MODEL DRIVEN DEVELOPMENT OF DIGITAL LIBRARIES
Validation, Analysis and Code Generation

Esther Guerra
Dep. Computer Science, Universidad Carlos III, Madrid (Spain)

Juan de Lara
Dep. Computer Science, Universidad Autónoma, Madrid (Spain)

Alessio Malizia
Dep. Computer Science, University La Sapienza, Rome (Italy)

Keywords: Digital Library, Model Driven Development, Formal Methods, Meta-modelling, Graph Transformation.

Abstract: This paper shows our model-driven approach for the formal construction and validation of Digital Libraries (DLs). We have defined a Domain Specific Visual Language (DSVL) called VisMODLE, which allows the description of a DL using five different viewpoints: services, behaviour, collections, structure and society. From a meta-model based description of the different viewpoints, we have generated a modelling environment for VisMODLE. We have provided the environment with a code generator that produces XUL code for the DL's user interface and composes the application using predefined components that implement the different services. Moreover, we have also added validation and simulation capabilities to the environment. Using the behavioural models (state-machine based), we can visually animate the system. In addition, the combined behaviour of actors and services can be transformed into a Petri net for further analysis.

1 INTRODUCTION

The concept of Digital Library (DL) seems hard to be completely understood and evades definitional consensus. Licklider (Licklider, 1965) visualized a collection of digital versions of the worldwide corpus of published literature and its availability through interconnected computers. More recently, a Delphi study (Kochtanek and Hein, 1999) of DLs coalesced a broad definition: organized collection of resources, mechanisms for browsing and searching, distributed networked environments, and sets of services objectified to meet users' needs. The Presidents Information Technology Advisory Committee (PITAC) Panel on DLs treats them as the networked collections of digital texts, documents, images, sounds, scientific data and software, that make up the core of today's Internet and tomorrow's universally accessible digital repositories of human knowledge (Reddy and Wladawsky-Berger, 2001). Underlying these definitions there is the consensus agreement that DLs are fundamentally complex due to its inherently interdisciplinary nature. They are usually built from scratch using specialized architectures that do not benefit from previous DLs and software design experiences. The lack of formal models leads to branching efforts and has made interoperability (in both metadata and software levels) a crucial problem in the DL field.

From the beginning of computer science, software engineers have sought methods to increase the productivity and quality of applications. A means to achieve such goal is to increase the level of abstraction of system descriptions. In Model-Driven Software Development (MDSD) (Völter and Stahl, 2006), models are the primary asset, from which the code of the application is generated. The idea is to capitalize the knowledge in a certain application domain by providing developers with DSVLs describing the domain concepts. DSVLs are less error-prone than other general-purpose languages and easier to learn because the semantic gap between the user's mental model and the real model is smaller. Thus, from high-level, possibly visual models, a number of platform artifacts are generated which sometimes account for the 100% of the final application. Since the code generation process is automated from the models, there is

Guerra E., de Lara J. and Malizia A. (2007).
MODEL DRIVEN DEVELOPMENT OF DIGITAL LIBRARIES - Validation, Analysis and Code Generation.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Web Interfaces and Applications*, pages 35-42
DOI: 10.5220/0001265900350042

a strong need to validate and verify them. In this respect, techniques for simulating the models (i.e. performing an animation showing their operational semantics) as well as to transform them into formal semantic domains for further analysis are of great interest.

Our work applies MDSD techniques to the DL domain. In (Malizia et al., 2006) we presented a DSVL called VisMODLE oriented to the description of DLs, and built a modelling tool for it with an integrated code generator. VisMODLE is visual, which makes easier for people to learn and interpret; it is domain-specific, leaving less room for misunderstanding; and is formal, making possible the simulation and validation of the models. It allows the specification of the different aspects of a DL using four dimensions (or diagrams): services, collections, structure and society (i.e. interactions of services and actors).

In this work, VisMODLE is extended with a new type of diagram to specify behaviour (based on state machines). The operational semantics (i.e. simulator) of the new diagram type is formalized by using graph transformation (Ehrig et al., 2006). This simulator allows a visual animation of the models (i.e. seeing messages being produced/consumed by actors and services) in order to validate the DL design and understand its behaviour. We have also designed a transformation from VisMODLE to Petri nets (Murata, 1989), a formal notation useful to specify concurrent and distributed systems. Its analysis techniques allow investigating system properties such as deadlocks, state reachability and invariants.

