise is found in the graph, some of its consequences have to be found as well for the occurrence to be valid. There are two special application conditions called Negative and Positive Application Condition (NAC/PAC). A NAC has a premise but no consequence, and finding an occurrence of the premise makes invalid the occurrence of the positive graph. A PAC has no premise but only consequence graphs. In that case at least one occurrence of some of the consequences has to be found as well.

In addition, a TGP can be initialized with a partial match whose elements are given as *arguments*, and the output can be filtered according to the elements

identified by *output*. In our case, the arguments of the TGP are the elements in the semantic domain that we want to back-annotate (if any), and the output are the elements in the original domain resulting from the back-annotation.

Fig. 8 shows on the left a sample TGP. Its positive triple graph relates a VisMODLE state machine (upper part) and a Petri net in the semantic domain (lower part) through a correspondence graph. The pattern seeks two connected states belonging to the same state machine, and the related Petri net transition. The argument of the pattern is the element labelled 1 (the Petri net transition), and the output are the elements labelled 2 through 5. Thus, the pattern specifies that if some analysis returns a Petri net transition, then it is passed to the pattern as a parameter, and the related state machine transition, together with its adjacent states and the container state machine are the equivalent result in terms of the original notation.

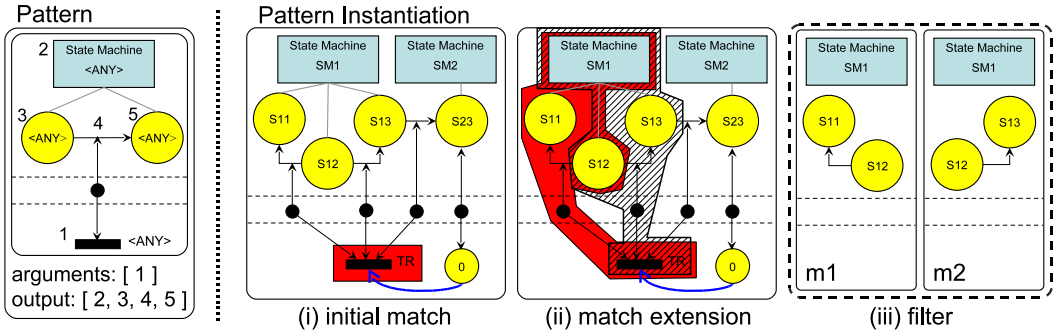


Fig. 8. Back-annotation triple graphical pattern and instantiation.

To the right of the same figure, the TGP is instantiated in a triple graph with two different state machines. In step (i) the match is initialized with the transition `TR` received as argument. In step (ii) the match is extended to the complete positive graph. In this case, two instances of the pattern are found. Note that both matches do not have to be disjoint, but they must be different. On the contrary, there is not a match containing the transition from `S13` to `S23` because its source and target states belong to different state machines. Finally, in step (iii) the matchings are filtered so that only the elements specified as output in the pattern are returned as result. In this way, we obtain the part of the state machine that should be returned as a result of the back-annotation. Should the analysis method returns more than one transition, the process is repeated for each one of them.

Note how TGPs overcome the 1-to-1 restriction for the back-annotation, since they provide the required flexibility to express arbitrary relations (even negative ones) among the elements in the source and target models. Moreover, by using TGPs we decouple the transformation into the semantic domain from the back-annotation process. In this way we can use the same transformation but several TGPs in order to back-annotate analysis results in different ways, depending on the particular property being verified.

BaVeL has been provided with the visual concrete syntax illustrated in Fig. 9. In this way, the properties to be analyzed in a semantic domain, together with the data input and output mechanisms, are gathered in a single diagram using an intuitive representation. In this syntax, properties are represented as black boxes. Pre- and post-processing actions are shown as clouds. The rest of inputs and outputs are visualized as icons similar to the elements or actions they represent: a graph where an element is being selected, a file, a text field, a funnel to represent the filtering action, a printer, a dialog window, a graph where an element is being highlighted to denote back-annotation, and an application-like window. Finally, decision points are represented as a diamond shape, where the different outgoing branches show the associated conditions to be evaluated.

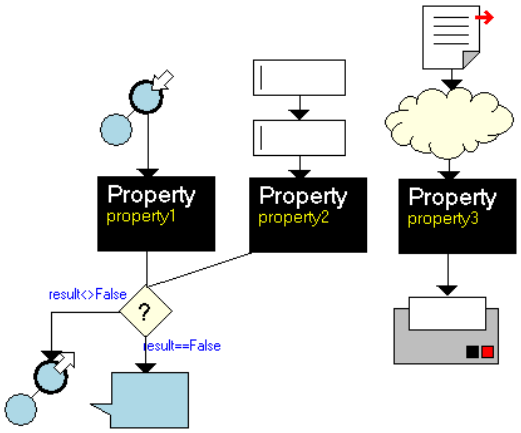


Fig. 9. BaVeL model example.

The BaVeL model example shown in Fig. 9 contains the definition of three properties named `property1`, `property2` and `property3` respectively. The first one requires the user to select a graph element before performing the analysis. If the analysis result is different from false, then it is annotated to the original model by a back-annotation TGP (left branch of the conditional point); otherwise, the result is summarized in a dialog window (right conditional branch). The second property requires the user to input two textual values, and defines the same annotation mechanism as the first property. Note that it is possible to reuse input and output elements so as to facilitate the definition and maintenance of a BaVeL specification. Finally, the third property requires the user to select a file and performs certain internal data pre-processing. In this case, the obtained result is printed.

6 Implementation of the Framework

The presented consistency framework has been implemented in the meta-modelling tool AToM^3 [21]. This tool allows the description of DSVLs by means of a meta-model, and their manipulation with graph transformation. In order to support the definition of MV-DSVLs and provide consistency of

system views [17], we extended the tool by using its meta-modelling and code-generation capabilities. In particular, we defined the meta-model shown in Fig. 10 with AToM^3, and this automatically generated a new modelling environment that allows defining viewpoints, semantic views and analysis method calls. The new environment was completed with hand-written code and integrated into AToM^3 itself.

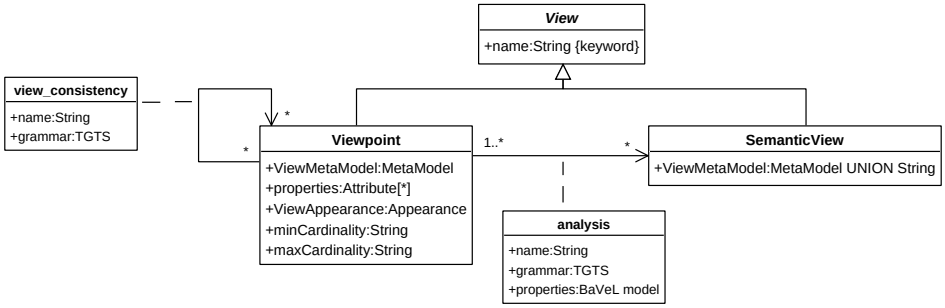


Fig. 10. Meta-model for multi-view domain specific visual languages.

