Chapter 9

TRANSFORMATIONAL DEVELOPMENT OF USER INTERFACES WITH GRAPH TRANSFORMATIONS

Quentin Limbourg and Jean Vanderdonckt
IAG – School of Management, Université catholique de Louvain,
Place des Doyens 1 – B-1348 Louvain-la-Neuve (Belgium)
{limbourg,vanderdonckt}@isys.ucl.ac.be
URL : http://www.isys.ucl.ac.be/bchi/members/{qli,jva}
Tel : +32 10/47 {83 84, 85 25} – Fax : +32 10/47 83 24

Abstract In software engineering transformational development aims at developing software systems by transforming a coarse-grained specification to final code (or to a detailed specification) through a sequence of small transformation steps. Transformational development is known to bring benefits such as: correctness-preserving of the development cycle, explicit mappings between development steps, reusability and reversibility of transformations. No piece of literature provides a systematic formal system applying transformational development to user interface engineering. To fill this gap, a methodology, called TOMATO, is described in three facets: 1) A development cycle is defined to outline possible transformations. 2) A language for supporting the methodology is presented relying on graph transformations, a mathematical system for expressing specifications and transformation rules. 3) A tool implementation, using a visual syntax, is illustrated.

Keywords: Forward engineering, Graph grammar, Graph theory, Mapping problem, Program transformation, Reverse engineering, Transformational approach.

1. INTRODUCTION

A state of the art [18] in the field of engineering methods of user interface shows that no method provides an integrated view of the abstractions needed to build a user interface along with an explicit mechanism to manipulate these abstractions throughout a development cycle. More specifically, there is no general logical mechanism to incorporate and manipulate design knowledge in user interface creation tools [2,13] nor any system for relating abstractions needed for this purpose. This problem has been referred to as the *mapping problem* by Puerta and Eisenstein in [16].

This paper addresses the lack of a disciplined and explicit mechanism for supporting user interface development in a transformational approach from early requirements to the final code. We draw the bases of such a mechanism along with the explicit definition of an algorithmic method able to perform automatically (or semi-automatically) the transformation of specification models.

A *UI specification model* consists in a series of representations (called *component models*) pertaining to various facets of the UI such as: user's task, domain objects, UI presentation and dialog, user's characteristics, computing platform, physical environment of interaction, etc [14,18]. A consistent effort has been done in the literature to integrate these specification models in an explicitly articulated and coherent manner. TOMATO methodology (standing for "formal meThOdology for MApping user interface specificaTiOn models) is composed of a development cycle and a language (Tomato-L). It is aimed at supporting transformational development of UIs. Within Tomato any UI artefact is internally represented by a set of models that are analyzable, editable, and processable by software means [14]. Each model is stored in a model repository in a UI specification language based on graph theory. This UI internal representation is then subject to production rules that progressively transform abstract concepts into concrete concepts so as to finally create a full description of a final UI [20]. Once this description is obtained, a rendering tool can be used to produce the running code. Such renderers have already been developed and discussed (www.uiml.org).This paper is focused on the transformation development that leads to the final description.

In the context of model-based development environments [18], at least four works can be cited: Mobi-D [13,14,15,16], Teallach [6], TIDE (www.uiml.org) and TERESA [12]. All these works represent significant attempts to incorporate design knowledge for a user interface design tool. The above tools are advanced in the sense that they support the explicit mappings between the different models, the different views, and steps of the method. In Tomato methodology, these mappings are not hand coded and built-in in the software. Rather, they are graphically expressed in the environment, which allows to exploit these mappings in a flexible way. With respect to the application of graph transformations to user interface development, two contributions can be mentioned: Freund *et al.* [5] and Sucrow [19]. Both approaches make an interesting use of graph transformations but have a too narrow conceptual coverage to address a fully defined UI development cycle.

The purpose of this paper is not to prove that a complete and consistent set of rules can be achieved to store a comprehensive part of design knowledge. Rather, it is intended to show how we can apply transformations from the abstract to the concrete domain in a seamless manner. The remainder of this paper is structured as follows: first, the transformation development life cycle supported by the Tomato methodology will be described. Then, the underlying language and its supporting tools are discussed. An example is presented to introduce the method. Finally, a related work section shows that this type of work remains unprecedented. The conclusion summarizes the main benefits of the approach, while contrasting with potential shortcomings.

2. TRANSFORMATIONAL DEVELOPMENT WITH TOMATO

2.1 Context and Aim

An example is herby exposed in order to better introduce TOMATO methodology. A simple scenario is proposed: a doctor at the hospital has to record information on her patient medical history. For this purpose, she has to input identity information, medical history i.e., general pathologies, heart pathologies, and other problems.