Paper organization. Section 2 introduces VisMODLE. Section 3 shows our MDSD approach for generating DLs. Sections 4 and 5 describe the simulation and analysis techniques we propose, and its application to VisMODLE. Section 6 presents related work, and section 7 ends with the conclusions and future work. As a running example, we develop a simple DL for a university library.

2 VISMODLE, A DSVL FOR DIGITAL LIBRARIES

VisMODLE is a DSVL for the Visual MOdelling of Digital Library Environments. A preliminary version of the language was presented at (Malizia et al., 2006), but we have added a new diagram type for expressing behaviour. In this way, the specification of a DL in VisMODLE encompasses now five complementary dimensions: multimedia information supported by the DL (Collection Model); how that information is structured and organized (Structural

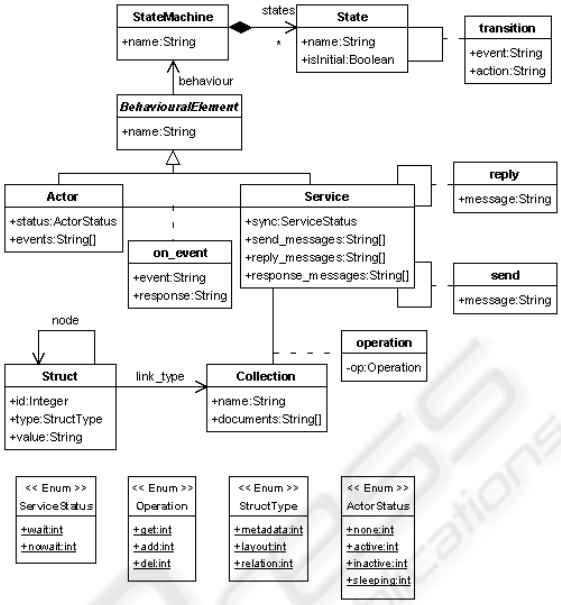

Figure 1: The VisMODLE Meta-model.

Model); the services of the DL (Service Model); the societies of actors and services that interact to carry out the DL behaviour (Societal Model); and the individual behaviour of actors and services (Behavioural Model). Figure 1 shows the complete meta-model.

Collections are sets of static (e.g. text) and dynamic (e.g. video) elements. Our running example includes a collection model (not shown for space constraints) with a collection (called *Library*) of two documents: long1.pdf and long2.pdf. The only relevant entity for this kind of diagram in the meta-model is class *Collection*.

The *Structural* diagram specifies how parts of a whole are arranged or organized. Structures can represent hypertexts, taxonomies, system connections, user relationships and containment. The window to the right in Figure 2 shows the structural model for the running example. Thus, collection *Library* is made of documents structured with *Publication*, *Author* and *Title* metadata information. The meta-model specifies that metadata entities (class *Struct*) can be connected together with the *node* relation (organized as a tree) and linked to a collection by a *link_type* relation.

Services describe activities, tasks, and operations (the functionality). Human information needs, and the process of satisfying them in the context of DLs, are well suited to description with services, including fact-finding, learning, gathering and exploring. In this diagram type, we describe the kind of messages services can produce and consume, and their synchronization. The only relevant entity from the meta-

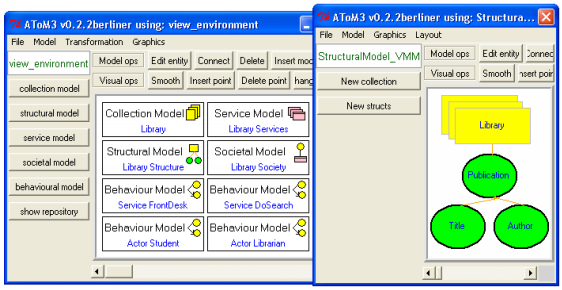

Figure 2: Modelling Environment for VisMODLE.

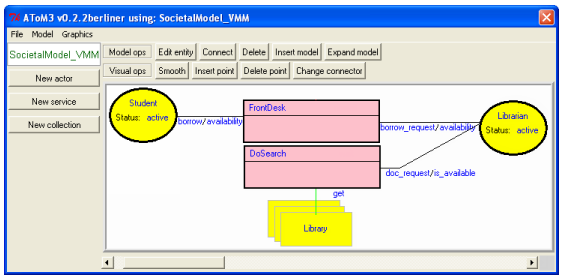

Figure 3: A *Societal* Model.