As shown in Fig. 10, the meta-model for the new tool defines two kinds of views: viewpoints (class *Viewpoint*) and semantic views (class *SemanticView*). A viewpoint has a meta-model, a list of attributes (e.g. author, description, etc.), a visual appearance and a cardinality. The cardinality attributes are useful, for example, in order to specify that a certain diagram type is mandatory for a particular MV-DSVL. Viewpoints define consistency relations (*view_consistency* association) that contain the necessary TGTSs to provide syntactic consistency, as explained in Section 4.

Semantic views are used to specify a semantic domain by means of either its explicit meta-model, or by giving the name of an existing one. Viewpoints and semantic views are related through analysis relations with a name, a TGTS that transforms the former to the latter, and a BaVeL model that contains the set of properties to be analyzed together with the annotation mechanisms that will be used to show the obtained results in the most appropriate way.

A screenshot of the tool generated from this meta-model, being used to specify a MV-DSVL, is shown in Fig. 11.

7 Case Study: VisMODLE, MDD Approach for Digital Libraries

In this section we illustrate the previous concepts by generating an environment for VisMODLE, the notation presented in Section 2, by using AToM^3. In this tool, the MV-DSVL designer starts by giving the meta-model of the whole language. As an example, window "1" in Fig. 11 contains the full meta-model of VisMODLE. Next, the designer declares the diagram types. For example,

window “2” shows the definition of the five VisMODLE viewpoints in the newly developed tool. For each viewpoint, the portion of the complete meta-model that belongs to it has to be specified, as window “3” shows for the structural viewpoint. In order to ensure that the viewpoint meta-model is actually a submodel of the complete meta-model, the tool first presents the complete meta-model and the designer is allowed to delete classes, associations and attributes. Moreover, he is allowed to add extra constraints and change the visualization of classes and associations. As each viewpoint has its own meta-model, intra-diagram syntactic consistency is achieved as in the case of environments for single-view DSVLs.

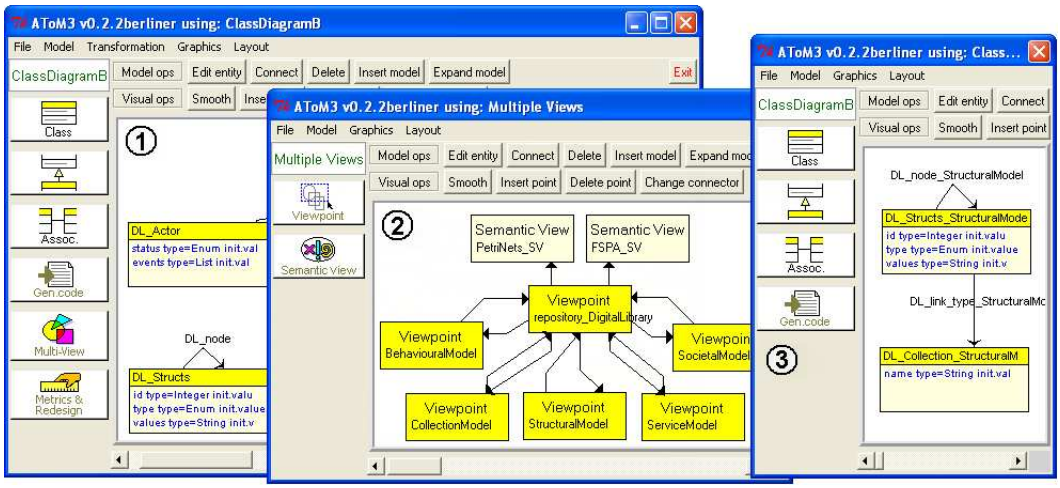


Fig. 11. Definition of the environment for VisMODLE.

A special viewpoint called `repository_DigitalLibrary` is automatically created and managed by AToM^3 (i.e. the designer is not allowed to change its meta-model or any of its properties). It contains the whole meta-model and is used for ensuring inter-diagram syntactic consistency as shown in Fig. 5. Consistency relations, shown as arrows between the repository and the viewpoints in window “2”, contain the TGTSs that specify how the view elements will be copied into the repository of the generated environment and will be kept consistent [17]. These rules are automatically generated from the meta-model information, but can be modified by the MV-DSVL designer.

In order to provide dynamic semantics consistency and analysis techniques to an environment, the new tool allows defining semantic views for the viewpoints and the repository. In the case of VisMODLE, we are interested in verifying certain properties such as: checking if an actor or service reaches a deadlock state (in the case of services, we normally want them always available, so they should not define final states); verifying that the DL is never blocked; detecting whether a given actor or service accomplishes a certain activity or sequence of tasks that make it reach a certain state, as well as which activities are never completed preventing the actor or service to reach a certain state (which can be considered a design error); checking which requests are made in

any possible execution flow; analysing if a service can receive an unbounded number of requests that could lead to overflow; or checking if the message exchange specified by the societal models allows the execution of the behaviour of the state machines.

Petri nets [26] provide analysis techniques that allow investigating system properties such as deadlocks, reachability of states and invariants. For this reason we have enriched the definition of VisMODLE with a semantic view called `PetriNets_SV` for its repository, as it is shown in window "2" of Fig. 11. The arrow from the repository to the semantic view contains the TGTS defined by the MV-DSVL designer for building the latter from the former, as well as the BaVeL model specifying the workflow of the verification process. In addition, we have defined a second semantic view called `FSPA_SV` for the analysis of the repository by using a finite state process algebra (FSPA) [23]. This formalism allows verifying additional properties such as refinement of processes, and is used to check if the society specification is consistent with the behaviour of the state machines.

The next two subsections describe the transformations from VisMODLE to Petri Nets and FSPA, the definition of the property analysis to be verified in each domain, and the BaVeL models with the mechanisms used to return the verification results. Recall that this information is provided by the MV-DSVL designer, but is completely transparent to the end user. It is worth emphasizing that the first case with Petri nets is an example of visual semantic domain, where the analysis is performed internally in AToM^3. On the contrary, FSPA is textual and we use an external analysis tool [20].

7.1 Analysis of VisMODLE by Petri Nets: An Example of Visual Semantic Domain and Internal Analysis Tool

Fig. 12 shows some triple rules to transform the VisMODLE repository into Petri nets. The transformation translates the state machines of each actor and service into what we call Petri net modules. For this purpose, the rule *State2Place* translates states into places labelled with the state's name. The rule *Transition2Transition* translates transitions between states into Petri net transitions labelled with a composition of the states's names and the event of the state machine transition [1]. The labelling is specified in the "action" section of the rules, which allows assigning attribute values.

The TGTS contains other triple rules, not shown in the paper, which complete the transformation as follows. Places associated to initial states are given one

[1] This heterogeneous mapping between a Petri net transition (a node) with a state machine transition (an edge) is allowed due to the theory we developed in [16].