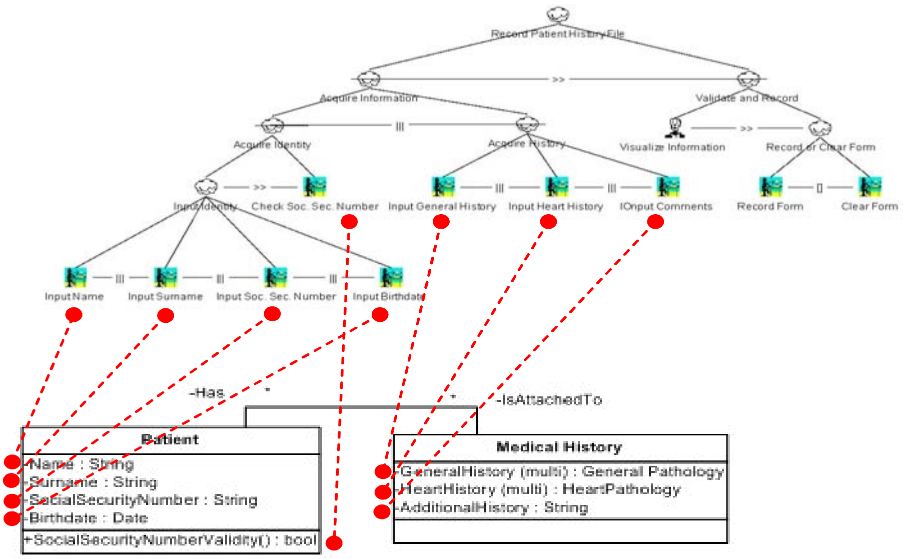


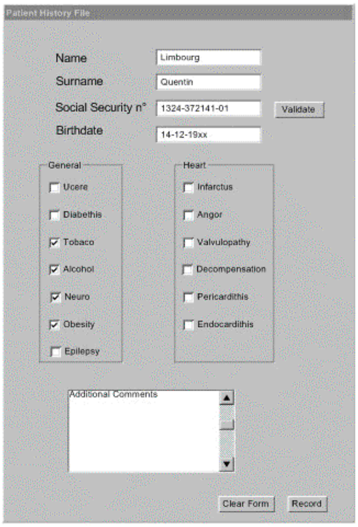
Figure 1. A Conceptual Model (Domain + Task).

Using TOMATO methodology, a developer may initiate the development by a conceptual schema expressing either domain concepts or a task specification or both. Fig. 1 shows a domain and task specification along with the mappings between both models. In this example, input tasks are mapped onto the attribute or the attribute set they are concerned with. Tasks involving a system function are mapped onto domain operations. Having this specification, the designer can pick up a derivation heuristic in a database and generate a detailed specification of the desired UI. The heuristics exploited in this example are listed in Fig. 2. These heuristics are expressed in natural language for the good comprehension of the example. The resulting specification is illustrated by Fig. 3.

Now the developer is told that her system is not suited for patients admitted through the emergency service. In this case a handheld platform has to be used. In consequence, the developer selects an appropriate derivation to transform the specification of the UI of Fig. 3 into a specification suited for a small display device. The resulting specification is illustrated by Fig. 4.

R0: generate a "main" window;
R1: for each multi-valued class attribute, generate a group box whose name is the name of the attribute;
R2: for each multi-valued class attribute whose domain is enumerated and is associated with a group box, generate a checkbox whose label is the label of the enumerated value;
R3: for each class attribute of type string and not multi-valued, generate a label and an input field whose, respectively, caption and name is the name of the attribute. The label and the input field being topologically bound together;
R4: for each attribute of type long string, generate a multiple line edit box;
R5 for each operation class, generate a button whose label is the name of the operation;
R6 (A and B) : each object belonging to a same window are placed following an order depending on the task they allow to accomplish.

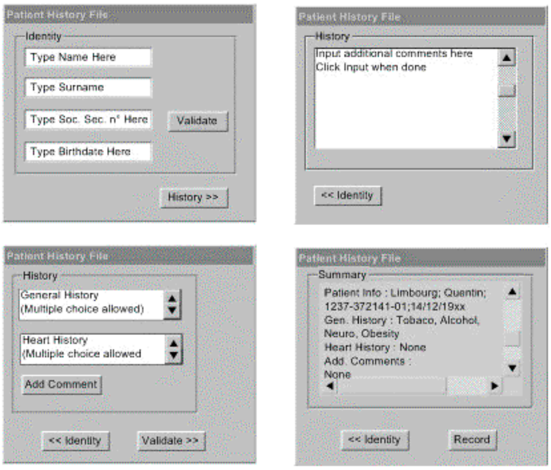
Figure 2. Heuristic sample of the working example.