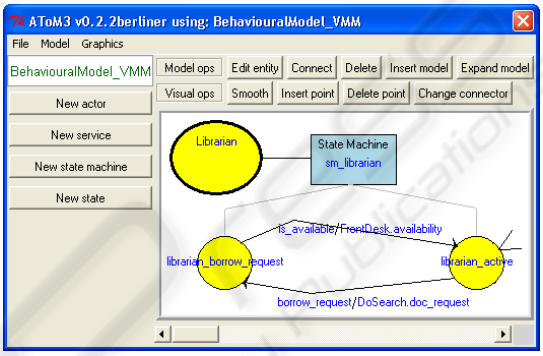

Figure 4: A *Behavioural* Model.

model for this diagram is class *Service*. In our example, we make two services available: *FrontDesk* and *DoSearch*. The former is responsible for managing communications between actors and is asynchronous (*sync* attribute is set to *nowait*), while the latter executes queries on the DL and is synchronous.

A *Society* is a set of entities and their relations. The entities include actors as well as hardware and software components, which either use or support services. Our meta-model is CSCW (Computer Supported Cooperative Work) oriented and thus includes the explicit description of the communication between entities and services. In complex systems such as DLs it is important to consider also the actors involved in the usage and so we introduced the society concept. A society is the highest-level view of a DL, which exists to serve the information needs of its entities and to describe the context of its use. *Society* diagrams are similar to UML collaboration diagrams. Figure 3 shows the society model for our example involving actors *Student* and *Librarian*. The scenario represents a *Student* borrowing a paper from the *Library*; he interacts with the *FrontDesk* service requesting the paper and obtaining a response message about its availability. The *FrontDesk* service forwards the borrow request to the *Librarian* actor. Then it sends a doc request message to the *DoSearch* service, which queries the document collection (*get* operation) using metadata information provided by the borrow request, and waits the result to send back the response. The service returns an *is_available* boolean message which is propagated as a response to the *Librarian* and eventually to the *Student*.

Behavioural diagrams are used to specify the individual high-level activity of both services and actors by means of state machines. The transitions of the state machines are fired whenever a certain event occurs, which corresponds to the arrival of a message to the actor or service. In the transition, an action can also be specified, which is the sending of another message. Figure 4 shows the behavioural diagram for the *Librarian* (initially in state *active*). If he receives an event *borrow_request*, he sends a *doc_request* message to service *DoSearch* and changes his state to *borrow_request*. Then, if he receives an event *is_available*, he becomes *active* again and sends the *availability* to service *FrontDesk*. For the current DL example, we have defined four different behavioural models, for actors *Student* and *Librarian* as well as for services *FrontDesk* and *DoSearch*.

3 MODEL DRIVEN APPROACH TO DIGITAL LIBRARIES

The overall architecture of our MDSD approach for DLs is shown in Figure 5. The upper part depicts the process of designing the DSVL VisMODLE with the help of experts in the field of DLs, and its implementation in the meta-modelling tool AToM^3.

The environment for VisMODLE was defined by using the meta-modelling capabilities of AToM^3. Figure 6 shows a step in its definition. The window at the background (labelled “1”) partially shows the complete VisMODLE meta-model. The tool allows splitting the meta-model into different diagram types (called *viewpoints*). The five VisMODLE viewpoints are shown in the window labelled “2” in the figure.

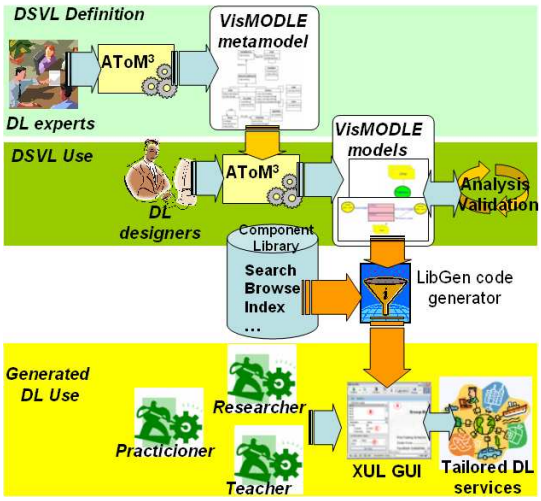

Figure 5: The MDSD Architecture for DLs.