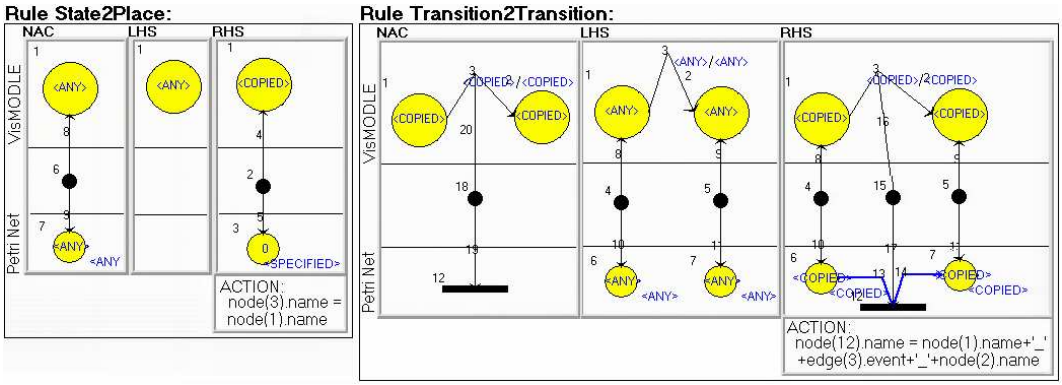


Fig. 12. Some triple rules for the transformation from VisMODLE to Petri nets.

token, zero otherwise. Besides, a place is created for each message an actor or service can invoke. These places are the interface of the Petri net module for the actor or service. In the case of synchronous services, one extra place is created for constraining the capacity of the interface place and thus ensuring at most one available message.

If a state machine transition needs an event for its execution, then a Petri net arc is created from the place corresponding to the event message to the Petri net transition corresponding to the state machine transition. For synchronous services, an arc is created from the associated constraining place to the transition as well. Similarly, if the execution of an event produces some action, an arc is created from the Petri net transition to the corresponding place of the behavioural element's interface specified in the action. Again, a capacity constraint place is generated for the synchronous case.

One of the advantages of using TGTSs is that it allows verifying properties of the transformation itself, such as termination or confluence. Studying the structure of the rules can prove termination [8]. For instance, our TGTS is terminating as the elements created by each rule are also NAC of these rules, therefore they will be only applied once for each initial match. Confluence can be studied by using critical pair analysis [8], which obtains the minimum graphs such that applying one rule disables another rule in the same grammar. The confluence of a graph grammar can be demonstrated by proving the confluence of all its critical pairs. In our TGTS we only detected conflicts of some rules with themselves (necessary, as we wanted the transformation to be terminating), which do not have a negative impact on the overall confluent behaviour of the transformation. Finally, syntactic correctness of the resulting model is guaranteed, as it has to be a valid instance of the target meta-model.

Fig. 13 shows the repository of the example university DL, and Fig. 14 depicts the resulting Petri after its transformation. Although not shown, both models are related through a correspondence graph created by the TGTS and used to back-annotate the analysis results. This resulting Petri net model is not usually

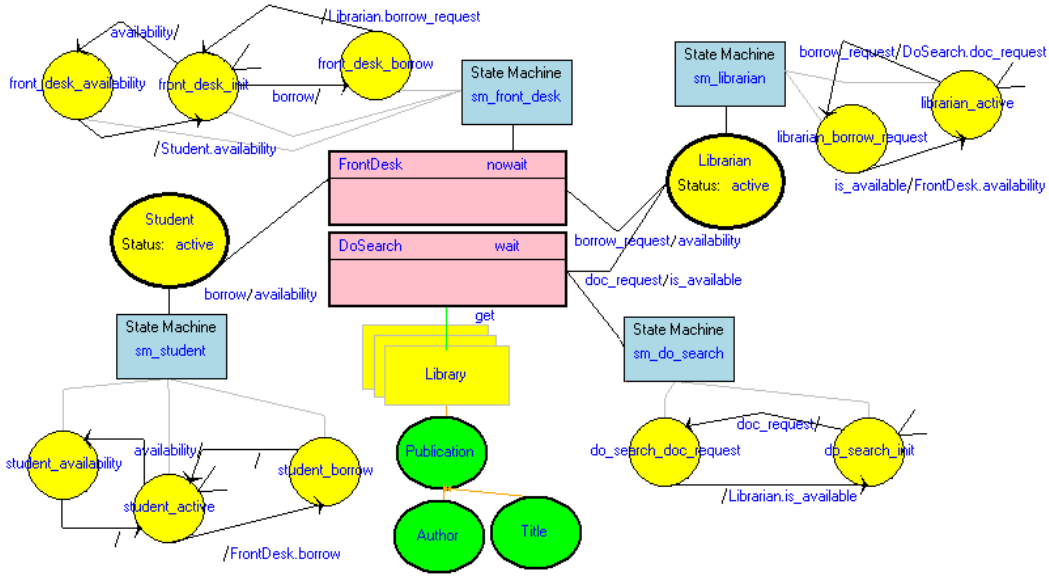


Fig. 13. Repository model for the DL example.

shown to the end-user, but used as an intermediate step for the verification of properties. Note also that the only synchronous communication is the one between the actor `Librarian` and the service `DoSearch`. This is reflected in the two places `DoSearch.doc_request` and `DoSearch.doc_request'` which ensure that at most one `doc_request` message is received by the service at a time.

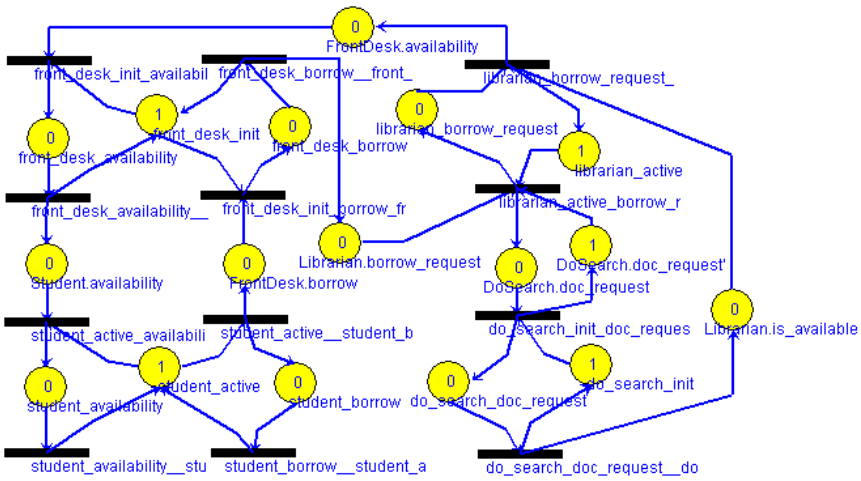


Fig. 14. Petri net resulting from the transformation.

Together with the previous TGTS, we specified a BaVeL model defining a set of properties for the verification of VisMODLE designs using Petri nets. The analysis is performed by calculating the coverability graph of the Petri net resulting from the transformation, and then applying model checking [6] on such graph to verify the system properties. These properties are expressed using Computational Tree Logic (CTL) [6]. The result of checking a property on a model is the set of states satisfying the given property.

name	**Behaviour Inaction (deadlock)**
description	A behavioural element *be* (actor or service) reaches a state that cannot be left.
action	evalExpression_states($\bigvee_i$ *A G (*s_i*)*), where s_i are the states of *be*.
name	**State Reachability**
description	A behavioural element can reach a state s as result of performing certain activity made of a sequence of tasks.
action	evalExpression_path(s).
name	**Unreachable States**
description	A behavioural element does not define unreachable states. This property allows detecting unnecessary states for an actor or service, as well as design errors.
action	evalExpression_states($be.s_i$), where s_i are the states defined by the element.
name	**Request Submission**
description	A request r is submitted to a behavioural element *be* in every possible execution flow. This allows testing whether a certain task is always performed.
action	evalExpression_states(*A True U be.r*). If the initial state does not belong to the result, a counterexample is generated with evalExpression_path(¬ (*E True U be.r*)).
name	**DL Availability**
description	The DL is always available, its execution is never blocked.
action	evalExpression_states(*E True U deadlock*). (Predicate *deadlock* is True in states with no successor)
name	**Request Overflow**
description	A service *se* can receive an unbounded number of requests leading to overflow.
action	evalExpression_states($\bigvee_i$(*se.*r_i*[w]*)), where r_i are the requests that *se* can receive. (Our coverability graph algorithm labels a place in a given state with "[w]" if it can receive an unbounded number of tokens).