Name: Limbourg
Surname: Quentin
Social Security n°: 1324-372141-01
Birthdate: 14-12-19xx

General: [ ] Ucere, [ ] Diabethis, [✓ Tobaco], [✓ Alcohol], [✓ Neuro], [✓ Obesity], [ ] Epilepsy

Heart: [ ] Infarctus, [ ] Angor, [ ] Valvulopathy, [ ] Decompensation, [ ] Pericardithis, [ ] Endocardithis



Patient Info : Limbourg; Quentin; 1237-372141-01;14/12/19xx
Gen. History : Tobaco, Alcohol, Neuro, Obesity
Heart History : None
Add. Comments : None

Figure 3. (Left) & 4 (Right) Recording Patient History File.

The initial specification has been split up into four smaller interaction spaces. Navigation between these interaction spaces has been automatically generated. Some widgets have been replaced by a degraded equivalent (e.g., a group of text boxes has been replaced by a multiple selection list box [22,23]). Because patient information is scattered between several interaction spaces so that the task of checking the information before recording the file can not be done appropriately, a summary interaction space in generated. This example shows a possible application of Tomato methodology. Next section exposes this methodology in a systematic way.

2.2 Development Cycle: TOMATO Cycle

Tomato cycle complies with transformational development theories. Transformational development can be viewed as a development process that takes as input a high level specification and produces as output a more concrete specification (i.e., implementation oriented) or an executable program. The transformation process itself takes the form of a sequence of small transformation steps. Each step preserves some desirable properties (e.g., correctness or consistency [9]).

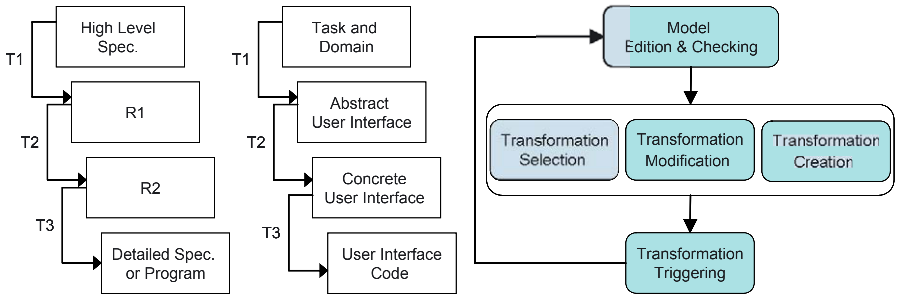


Figure 5. Tomato Cycle.

Fig. 5 illustrates the sequence of abstractions and designer's tasks to transform a high level specification into a refined specification through the following steps [3]: task and domain, abstract user interface, concrete user interface, and user interface code.

With such a transformational development, the role of the developer is very different from traditional approaches. In traditional approaches, the developer receives a specification, tries to fully understand it and implements what he has understood in a specific development environment. With transformational approaches [10], the developer receives a specification, tries to fully understand it, edits it, selects/modifies/creates an appropriate transformation and applies it to the initial specification in order to finally obtain a refined specification. Properties of the resulting model can then be checked against a set of rules expressing coherence or usability properties.

Regarding the role of the developer, three types of transformations can be identified [10]: manual transformations require intervention of the developer at each stage for choosing the appropriate transformation, semi-automatic transformation and fully automatic transformation. TOMATO cycle adopts a manual approach i.e., the developer builds the sequence of transformation himself. This is explained by the fact that transformations are heuristics. There is no single way to transform a task model into a presentation, or a presentation adapted to a large display to a presentation adapted to a small display, etc.

2.3 THE TOMATO-L

TOMATO-L is a language that enables the expression of concepts needed to build a user interface. TOMATO-L structure is defined in Fig. 6.