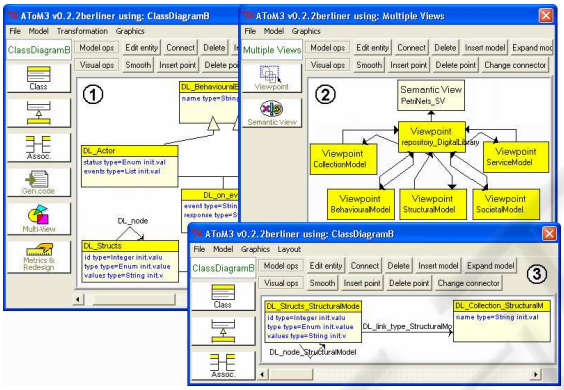

Figure 6: Building the Environment for VisMODLE.

Window "3" contains the portion of the meta-model belonging to the *Structural* viewpoint. A special viewpoint named *repository_DigitalLibrary* contains the complete meta-model. In this way, in the generated environment, the user builds instances of the different viewpoints, and a repository model is created in the background with the gluing of the different diagrams the user has built. Consistency relations (shown as arrows in window "2") specify how the different diagram elements are copied into the repository, how the different diagrams are kept consistent and how changes are propagated to the other diagrams if necessary. These arrows contain triple graph grammar (TGG) rules, which are automatically generated by AToM^3 (Guerra and de Lara, 2006b). In addition, a semantic view has been defined that allows expressing the semantics of the repository by using a transformation into Petri nets (see section 5). This is also performed by using TGG rules.

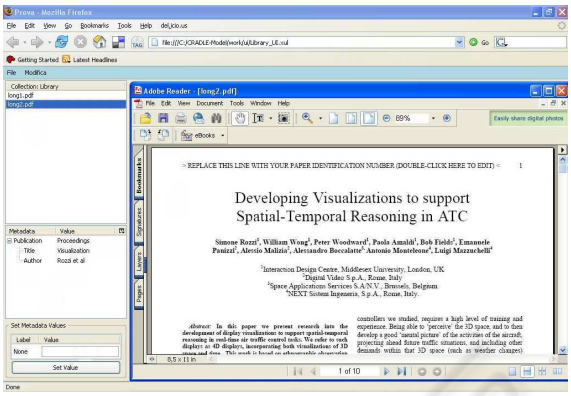

Figure 7: Generated GUI for the Example DL.

With the information described before, AToM^3 generated a customized modelling environment for VisMODLE (shown at the background of Figure 2). DL designers can use this environment to build VisMODLE models (layer "DSVL use" in Figure 5). The environment also integrates a simulator that allows the visual animation of the models. The simulator helps to validate and understand the DL design by observing the message flow between services and actors (see section 4). As stated before, some analysis capabilities have been also integrated based on model transformation into Petri nets (see section 5).

Finally, we have provided the environment with a code generator (called "LibGen" in Figure 5) able to produce XUL code for the user interface (UI) of the DL. The generator also selects predefined service components to implement the required functionalities of the VisMODLE models (Malizia et al., 2006). Figure 7 shows the generated interface for the example (for the *Librarian*). The generated UI is built upon a set of XUL template files that are automatically specialized depending on the attributes and relationships designed in the modelling phase. The layout template for the UI is divided into two columns. The left part manages the collections of documents and its metadata information. The right part manages visualization and multimedia information obtained from documents. The basic features provided with the UI templates are document loading and visualization, metadata organization and management.

4 SIMULATION OF DL MODELS

In order to visualize and better understand the behaviour of a DL, we have built a simulator for VisMODLE. The simulation is performed in the repository, since it needs all the dynamic information expressed

in VisMODLE (i.e. the societal and behavioural models). The simulation consists of the animation of the state machines that describe the behaviour of actors and services. In the simulation, it is checked that the events and actions specified in the state machines are coherent with the message interchange specified in the society. Note how simulation is a useful tool for the early detection of design errors.