Table 1
Property analysis for VisMODLE.

In general, we have identified four possibilities in order to show the result of a specific analysis from the Petri nets semantic domain. The first one is showing the resulting states in the original model by using the correspondence nodes created during the transformation, and making use of TGPs that specify which elements should be highlighted. Showing the result of an analysis may imply applying the pattern several times or applying several patterns. In addition, a means to navigate through the different analysis solutions is needed. The second possibility is testing whether a certain state, typically the initial state, belongs to this set in order to answer *true* or *false*. The third is reporting the result (e.g. a string, a matrix) in a window. Finally, we may want to show the execution flow from the initial state to a certain state satisfying a given property.

For VisMODLE, we defined a BaVeL model with the six properties of Table 1. The functions *evalExpression_states* and *evalExpression_path* return the set of net states that satisfy a CTL expression and the sequence of transitions that leads to a state satisfying it, using a model-checker implemented in AToM^3. The BaVeL model that defines all these properties is shown in Fig. 15.

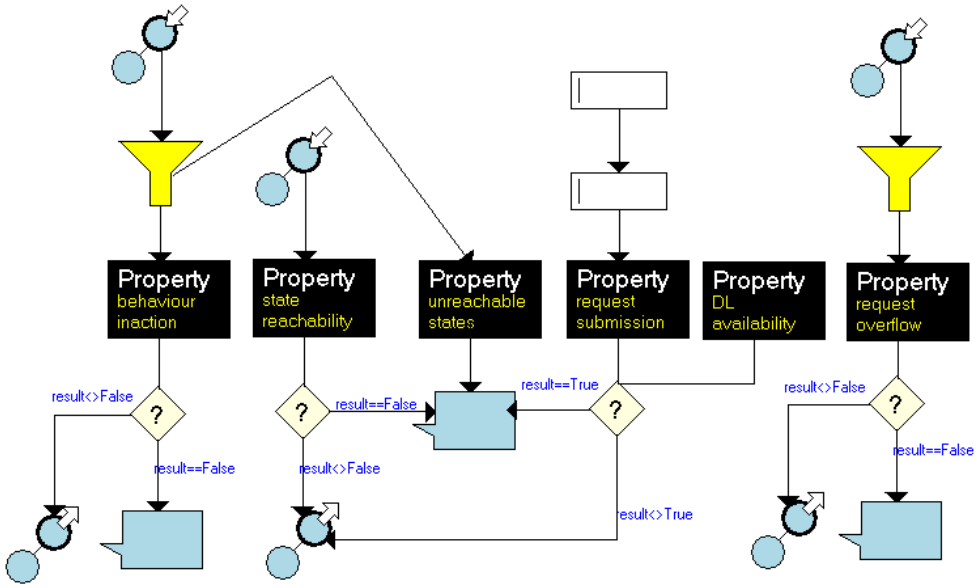
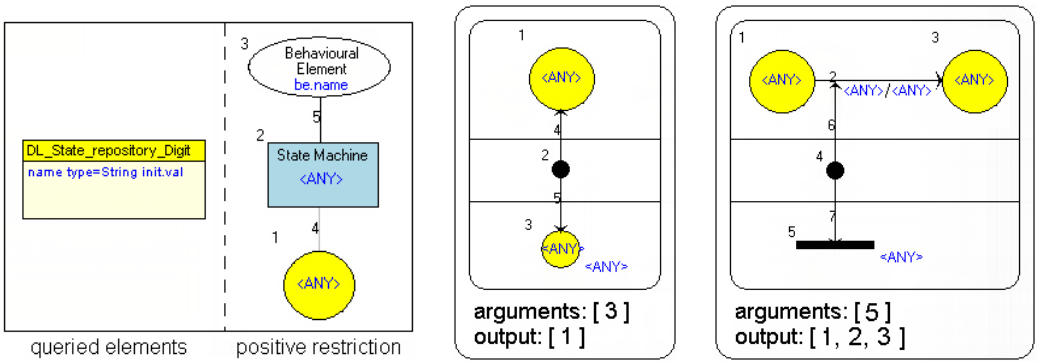


Fig. 15. BaVeL model for VisMODLE and Petri nets.

For example, property `Behaviour Inaction` in this model (the first one to the left) requires the user to select the actor or service `be` to analyze, thus an element of type *InputGraphElement* is specified as input. Then, as the CTL expression evaluated by the property uses the name of the element's states, an *InputFilter* node is included in order to obtain them. The filter contains the graph query pattern of Fig. 16(a), which obtains all states (given by the left meta-model) connected to the selected behavioural element (given by the positive restriction to the right). The restriction makes use of variable `be`, which was assigned to the previous input graph element in the model, and seeks all states from the state machine connected to the behavioural element with `be`'s name. Note that the same input elements are reused by property `Unreachable States` in the BaVeL model, being defined just once.



(a) Graph query pattern (b) TGP 1 (c) TGP 2

Fig. 16. (a) Filter and (b) back-annotation TGP for property `Behaviour Inaction`. (c) TGP for properties `State Reachability` and `Request Submission`.

The analysis of this property obtains partial deadlocks, i.e. all Petri net con-

figurations in which the given actor or service gets blocked. If some configuration is found, then it is reflected in the VisMODLE model (left branch of the decision point attached to the property in Fig. 15). The back-annotation is performed through the TGP of Fig. 16(b). If no block is detected, a dialog box is shown stating that the property is false (right branch of the same decision point).

For the annotation of analysis results of other VisMODLE properties we have used other back-annotation TGPs. In particular, properties `State Reachability` and `Request Submission` perform the annotation of analysis results by using the TGP shown in Fig. 16 (c), while property `Request Overflow` uses the same TGP but with single element 2 as output.

7.2 Analysis of VisMODLE by FSPA: An Example of Textual Semantic Domain and External Analysis Tool

As second semantic domain for the analysis of VisMODLE we have used the FSPA process algebra [23]. This algebra provides tools for analysing process refinement, which we have used in order to verify that societal diagrams are compatible with the aggregate behaviour of the state machines. Even though FSPA is a textual formalism, our framework can deal with it by providing a meta-model for its abstract syntax. In this way it is possible to define a TGTS to perform the transformation as in the previous case. In addition we had to build a code generator to synthesize the textual language.