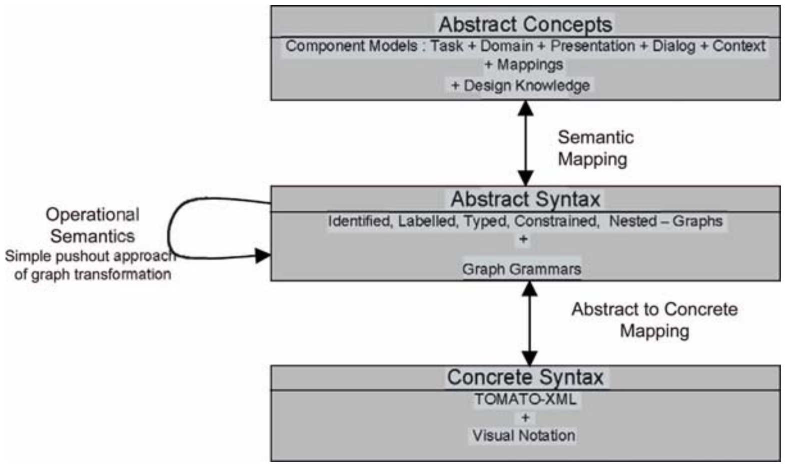

Abstract Concepts
Component Models : Task + Domain + Presentation + Dialog + Context
+ Mappings
+ Design Knowledge
Semantic Mapping
Abstract Syntax
Identified, Labelled, Typed, Constrained, Nested – Graphs
+
Graph Grammars
Operational Semantics
Simple pushout approach of graph transformation
Abstract to Concrete Mapping
Concrete Syntax
TOMATO-XML
+
Visual Notation

Figure 6. TOMATO Language.

2.3.1 Abstract Concepts

The *abstract concepts* we consider to formalise with graph structures consist in abstractions needed to build a UI in a model-driven approach [11]. It is impossible to list here all language elements. Nonetheless, these abstractions can be categorised into three main classes.

1. **Component models** partition concepts needed to construct a UI. Component models allow building several views on a UI. These views help to answer questions like: what tasks does my UI support? What objects does it manipulate? How does it look like? How does it behave? Component models have been listed in [14]. They consist in: (1) a *task model* represents a decomposition of user tasks in interaction with a system in order to reach a specific goal, (2) a *domain model* represents the concepts ma-

nipulated by the user while interacting with the system, (3) a *presentation model* consists in a specification of a hierarchy of graphical elements composing a UI along with their respective topological constraints, (4) a *dialog model* represents the dynamic aspects of a presentation model. Dialog modelling can concern different levels of granularity. We focus here on navigational aspects i.e., window transitions. 5) a *context model*. The context model essentially serves to describe in which conditions a specific UI specification is valid or not. It is beyond the scope of this paper to discuss extensively the context model. Schematically, our context model is composed of [3,12] (a) a user model describing the main characteristics of some user's users ? stereotypes (b) a platform model containing the description of hard- and soft- resources exploited to render a UI (c) an environment model describing environmental factors affecting the way users interact (noise or light level, stress conditions,…).

2. **Mappings** are relationships between component models [15,16]. These relationships are very interesting as they realize the integration of component models into one whole specification instead of having a collection of unrelated abstractions (this partly provides seamlessness to our method). Concretely, expressing mappings allows us to answer questions like: what objects do I need to accomplish this task? (task-domain mapping) What graphical objects support this task or represent this object? (<task, domain>-presentation mapping).
3. **Design Knowledge** is the knowledge that is put into practice while building a UI [13]. In our perspective, applying design knowledge means manipulating component models and mappings. Design knowledge allows answering questions like: what widgets are more appropriate to represent such domain object? How should I lay out objects into a container? Which navigation is preferred by a user stereotype? What kind of transition should I have between two windows? [24]. More detailed examples are provided in Section 2.4.

2.3.2 Abstract Syntax and Operational Semantics

The *abstract syntax* is defined as the hidden structure of a language, its mathematical background [9]. Our abstract syntax takes the form of a directed graph. A graph *g* is defined as a quadruple (V, E, source, target) such that (1) V is a finite set of vertices (2) E is a finite set of edges (3) source: E $\to$ V is an injective function assigning a source to each edge of E (4) target: E $\to$ V is an injective function assigning a target to each edge of E. To enable the expression of a specification model within a graph structure we *enrich* the initial definition of graph with several interesting features. Most important features are: (1) labelling: enables each edge or node to be labelled

(2) typing: enables edges and nodes to be classified into types (3) constraining: enables to attach to nodes and edges constraints of various types (e.g., cardinality constraint) (4) nesting: enables to nest a graph into another graph.

After expressing models, the abstract syntax of TOMATO-L expresses design knowledge via *graph grammars*. Graph grammars are set of rules, called *productions*. Productions aim, in this context, at transforming the graph representing UI artefacts. In order to transform graphs (i.e., UI artefact transformation), a grammar is applied to an initial graph, called *host graph* leading to a *resultant graph.* A resultant graph is said *final* if there is no more applicable production to this graph. It is said *intermediate* in the opposite case. The application of a production is called a *graph transformation step* [7], for short *a derivation.*

The *operational semantics* of a language describes the way an automaton (called interpreting automaton) transforms an input into an output [9]. The behaviour of the automaton for graph transformation depends on the chosen transformation technique. The technique used in this work is known as *Single PushOut approach* (SPO). It is illustrated in Fig. 7.