The repository has been enriched with special elements used only for simulation purposes, but not for system specification. These elements are not part of VisMODLE; therefore, they do not appear in any diagram type. Thus, behavioural elements (i.e. actors and services) can receive *messages* through a special element called *input*, which in addition has a pointer to the current state of their state machine. Messages are depicted as blue rectangles with its name inside, while input elements are shown as black small rectangles. In the simulator, for a transition to be executed, a message matching the event specified in the transition has to be found in the behavioural element's input. The other possibility is that the transition does not require any event. On the other hand, when a transition that defines an action with form "target_element.message" is executed, then such message is created in the target element's input.

Figure 8 shows the DL example being simulated. A student has asked for a book, so he is in state *student_borrow* (i.e. the input for actor *Student* has a pointer to the *student_borrow* state). That means that the transition going from state *student_active* to the current state has been executed, so a message *borrow* has been sent to service *FrontDesk*, as the action of such transition indicates. Indeed, the state machine for service *FrontDesk* (upper left corner) has a message *borrow* as input. The next simulation step would execute the transition going from the service's current state to state *front_desk_borrow*, since the transition requires an event *borrow*, available in the input.

This simulator has been built by using the graph transformation capabilities of AToM^3. Graph transformation (Ehrig et al., 2006) is an abstract, declarative, visual, formal and high-level means to express computations on graphs. Roughly, graph grammars are composed of rules, with graphs in their left and right hand sides (LHS and RHS respectively). In order to apply a rule to a graph (called *host graph*), first a matching (an occurrence) of the LHS has to be found on it. Then, the rule is applied by substituting the match in the host graph by the rule's RHS. In addition, rules can define application conditions that restrict their applicability. One of the most used are the so-called Negative Application Conditions (NACs). These are graphs that must not be present in the host

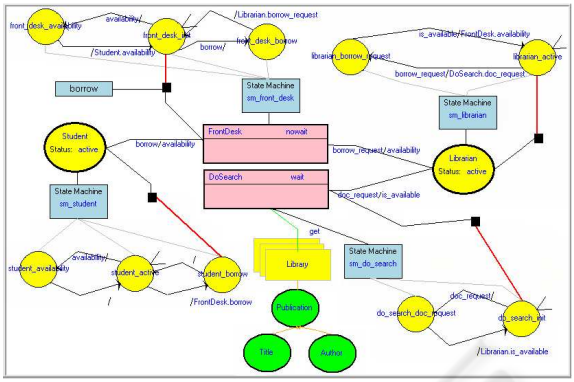

Figure 8: A Step in the Simulation of the Example DL.

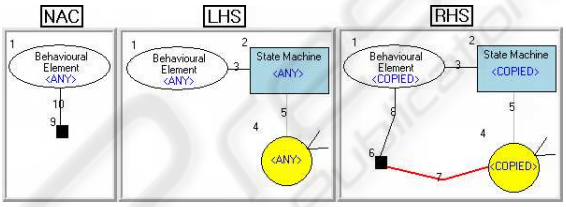

Figure 9: Simulation Rule *create input*.

graph for the rule to be applied. On the contrary, Positive Application Conditions (PACs) are graphs that must be present in the host graph in order to apply the rule. Finally, we can combine meta-modelling and graph transformation allowing abstract nodes to appear in rules (Ehrig et al., 2006). In this way, nodes can be matched to instances of any subclass, greatly improving the expressive power of rules.

Our simulator is made of five rules, three of them shown in Figures 9, 10 and 11. Rule in Figure 9 assigns an *input* element to each behavioural element, that is, to each actor and service in the repository (i.e. each possible concrete subclass of class *BehaviouralElement*). Initially, the input does not contain any message, and points to the initial state of the behavioural element's state machine. If the behavioural element already has an associated input element (NAC), the rule is not applied. In this way, we are sure there is only one input for service and actor.