Fig. 17 shows the meta-model we have designed for FSPA. It does not cover all the language, but only the fragment we needed to capture the VisMODLE semantics. FSPA is a notation for specifying concurrent systems through the aggregate behaviour of processes. A process has a definition of its behaviour (class *ProcessDefinition*), and is defined by the actions it engages in. Processes can be defined as a choice of several processes (class *Choice*), as well as be composed in a sequential or parallel way. In the latter case, processes must synchronize through their common actions. Moreover, we can specify renaming and hiding of actions, as well as arrays of processes (class *Range*).

For the analysis of VisMODLE, we have provided a TGTS that translates each state machine into a set of processes. Each state is mapped into a process, while each event and action in a state machine transition is transformed into an action prefix. If an asynchronous service or an actor owns the state machine, then a process emulating a buffer is created for each one of its possible incoming messages. The concurrent behaviour of the system is modelled by creating a process made of the parallel composition of each state machine. Then, the TGTS uses the connectivity and allowed input/output messages of

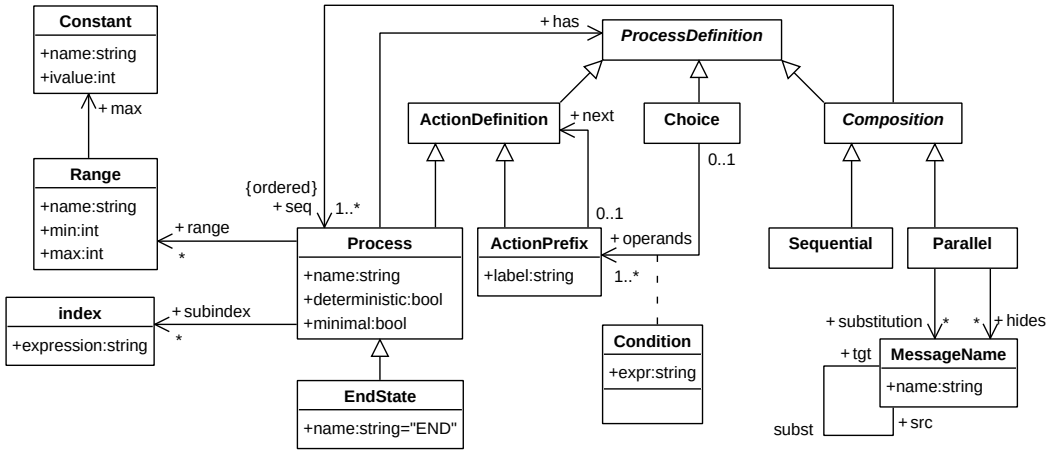
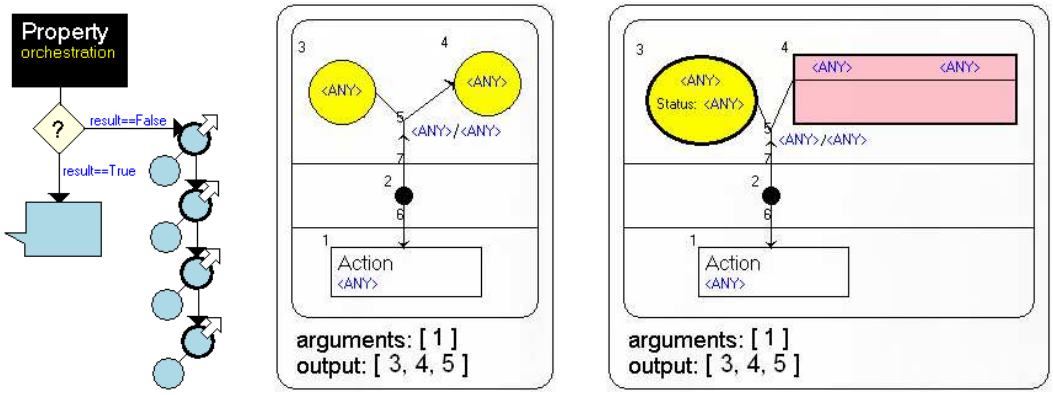


Fig. 17. Meta-model for a fragment of FSPA.

each service specified in the societal diagrams, to produce another set of concurrent processes. These processes are similar to a rudimentary web-service choreography description language [11].

From the FSPA model we generate code in a file and call the LTSA [20] tool by directly using its API. With this tool we check whether the parallel composition of the state machines and the choreography yields a process equivalent to the state machines. If this is so, the societal diagrams are compatible with the state machines. The tool can return either that the processes are equivalent, or a trace showing a failure (i.e. a sequence of prefix actions). In the latter case we highlight the state machine transitions and channels in the societal model that produced the conflict.



(a) BaVeL Model (b) 1^{st} TGP in (a) (c) 2^{nd} TGP in (a)

Fig. 18. Annotation of verification results in the semantic domain FSP.

Fig. 18(a) shows the BaVeL model for the verification using FSPA. It includes one property called `orchestration`, which uses four back-annotation TGPs to highlight the state machine transition, the channels from actors to services, and the two different kinds of connections between services. Two of the patterns are shown in Fig. 18(b) and (c). The first one highlights the state

machine transition associated with an action prefix, together with its source and target states. The second one signals the channel, the source actor and the target service associated with an action prefix.

7.3 Generated Modelling Environment for VisMODLE

Fig. 19 shows the automatically generated environment for VisMODLE. The window in the background shows several system views created by the end-user. The window on top contains a structural view with the structure of the collection of documents for the example DL. The tool automatically guarantees intra- and inter-diagram syntactic consistency as previously explained. Moreover, we built a code generator that synthesizes the user interface for the DL system from the VisMODLE specifications [24].

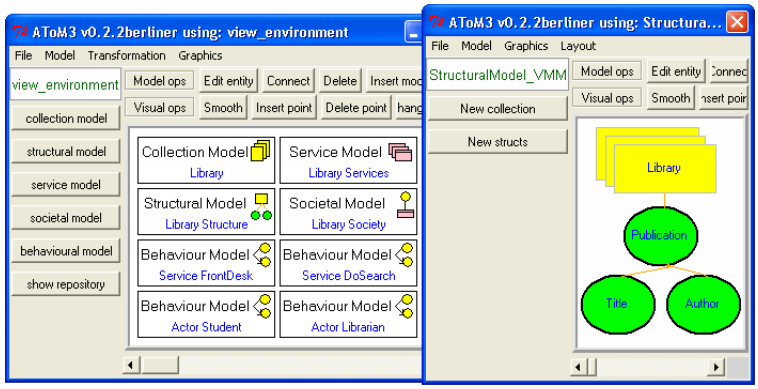


Fig. 19. Generated environment for VisMODLE.

The environment also provides the user with the analysis capabilities defined by the MV-DSVL designer. In the interface of the source viewpoint, a button is generated for each property specified in the BaVeL models. Thus, performing an analysis just implies clicking on the generated button, and the verification proceeds as specified in the BaVeL model, hiding the internals of the analysis process. If the annotation is specified by a TGP, the output elements obtained from its application are highlighted in the original model, as well as summarized in a dialog window. If an analysis returns several solutions, a navigator allows browsing through them. In addition, a button is generated that allows showing the result of executing the transformation. This can be used for debugging or simulation purposes.