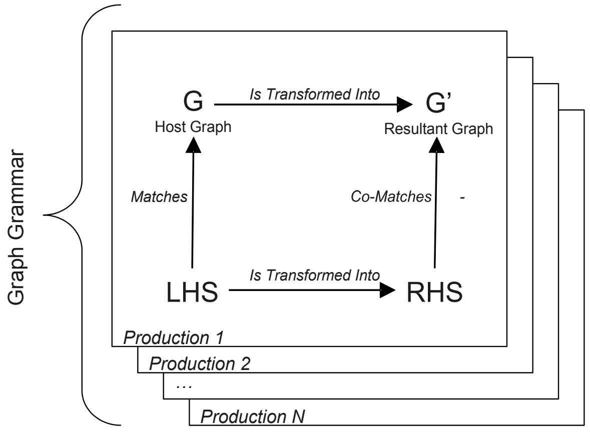


Figure 7. Production and Grammar in the Single Pushout Approach of Graph Transformation.

When a Left Hand Side (LHS) matches into a host graph G, it is replaced by a Right Hand Side (RHS). G is resultantly transformed into G'. All elements of G not covered by the match are considered as unchanged.

In order to achieve a better level of expression of productions, the mechanism of LHS match is complemented with 1) Positive Application Conditions (PAC), expressed as textual Boolean expressions on variables of the LHS and 2) Negative Application Conditions (NAC). A NAC is an additional condition to a production that contains a graph with which the host graph *must not* match with. In addition, several technical problems may arise while applying a grammar to a host graph e.g., conflicts between rules, occurrence of dangling edges or dependencies between rules leading to an indeterminable resultant graph. We deal with this problem by adopting a conservative and cautious approach by (1) identifying production conflicts a priori when possible, (2) erasing all dangling edges in resultant graphs, (3) constraining the application of productions to a specific order (programmed graph rewriting).

2.3.3 Concrete Syntax

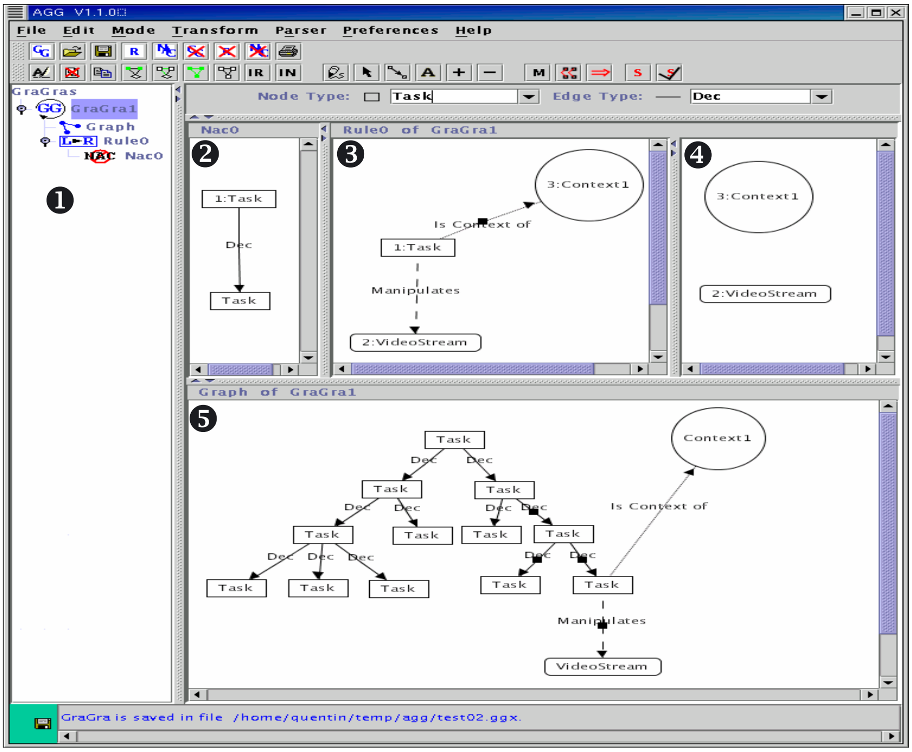
The *concrete syntax* of a language is its external appearance. Tomato-L has two concrete syntaxes: (1) a graphical syntax which consists in the notation used in this paper. Its elements are just boxes, arrows and labels. The advantage with this notation is that it is visual. The disadvantage is that it can not, as is, be manipulated by an automaton (2) a textual syntax (called TOMATO-XML) of XML files is also provided.