The other four simulation rules perform a simulation step (i.e. a state transition if the required event was produced) for different cases. For example, the rule in Figure 10 considers transitions where no action is defined. In this case, if a behavioural element receives a message whose label matches some of its outgoing transitions (LHS), then the message is processed and the current state is changed (RHS). The attribute condition in the LHS checks that the message name in the input is equal to the event specified

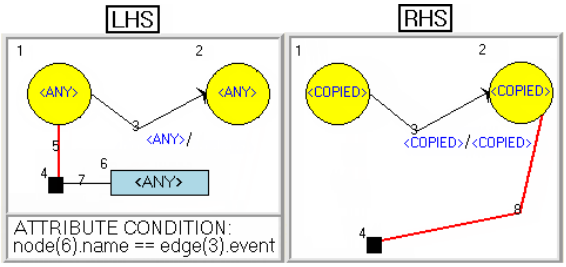

Figure 10: Simulation Rule *process event*.

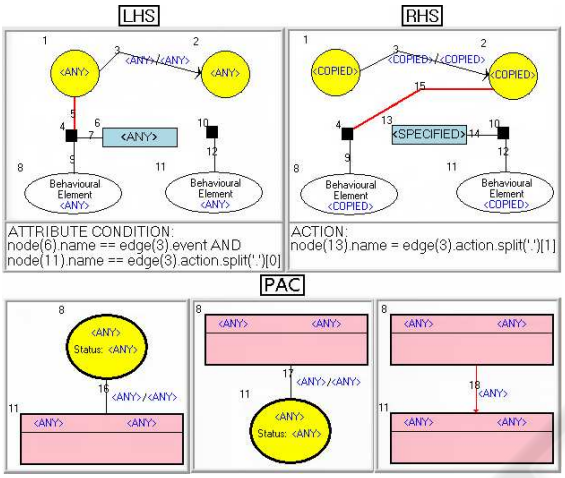

Figure 11: Simulation Rule *process event with action*.

in the outgoing transition of the current state.

Similarly, rule in Figure 11 considers transitions with an action that sends a message to another behavioural element (i.e. the action has the form "target_element.message"). Here, in addition to the previous conditions for a behavioural element to change its state, it is necessary that the target element accepts messages of that kind. This is given by the PAC below, which checks that the corresponding link has been specified in the societal model. Indeed, one of these three positive conditions has to be fulfilled in order to be able to apply the rule: either the source element is an actor that sends a message to a service (first case), or it is a service that sends a message either to an actor (second case) or to another service (third case). The use of abstract nodes (labels 8 and 11) together with the three PACs allows using a single rule for the three cases. This rule deletes the message from the input, changes the current state and creates a message in the target element input. The action below the RHS assigns the message specified in the action's transition as the name of the newly created message.

Finally, two similar rules (not shown in the paper) consider transitions where either an action is performed without the necessity of an event, or a transition occurs with neither event nor action associated.

5 ANALYSIS OF DL MODELS

We have provided the VisMODLE environment with some analysis mechanisms to validate the dynamics of a DL specification. More in detail, given the set of models that conform a DL design, we can:

1. check if some behavioural element reaches a deadlock state (i.e. local deadlock). In the case of services, we normally want to keep them always available, so they should not define final states. Final states are shown highlighted in the model as result of the analysis, and they are presented textually in a dialog window as well.
2. check whether a given behavioural element reaches certain state. The sequence of state transitions leading to the requested state is shown highlighted as a result, and also textually.
3. ask if a given message is always sent in any possible execution flow. The answer (*true* or *false*) is shown in a dialog window.
4. check if the system execution always finishes (i.e. global deadlock).
5. detect if there is some state for a behavioural element that is never reached in the given society. This may be considered a design error, since we should not define states that are not possible.
6. check if some behavioural element can receive an unbounded number of messages at some point, which could lead to an overflow.

In order to provide these analysis mechanisms, we have expressed the operational semantics of the models by means of Petri nets (Murata, 1989) (using a semantic view, see Figure 6). Thus, we have defined a triple graph transformation system (TGTS) (Guerra and de Lara, 2006a) that transforms the repository into the equivalent Place/Transition Petri net. Once the net is obtained, we internally use analysis techniques based on the reachability/coverability graph, as well as model-checking (de Lara et al., 2003). Similarly to graph grammars, TGTSs are made of rules that contain, in this case, triple graphs (instead of simple ones). Triple graphs are made of three different graphs: source, target and correspondence. In our case, the source graph is the repository, the target graph is the Petri net resulting from the transformation, while the correspondence graph contains elements relating the elements in the other two graphs.