Fig. 20 shows the repository interface of the generated VisMODLE environment. It includes one button for each defined property verification. In order to check a property, the end-user has to open the repository and click on one of these buttons. Internally, the tool performs the transformation to the semantic domain (Petri nets or FSPA), executes the analysis and returns the results. Fig. 20 shows the result obtained after executing the analysis called `state`

`reachability` for state `do_search_doc_request`. Such state was selected by the user by clicking on it (i.e. the input to the analysis was modelled as a graph element). If this state is reachable, it means that student requests can reach the search engine (`DoSearch` service). As in this case the selected state is reachable, the TGP shown in Fig. 18(c) is internally used to reflect the analysis results on the repository model elements. In the figure, the window to the right allows navigating through the different solutions (i.e. the different path executions) that lead to such state. The elements conforming a single solution are shown in the navigator and highlighted in the model. In this way, the analysis result is shown in terms of the original notation to the user, who does not need to have knowledge about any formal method.

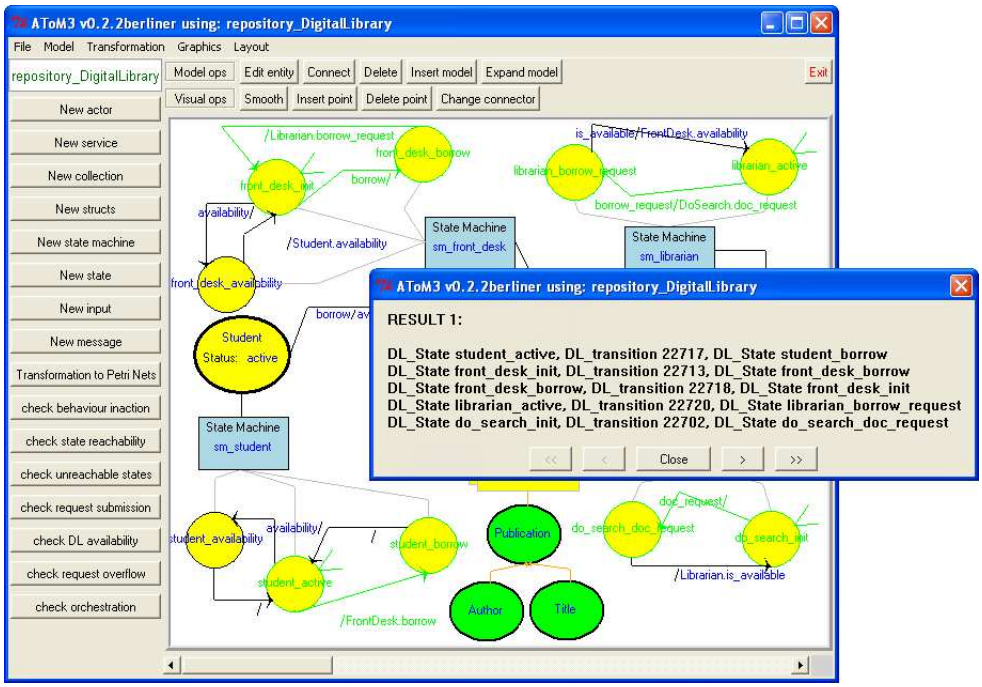


Fig. 20. Result of testing the reachability of state `do_search_doc_request`. The sequence of transitions leading to the state are shown highlighted and summarized in a dialog window.

For the example DL, the verification methods proved to be very useful in finding potential problems. The properties verified with Petri nets showed reachability of all states in every state machine, no global or partial deadlock (i.e. inaction of all or some behavioural element) and buffer overflow of the `FrontDesk` service. The latter is produced because students can send an arbitrary number of `borrow` messages to the `FrontDesk` service before obtaining a reply. On the other hand, changing the `FrontDesk` service to be synchronous produces a global deadlock. Interestingly, the fact that `DoSearch` is synchronous or not is irrelevant for deadlocks.

The analysis with FSPA showed that the aggregate behaviour of the state machines was not compatible with the societal model. This is because the latter

assumes that the student has to produce a `borrow` message before obtaining an `availability` response, while in the state machine this is not so. Thus, the system returned a trace made of just one action prefix, `availability`, pointing out an error in the behaviour of the student. After modifying it we obtained that the models were compatible.

8 Evaluation

Next we evaluate the presented approach from three different perspectives: the usability of the BaVeL notation, its expresiveness, and the usability of the generated environments.

Usability of BaVeL. We have tested it by using the Cognitive Dimensions framework [13]. The significant results are as follows.

Abstraction gradient. An abstraction is a grouping of elements to be treated as one entity. In this sense, BaVeL is abstraction-tolerant. It provides twelve high level abstractions of input and output processes. These abstractions are intuitive as they are visualized as the process they represent, and easy to learn as their configuration implies few simple attributes. Although BaVeL does not allow building new abstractions from scratch, new input and output ways can be coded in *Preprocessing* and *Postprocessing* elements.

Closeness of mapping. BaVeL elements have been assigned icons that resemble what they represent (e.g. printing is associated to a printer icon and a file is represented by a paper sheet). The elements that do not have a correspondence with a physical object in the real world have icons borrowed from well-known notations (e.g. conditionals are represented as diamonds, like in UML activities).

Consistency. A notation is consistent if a user knowing some of its structure can infer most of the rest. In BaVeL, when two elements represent the same entity but can be used either as input or as output, then their shape is equal but incorporates an incoming or an outgoing arrow in order to differentiate them. See e.g. the icons for input/output files, or those for graph nodes representing either a data input selection or the highlighting of an output result.

Diffuseness/Terseness. A notation is diffuse when many elements are needed to express one concept. BaVeL is terse and not diffuse, as each entity expresses a meaning on its own.

Error-proneness. Data flow visualization reduces the chance of errors at a first level of the specification. On the other hand, some mistakes can be unadvert-

edly introduced when specifying triple patterns, since it is possible to express relations between source and target models which are not created by the associated TGTS. However, these mistakes should be considered programming errors more than "slips", and may be detected through progressive evaluation.

Hidden dependencies. A hidden dependency is a relation between two elements that is not fully visible. In BaVeL, every dependency that matters to the user is visually represented as a data flow by means of directed arrows.

Progressive evaluation. Each analysis property can be tested once it is defined, it is not necessary to wait until the whole BaVeL model is finished. The evaluation is done by generating the visual environment for the DSVL (just clicking a button), which will include a widget to execute each analysis property.

Viscosity. BaVeL has a low viscosity because making small changes in a part of a specification does not imply lots of readjustments in the rest of it. Properties as well as input and output elements are encapsulated in separate objects. The only local changes that could imply performing further changes by hand are deleting input elements or changing its name; however this would imply minimal changes (just removing or updating references to them) and would only affect a small set of subsequent elements in the same data flow.

If the TGTS changes, no modification of the BaVeL specification is necessary, as far as the target domain remains the same. If the DSVL meta-model changes, we should update most TGPs, but this cannot be considered a small change.

Visibility. A BaVeL specification consists of a single diagram. Empirically, we have observed that this model usually involves no more than six or seven properties, which results in small models. Different, independent BaVeL models can be simultaneously shown in different windows.