Existing UI Description Languages (UIDLs) like XIML (http://www.ximl.org), UIML (http://www.uiml.org), and XHTML are limited to the expression of a concrete syntax. TOMATO concrete syntax is governed by an XML schema. It is logically derived from its abstract syntax as its structure is twofold: a set of nodes describing the elements populating the model at hand, a set of relationship describing the relationships between these different elements. An excerpt of an instance file in shown in Fig. 8.

2.4 TOOL IMPLEMENTATION

The principles exposed above could be put into practice in various programming environment enabling an easy expression and manipulation of graph structures (e.g., Prolog). An environment called AGG (Attributed Graph Grammars tool) is used for this experiment. AGG can be considered as a genuine programming environment based on graph transformations [7]. It provides:

1) A programming language enabling the specification of graph grammars.
2) A customisable interpreter enabling graph transformations. AGG was chosen because it allows the graphical expression of directed, typed and attributed graphs (for expressing specifications and rules). It has a powerful library containing notably algorithms for graph transformation [7], critical pair analysis, consistency checking, positive and negative application condition enforcement. AGG user interface is described in Fig. 8. Frame 1 is the grammar explorer. In Fig. 8, frames 2, 3 and 4 enable to specify sub-graphs composing a production: a negative application (frame 2), a left hand side (frame 3) and a right hand side (frame 4). The host graph on which a production will be applied is represented in Frame 5.



GraGra is saved in file /home/quentin/temp/agg/test02.ggx.

Figure 8. AGG User Interface.

Fig. 9 illustrates a rule used to perform the example exposed in section 2.1 (R1 in Fig. 2). It asserts that for every possible multi-valued attribute, a group box is generated. The group box's name is the name of its corresponding attribute. A negative application condition (left) avoids an infinite iteration of this rule. In order to ensure a possible manipulation of the output produced by AGG, an export function towards TOMATO textual syntax has been realised. An import function is currently under development.

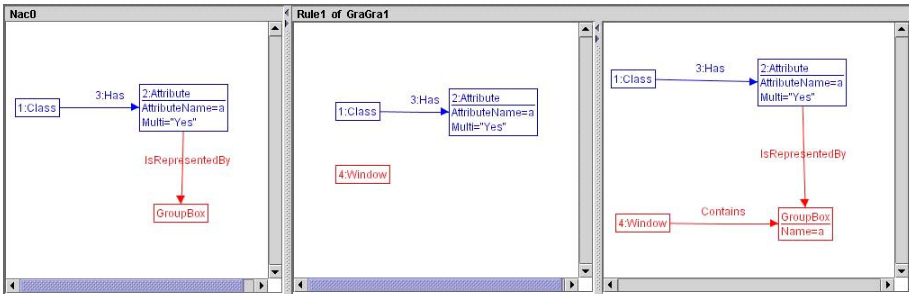


Figure 9. Composition of a rule: a NAC (left), a LHS (center), a RHS (right).

Experiments showed that AGG is a proper environment for defining and applying rules. Unfortunately, it shows poor in terms of usability for specifying large UI models. Indeed, it may appear somewhat abstract to the designer to describe a UI appearance with a set of nodes and relationships. An external tool for visually ("WYSIWYG" style) manipulating abstract and concrete UI models is under development in our lab. At this time, the tool allows specifying graphically a UI in terms of concrete widgets. A property sheet allows the detailed specification of the widget properties. The tool exports the specification of the created UI in a syntax that is compliant with Tomato concrete syntax. An import function, and consequently a rendering function, of Tomato concrete syntax are currently being developed. The main features of our tool experimentation can be summed up as follows.

1) A 'design editor' allows the creation and the consolidation of models exploited in the development process. A specific environment enables the design of UI appearance by direct manipulation of widgets.
2) A 'design derivator' enables the transformation of a model into another model.
3) A 'rule editor' enables the definition of new transformation rules.
4) A 'rule validator' enables the designer to identify conflicts within a set of rules. The critical pair analysis technique is used for this purpose.
5) A 'design analyser' enables the verification of desirable properties of the manipulated artefacts such as basic consistency rules, type checking or even usability properties (i.e., IFIP properties like reachability, browsability).

3. CONCLUSION

In this paper, a formal development methodology (Called TOMATO-M) enabling the construction of user interfaces through a set of transformations has been presented.

This methodology relies on: 1) the representation of manipulated abstractions with directed, labelled, attributed, and typed graphs. 2) the progressive transformation of higher level specification models to lower level specification code via the application of graph transformations 3) the expression of design knowledge with an explicit and developers accessible language.