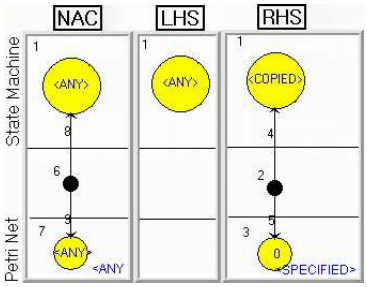

Figure 12: Triple Rule *SM state to PN place*.

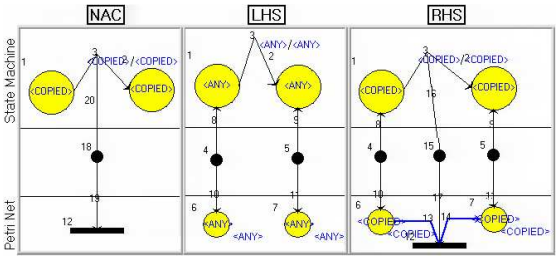

Figure 13: Triple Rule *SM transition to PN transition*.

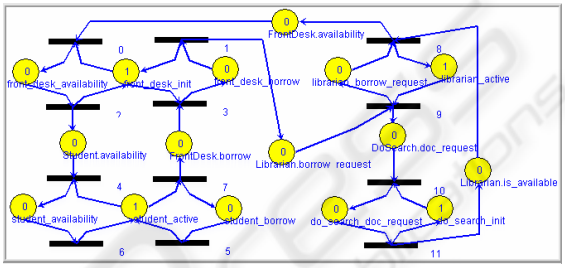

Figure 14: Place/Transition Petri Net for the DL Example.

Figures 12 and 13 show some rules of the TGTS that build the Place/Transition Petri net from the VisMODLE repository. The idea of the transformation is transforming the state machines of each behavioural element into what we call Petri net modules. With this purpose, states are translated into places (triple rule shown in Figure 12) and transitions between states are translated into transitions between places (triple rule in Figure 13). If a state is initial, then the associated place will contain one token. In other case, the place will be empty. In addition, an extra place is created for each possible message invocation (i.e. for each possible event and action specified in each state machine). These places are the *interface* of the Petri net module of the behavioural element. If a transition needs an event for being executed, then an arc is created from the place corresponding to the event message, to the Petri net transition corresponding to the state machine transition. In that case, the Petri net transition can be executed only if an event of that type is received (i.e. a token is in the right interface place), and in addition, the behavioural element is in the right state (i.e. a token exists in the place corresponding to the state source of the transition). Similarly, if a transition specifies an action, an arc is created from the Petri net transition to the right place of the behavioural element's interface specified in the action. In this way, executing the transition implies creating a token in the right interface place, and results in the interconnection of the different modules. Figure 14 shows the resulting Petri net after applying the TGTS to the DL example.

Each analysis provided to the VisMODLE modelling environment implies evaluating a CTL logical expression on the net's coverability graph (de Lara et al., 2003). With this purpose, we use a function that calculates the coverability graph of the net, as well as a model checker to evaluate the expression, both implemented in AToM^3. In order to use these analysis techniques, users of the VisMODLE environment do not have to know Petri nets at all. All the analysis process is hidden. CTL expressions are defined when the modelling environment is generated, and hidden to the final user. The results of analysing the Petri net are back-annotated and shown to the user in the VisMODLE notation, which is the notation that he knows. This is possible since we maintain in the correspondence graph the relations between the elements in the DL and the ones in the Petri net. AToM^3 provides declarative back-annotation mechanisms that allow the specification of the elements to highlight in the source model as a result of the analysis. For the example, the net showed neither global nor local deadlocks, all the states were reachable, and the tool signalled the possibility of unbounded number of *borrow* and *borrow_request* messages.

6 RELATED WORK

Formal models for DLs are rarely found, likely due to the complexity of the field. The approach of (Wang, 1999) defines DLs as a combination of a special-purpose database and a hypermedia-based UI, and formalizes them by using the Z language. In (Castelli et al., 2002) a multidimensional query language for DLs is used that rely on the notions of views and versions, metadata formats and specifications, and a first-order logic based language. (Gonçalves et al., 2004) presents a formal foundation theory on DLs, called 5S, based on streams, data structures, spaces, scenarios and societies. We were inspired on it in order to design VisMODLE. However, these formal ap-

proaches do not formally specify how to derive the DL implementation from the DL model.