Expressiveness of BaVeL. This paper has illustrated the expresiveness of BaVeL by defining different property analyses with different input and output requisites. For this purpose two different semantic domains have been considered, being one textual and the other one visual, and in addition we have shown the use of both built-in and external tools for the analysis. Moreover, our framework is general enough to be used with other DSVLs or even general-purpose languages such as UML [17], since it provides tools to specify languages and analysis rules in a formal way that does not depend on the features of the language. Indeed, we have applied this framework to a rather different MV-DSVL to analyze the specification of access policies for web systems [19] and thus verifying, at design time, some properties concerning security and web-content accessibility (e.g. that a specific web page is never shown to an unauthorized role).

Usability of the Generated Tools. The environments for single-view languages generated with AToM^3 have been extensively used, mostly in an academic setting, in different areas like software and web engineering, modelling and simulation, urban planning, etc. Concerning the multi-view and analysis support, the generated tools incorporate extra functionality which is accessible by clicking one button. However, depending on the kind of analysis, generating the results may take some time. For instance, the state reachability analysis in the DL example takes a few minutes. In general, from the application experience, we note the general agreement that automated syntactical consistency support greatly simplifies the design of complex systems. Finally, some users pointed out some technical limitations of the current implementation, such as the fact that it is not possible to open several views at a time.

Altogether, we believe this work contributes to make more efficient and less tedious the definition and maintenance of environments for MV-DSVLs. The analysis techniques embedded in the environments allow detecting errors at design time and fix them before they become more costly. Our meta-modelling-centric approach contrasts to the programming-centric approach of most CASE tools, where the language and the analysis tools are hard-coded so that whenever a modification has to be done (whether on the language or on the semantic domain) developers have to dive into the code. A meta-modelling-centric approach is more efficient since maintenance is at the model level, so that MV-DSVL designers do not need to be experts in the programming language in which the tool is implemented. Moreover, the intrinsic complexity of the formal specification of rules is hidden to the MV-DSVLs users. This efficiency of the MV-DSVL maintenance process is a key factor in immature domains where technology is still being developed, such as the web modelling domain. Since web implementation technologies are constantly evolving, modelling languages must be increased on a regular basis too.

9 Related Work

Approaches to model transformation for the analysis of systems by its translation into a semantic domain are frequent [8,16,31–33], but general approaches to back-annotate the results to the original notation are not so common. In [31] reference models are used to interrelate source and target models in a single graph. Reference models resemble our correspondence graphs, although the latter maintain the two graphs separated so that no additional structure is needed to maintain the mappings. Moreover, in [31] the back-annotation mechanism is not explicit, allowing only 1-to-1 back-annotations. This is not enough if a semantic element has to be annotated in several elements in the source graph, if the analysis returns a set of results to be consecutively shown to the user, or if the result cannot be presented in the source model (e.g. it is

a boolean value or it must be shown in an external tool).

Similarly, we can find different DSVLs targeted to the specification and generation of mappings between source and target concrete languages. Examples of such DSVLs are VML-G [14] for the bidirectional transformations between schemas, and FBM [14] for business integration. Our framework could even be implemented by using the standard model transformation language QVT [29] instead of TGTSs, as QVT creates traces between the source and target models as well. In any case, all these languages are useful for synchronization and change propagation, but lack a flexible mechanism (e.g. the BaVeL notation) in order to annotate target information when it cannot be reflected in the source model. We opted for TGTSs for several reasons: they have a formal basis, we can use concrete syntax in the rules which make them more intuitive (e.g. QVT rules use abstract syntax), it allows an explicit control of the mappings that is useful for complex transformations, and in the case of QVT, nowadays there are no tools that completely support its visual syntax.

There are some proposals to the simulation of DSVLs based on the operational semantics of a semantic domain. The approach in [10] supports animation, but no further analysis techniques are provided. The work in [2] couples the creation graph grammar rules of the source and target notations, and the simulation steps in the semantic domain are translated back to the original notation using a textual language. It is mentioned that results of more complex analysis (e.g. a path to a deadlock) can be shown in the original notation by firing the animation rules corresponding to the transition rules. Our back-annotation mechanism is easier to specify, as we rely on graphical, declarative patterns, and not in programming languages. We are also more flexible as: (i) we allow having more than one semantic domain, and use the most appropriate one to analyze different properties; (ii) the result of an analysis may be a set of objects, which are appropriately shown (e.g. if they are a set of states, we can navigate through the set) and (iii) we have a full-fledged DSVL for the specification of the verification workflow. In previous work [17], we also used animation in the repository for the visual validation of systems.

Regarding tools, many efforts can be found in integrating analysis methods in modelling tools. However, these are usually hard-coded, oriented to a specific source language, and based on the analysis in a specific semantic domain. The scarce tools that mention back-annotation either do not explain how it is performed, or the mechanism is hard-coded and only permits 1-to-1 back-annotation. For instance, HIDE [4] is an environment for the design and transformation-based validation of systems that, however, does not include an explicit mechanism for annotation of results. Another example is DEGAS [5], a framework for the analysis of UML models with annotation of results to UML. The design platform supporting the methodology interoperates with state-of-the-art UML modelling tools, playing the role of bridge

between these CASE tools and additional ones for the analysis. However, UML models must follow certain criteria in order to allow the analysis, and the transformation and annotation mechanisms are hard-coded with no possibility of extension. Xlinkit [27], a framework for checking static semantic consistency relations of distributed DOM trees, provides several tools for the visualization of hyperlinks between inconsistent elements in different documents (e.g. HTML reports, SVG graphics and stylesheets), and suggests to use the scripting languages provided by the CASE tools to annotate results to the very tools where the models were specified. The analysis of dynamic properties is not considered, and the visualization mechanisms are fixed. On the contrary, we have presented a flexible, general framework that includes explicit support for back-annotation. Moreover we allow the MV-DSVL designer to choose the most appropriate semantic domain for analysis, and provide easy-to-define, visual back-annotation mechanisms that generate usable analysis tools for the final user.

Much effort is being recently spent on tools and techniques that simplify the specification and generation of richer modelling tools for DSVLs. Though there are many approaches for the generation of tools, most of them are merely visual editors. However, the MDD approach needs more functional tools integrating quality control aspects such as formal verification. Tools like OpenArchitectureWare [32] are moving in this direction by integrating a number of additional tools helping in common MDD tasks, such as code generation, model transformation and reporting. Nonetheless, OpenArchitectureWare does not provide support for DSVLs or formal verification. The fact that some of these tools are integrated in the Eclipse framework [7] may facilitate the interoperability with further tools. However, it is our view that all these related tools have to be customized and tightly integrated for the given domain. Thus, we have proposed a meta-model centric approach where the environment is generated from the DSVL meta-model, together with additional domain-specific models for the functionalities to be generated. In this paper we have shown BaVeL for the specification of the verification workflow, but in previous work we developed another DSVL for the specification of metrics and redesigns [18].

Regarding meta-CASE tools, just a handful of them are able to guarantee the syntactic consistency between different models of the same system, usually through a common repository where these models are related. These relations are frequently expressed using textual notations (e.g. the GMF [12], MetaEdit+ [28] or Pounamu [34]), as well as partially graphical notations (e.g. GME [22] or JComposer [15]). Our approach is also based on the use of a repository, but we use high level, graphical, formal consistency mechanisms. To the best of our knowledge, there are no other tools providing high-level support to generate environments integrating hidden formal methods.