The traditional role of the developer is challenged with TOMATO as it consists in 1) the expression of specification models under the form of graphs 2) the access, definition, extension, restriction, testing, verification and, ultimately, the application of appropriate transformations corresponding to design heuristics. The advantages of such a method can be summed up as follows:

- A *logical expression of design knowledge*: rather than having concepts of components models used in the process being hard coded and built-in within the design tools. All design rules, heuristics, algorithms can be expressed through productions that can be logically and mathematically defined.
- A *flexible production process*: productions can be gathered in graph grammars to be executed on graphs representing the starting models (e.g., task, domain, and user) to obtain the final models (e.g., presentation and dialog). This process is flexible in the sense that it can be controlled (forward, backward, and both) by the tool engine, thus providing developers with a great degree of freedom.
- A *reusable and combinable way of using design knowledge*: any form of design knowledge, once expressed in the Tomato language, can be reused at any time, can be refined when experience is growing, can be stopped when needed, and can be combined with other rules to obtain a more or less sophisticated production process. Using the same graph grammar also reinforces the consistency of produced results.
- A *visual and mathematical expression*: while the developer can graphically express productions in AGG tool, each production is stored as a graph transformation rule, a mathematically sound concept.
- A *coverage for many particular methods*: each method or tool typically promotes its particular process. As productions can be arranged in bidirectional ways and can start from any model, we believe that multiple entry points and top-down or bottom-up approaches can be supported. For example, linking and deriving rules from Teallach [6] can be expressed in TOMATO. Similarly, multiple UIs for multiple contexts of use could be obtained through different graph grammars.

On the other hand, preliminary results obtained with the TOMATO method revealed that some abstraction effort is required by the person who is responsible to incorporate the design knowledge. But once the designed knowledge is introduced into the tool it can be experimented with a limited experience of the language [21].

ACKNOWLEDGEMENTS

This work has been supported by the Cameleon research project (http://giove.cnuce.cnr.it/ cameleon.html) under the umbrella of the European Fifth Framework Programme (FP5-2000-IST2), which we gratefully acknowledge. This work gave birth to the USIXML language (User Interface eXtensible Markup Language – www.usixml.org) that is used across the members of the Cameleon project. The authors would like to thank Cameleon partners who contributed to V1.2 of USIXML: Lionel Balme, Gaëlle Calvary, Cristina Chesta, Alexandre Demeure, Joëlle Coutaz, Jean-Thierry Lechein, Fabio Paternò, Stéphane Raymond, Carmen Santoro, and Youri Vanden Berghe. This paper is related to USIXML V1.4, an extension of USIXML V1.2 with dialog model, more inter-model mappings, a context model made up of user, platform, and environment, and the concrete user interface level.

REFERENCES

[1] Baresi, L. and Heckel R., *Tutorial Notes on Foundations and Applications of Graph Transformation, an Introduction From a Software Engineering Perspective*, in Proceedings of 1st Int. Conference on Graph Transformation, ICGT'02 (Barcelona, 7-12 September 2002), Springer-Verlag, Berlin, 2002, pp. 402-429, accessible at http://link.springer.de/link/service/series/0558/bibs/2505/25050402.htm

[2] Brown J., *Exploring Human-Computer Interaction and Software Engineering Methodologies for the Creation of Interactive Software*, SIGCHI Bulletin, Vol. 29, No. 1, January 1997, pp. 32-35.

[3] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J., *A Unifying Reference Framework for Multi-Target User Interfaces*, Interacting with Computers, Vol. 15, No. 3, June 2003, pp. 289-308.

[4] Engels, G. and Schürr, A., *Encapsulated Hierarchical Graphs, Graph Types and Meta Types,* in Proceedings of Joint Compugraph/Semagraph Workshop on Graph Rewriting and Computation, Electronic Notes in Theoretical Computer Science SEGRAGRA'95 (Volterra, 28 August-1 September 1995), Vol. 2, July 1995.

[5] Freund, R., Haberstroh, B., and Stary, C., *Applying Graph Grammars for Task-Oriented User Interface Development*, in Proceedings of International Conference on Computing and Information ICCI'1992 (Toronto, Ontario, 28-30 May 1992), IEEE Computer Society Press, Los Alamitos, 1992, pp.389-392.

[6] Griffiths, T., Barclay, P., Paton, N.W., McKirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C., and Pinheiro da Silva, P., *Teallach: A Model-Based User Interface Development Environment for Object Databases*, Interacting with Computers, Vol. 14, No. 1, 1 December 2001, pp. 31-68.