Other declarative approaches, such as the Digital Library Definition Language (Maly et al., 2000), the METIS framework (Anderson et al., 2003) and the FEDORAs structoid approach (Dushay, 2001), are not supported by a strict underlying formal theory.

To the best of our knowledge, none of these approaches provide a customized environment supporting code generation as well as validation (animation) and analysis techniques. By using MDSD techniques, we help DL experts to cope with the complexity of DL designs without dealing with coding. In addition, the use of a DSVL makes it easier for people to learn the domain concepts and interpret the models, and allows modelling interactions among DL systems and users (as proposed in the HCI field). Finally, our approach generalizes some metadata schemas such as DC, in the sense that DC data structures can be modelled by using our structure entities and their relationships.

7 CONCLUSIONS

In this paper, we have presented VisMODLE, a DSVL for building DLs in a model-driven way. The language is made of five diagram types (services, collections, structure, society and behaviour) to describe the aspects of a DL. We have generated a modelling environment for it with the AToM^3 tool. The environment integrates a code generator in order to produce the DL. Although it is in its alpha version, we have already used it to build prototypes. In addition, we have built a visual simulator using graph transformation to graphically validate the DL behaviour at the model level. In order to perform further analysis, a transformation into Petri nets has been designed, which allows checking model properties, such as reachability or deadlocks. This analysis is made by internally performing model-checking of the coverability graph by using predefined temporal logic formulae.

In the future, we intend to support XDoclet for the specification of the VisMODLE services. It allows automatic code generation, compliant with a standard, which simplifies coding for various technologies, such as Java or Web Services. It would be also interesting to generate code for some orchestration language to reflect the overall behaviour of the societal model.

ACKNOWLEDGEMENTS

Work supported by projects MODUWEB (TIN2006-09678) and MOSAIC (TIC2005-08225-C07-06) of the Spanish Ministry of Science and Education.

REFERENCES

Anderson, K. M., Andersen, A., Wadhwani, N., and Bartolo, L. M. (2003). Metis: Lightweight, flexible, and web-based workflow services for digital libraries. In *JCDL*, pages 98–109.

Castelli, D., Meghini, C., and Pagano, P. (2002). Foundations of a multidimensional query language for digital libraries. In *ECDL*, pages 251–265.

de Lara, J., Guerra, E., and Vangheluwe, H. (2003). Metamodelling, graph transformation and model checking for the analysis of hybrid systems. In *AGTIVE'03*, volume 3062 of *LNCS*, pages 292–298. Springer.

Dushay, N. (2001). Using structural metadata to localize experience of digital content. *CoRR*, cs/0112017.

Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag.

Gonçalves, M. A., Fox, E. A., Watson, L. T., and Kipp, N. A. (2004). Streams, structures, spaces, scenarios, societies (5s): A formal model for digital libraries. *ACM Trans. Inf. Syst.*, 22(2):270–312.

Guerra, E. and de Lara, J. (2006a). Attributed typed triple graph transformation with inheritance in the double pushout approach. Technical Report UC3M-TR-CS-06-01, Universidad Carlos III (Madrid).

Guerra, E. and de Lara, J. (2006b). Model view management with triple graph transformation systems. In *ICGT'06*, volume 4178 of *LNCS*, pages 351–366.

Kochtanek, T. R. and Hein, K. K. (1999). Delphi study of digital libraries. *Inf. Proc. Manag.*, 35(3):245–254.

Licklider, J. C. R. (1965). *Libraries of the Future*. MIT Press, Cambridge, Mass.

Malizia, A., Guerra, E., and de Lara, J. (2006). Model-driven development of digital libraries: Generating the user interface. In *Proc. MDDAUI'06*.

Maly, K., Zubair, M., Anan, H., Tan, D., and Zhang, Y. (2000). Scalable digital libraries based on ncstrl/dienst. In *ECDL*, pages 168–179.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580.

Reddy, R. and Wladawsky-Berger, I. (2001). Digital libraries: Universal access to human knowledge—a report to the president. In *PITAC, Panel on DLs*.

Völter, M. and Stahl, T. (2006). *Model-Driven Software Development*. Willey.

Wang, B. (1999). A hybrid system approach for supporting digital libraries. *JDL*, 2(2-3):91–110.