10 Conclusions and Future Work

In this paper we have described an approach that provides MV-DSVL designers with tools to visually specify rules for the syntactic and semantic validation of the different models making up a system, including the process of back-annotating the analysis results into the original notation. Verification of dynamic semantic properties is achieved by transforming the original model (the repository or a system view) to a semantic domain, and executing a set of analysis methods on it. In order to model the workflow of the verification process, as well as how the results of analyses should be shown back to the user, we have designed a DSVL called BaVeL. Among other back-annotation options, BaVeL allows defining how a verification result should be reflected in terms of the original notation, which is done by means of triple graphical patterns. We also presented the implementation of these concepts in AToM^3, and illustrated its use by building an environment for VisMODLE, a language for MDD in the area of Digital Libraries.

We are currently implementing additional analysis techniques based on Petri nets, such as algebraic and structural methods. We also plan to build a catalogue with the kind of properties that are frequently subject of investigation, so that the DSVL designers can easily reuse them for their integration in the environments under development. In this direction, it is also worth studying the use of DSVLs for letting the end-users in a certain domain specifying the properties to be verified. This contrasts with our current approach, where the DSVL designer fixes the verification properties.

In the short term, we are planning to improve the framework with the support of nesting when showing the back-annotation results (i.e. showing step-by-step all elements in each single solution). BaVeL could also be extended, for example, by adding special output nodes that allow transforming the results into other languages. This could be useful, e.g., to show error traces in the form of sequence diagrams. Finally, other interesting issue is to use the analysis results in order to guide the semi-automatic application of redesigns.

Acknowledgements. Work supported by the Spanish Ministry of Education and Science, projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN2006-09678). We thank the referees for their useful comments.

References

[1] ATL Home page. `http://www.sciences.univ-nantes.fr/lina/atl/`

[2] Baresi, L., Pezzé, M. Formal interpreters for diagram notations. *ACM*

Transactions on Software Engineering and Methodology 2005; 14(1):42-84.

[3] Berry, D. M. Formal methods: the very idea. Some thoughts about why they work when they work. *Science of Computer Programming* 2002; 42(1): 11-27.

[4] Bondavalli, A., Dal Cin, M., Latella, D., Pataricza, A. High-level integrated design environment for dependability (HIDE). Proc. of WORDS'99, 1999.

[5] Buchholtz, M., Gilmore, S., Haenel, V., Montangero, C. End-to-End Integrated Security and Performance Analysis on the DEGAS Choreographer Platform. Proc. of FM'05, vol. 3582 of LNCS, Springer, 2005, pp.: 286-301.

[6] Clarke, E. M., Grumberg, O., Long, D. *Model Checking*. The MIT Press, 2000.

[7] Eclipse Home page. `http://www.eclipse.org/`

[8] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science, Springer, 2006.

[9] Ehrig, H., Engels, G., Kreowski, H. J., Rozenberg, G. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1*. World Scientific, 1997.

[10] Ermel, C., Hölscher, K., Kuske, S., Ziemann, P. Animated simulation of integrated UML behavioral models based on graph transformation. Proc. of VL/HCC'05, IEEE CS Press, 2005, pp.: 125-133.

[11] Foster, H., Uchitel, S., Magee, J., Kramer, J. 2007. *WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography*. In Test and Analysis of Web Services, pp.: 87-119.

[12] GMF Home page. `http://www.eclipse.org/gmf`

[13] Green, T. R., and Petre, M. Usability analysis of visual programming environments: a "cognitive dimensions" framework. *Journal of Visual Languages and Computing* 1996; 7:131-174. Elsevier.

[14] Grundy, J. C., Hosking, J. G., Amor, R. W., Mugridge, W. B., Li, Y. Domain-specific visual languages for specifying and generating data mapping systems. *Journal of Visual Languages and Computing* 2004; 15(3-4):243-263. Elsevier.

[15] Grundy, J. C., Mugridge, W. B., Hosking, J. G. *Visual specification of multiview visual environments*. Proc. of VL'98, IEEE CS Press, 1998, pp.: 236-243.

[16] Guerra, E., de Lara, J. *Model view management with triple graph transformation systems*. Proc. of ICGT'06, vol. 4178 of LNCS, Springer, 2006, pp.: 351-366.

[17] Guerra, E., de Lara, J. *Meta-modelling and graph transformation for the definition of multi-view visual languages*. Chapter of the book *Visual Languages for Interactive Computing: Definitions and Formalization*, Idea Group, 2007. Edited by Fernando Ferri.

[18] Guerra, E., de Lara, J., Díaz, P. Visual specification of measurements and redesigns for domain specific visual languages. *Journal of Visual Languages and Computing* 2008; 19(3):399-425. Elsevier.

[19] Guerra, E., Sanz, D., Díaz, P., Aedo, I. A transformation-driven approach to the verification of security policies in web designs. Proc. of ICWE'07, vol. 4607 of LNCS, Springer, 2007, pp.: 269-284.

[20] LTSA Home page. `http://www.doc.ic.ac.uk/ltsa/`

[21] de Lara, J., Vangheluwe, H. *AToM^3*: A tool for multi-formalism modelling and meta-modelling. Proc. of FASE'02, vol. 2306 of LNCS, Springer, 2002, pp.: 174-188. See also the AToM^3 home page at: `http://atom3.cs.mcgill.ca`

[22] Lédeczi, A., Bakay, A., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J. and Karsai, G. *Composing domain-specific design environments*, Computer, 34 (2001), pp.: 44-51.

[23] Magee, J., Kramer, J. 1999. *Concurrency: State Models & Java Programs.* John Wiley & Sons.

[24] Malizia, A., Guerra, E., de Lara, J. Model-driven development of digital libraries: Generating the user interface. Proc. of MDDAUI'06, Vol. 214 of CEUR, 2006.

[25] Manna, Z., Pnueli, A. 1990. *A hierarchy of temporal properties*, Proc. of PODC '90, ACM Press, pp. 377-410.

[26] Murata, T. Petri nets: Properties, analysis and applications. Proc. of IEEE 1989; 77(4):541-580.

[27] Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology* 2003; 12(1):28-63.

[28] Pohjonen, R., Tolvanen, J. P. Automated production of family members: Lessons learned. Proc. of the International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing, 2002, pp.: 49-57.

[29] QVT Specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`

[30] Schürr, A. Specification of graph translators with triple graph grammars. Vol. 903 of LNCS, Springer, 1994, pp.: 151-163.

[31] Varró, D., Varró, G., Pataricza, A. Designing the automatic transformation of visual languages. *Science of Computer Programming* 2002; 44(2):205-227. Elsevier.

[32] Völter, M., Stahl T. *Model-Driven Software Development.* Wiley, 2006.

[33] Xie, F., Levin, V., Browne, J. C. ObjectCheck: A model checking tool for executable object-oriented software system designs. Proc. of FASE'02, vol. 2306 of LNCS, Springer, 2002, pp.: 331-335.

[34] Zhu, N., Grundy, J. C., and Hosking, J. G. Pounamu: a meta-tool for multi-view visual language environment construction. Proc. of VL/HCC'04, IEEE CS Press, 2004, pp.: 254-256.