[7] Ehrig, H., Engels, G., Kreowski, H-J., and Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformation*, Applications, Languages and Tools, Vol. 2, World Scientific, Singapore, 1999.

[8] Mens, T., *Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution,* in Proc. of International Conf. Applications of Graph Transformations with Industrial Relevance AGTIVE'1999 (Kerkrade, 1-3 September 1999), Lecture Notes in Computer Science, Vol. 1779, Springer-Verlag, Berlin, 2000, pp. 127-143.

[9] Meyer, B., *Introduction to the Theory of Programming Languages*, Prentice Hall, New York, 1990.

[10] Partsch, H., and Steinbruggen, R., *Program Transformation Systems*, ACM Computing Surveys, Vol. 15, No. 3, September 1983, pp. 199-236.

[11] Paternò, F., *Model-Based Design and Evaluation of Interactive Applications*, Springer-Verlag, Berlin, 2000.

[12] Paternò, F., and Santoro, C., *One Model, Many Interfaces*, in Proceedings of 4th International Conference on Computer-Aided Design of User Interfaces CADUI’2002 (Valenciennes, 15-17 May 2002), Kluwer Academics Publishers, Dordrecht, 2002, pp. 143-154

[13] Puerta, A.R., and Maulsby, D., *Management of Interface Design Knowledge with MOBI-D*, in Proc. of 2nd ACM International Conference on Intelligent User Interfaces IUI’97 (Orlando, 6-9 January 1997), ACM Press, New York, 1997, pp. 249-252.

[14] Puerta, A.R., *A Model-Based Interface Development Environment*, IEEE Software, Vol. 14, No. 4, July-August 1997, pp. 41-47

[15] Puerta, A.R. and Eisenstein, J., *Interactively Mapping Task Models to Interfaces in MOBI-D*, in Proceedings of 5th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems DSV-IS’98 (Abingdon, 3-5 June 1998), Springer-Verlag, Vienna, 1998, pp. 261-273.

[16] Puerta, A. and Eisenstein, J., *Towards a General Computational Framework for Model-Based Interface Development Systems Model-Based Interfaces*, in Proc. of ACM International Conference on Intelligent User Interfaces IUI’99 (Los Angeles, 5-8 January 1999), ACM Press, New York, 1999, pp. 171-178.

[17] Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformation*, Foundations, Vol. 1, World Scientific, Singapore, 1999.

[18] Szekely, P., *Retrospective and Challenges for Model-Based Interface Development*, in Proc. of 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI’96 (Namur, 5-7 June 1996), Presses Universitaires de Namur, Namur, 1996, pp.1-27.

[19] Sucrow, B., *On Integrating Software-Ergonomic Aspects in the Specification Process of Graphical User Interfaces*, Transactions of the SDPS Journal of Integrated Design & Process Science, Vol. 2, No. 2, June 1998, pp. 32-42.

[20] Vanderdonckt, J. and Bodart, F., *Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection*, in Proc. of the ACM Conf. on Human Factors in Computing Systems INTERCHI'93 (Amsterdam, 24-29 avril 1993), ACM Press, New York, 1993, pp. 424-429.

[21] Vanderdonckt, J., *Assisting Designers in Developing Interactive Business Oriented Applications*, in H.-J. Bullinger & J. Ziegler (eds.), Proceedings of 8th International Conference on Human-Computer Interaction of HCI International’99 (Munich, 22-26 August 1999), Ergonomics and User Interfaces, Vol. 1, Lawrence Erlbaum Associated Pub., Mahwah, 1999, pp. 1043-1047.

[22] Vanderdonckt, J., *Advice-Giving Systems for Selecting Interaction Objects*, in N.W. Paton & T. Griffiths (eds.), Proceedings of 1st Int. Workshop on User Interfaces to Data Intensive Systems UIDIS’99 (Edimburgh, 5-6 September 1999), IEEE Computer Society Press, Los Alamitos, 1999, pp. 152-157.

[23] Vanderdonckt, J. and Berquin, P., *Towards a Very Large Model-based Approach for User Interface Development*, in N.W. Paton & T. Griffiths (eds.), Proc. of 1st International Workshop on User Interfaces to Data Intensive Systems UIDIS’99 (Edimburgh, 5-6 September 1999), , IEEE Computer Society Press, Los Alamitos, 1999, pp. 76-85.

[24] Vanderdonckt, J., Limbourg, Q., and Florins, M., *Deriving the Navigational Structure of a User Interface*, in M. Rauterberg, M. Menozzi, J. Wesson (eds.), Proc. of 9th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT’2003 (Zurich, 1-5 September 2003), IOS Press, Amsterdam, 2003, pp. 455-462.