Rapid Software Prototyping Using Visual Language Techniques

Kang Zhang Guang-Lei Song Jun Kong

Department of Computer Science, University of Texas at Dallas
Richardson, Texas 75083-0688, USA
{kzhang, gxs017800, jxk019200}@utdallas.edu

Abstract

Rapid prototyping of domain-specific software requires a systematic software development methodology and user-friendly tools. Being both executable and easy to use, visual languages and their automatic generation mechanisms are highly suitable for software prototyping. This paper presents a software prototyping methodology based on the visual language generation technology, that allows visual prototyping languages to be specified and generated using an expressive graph grammar formalism. Executable prototypes and their verification and code generation are made possible by syntax-directed computations. The paper demonstrates this methodology through a prototyping example built on our current implementation.

1 Introduction

Prototyping is an effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. There has been no commonly accepted executable specification language or high-level programming language for computer-aided prototyping of quality domain-specific software. Existing prototyping tools are either too specialized to be useful in a wide spectrum of application domains, or too difficult to use by non-computing professionals. Also, few specification or prototyping languages or tools support non-disposable prototyping. In most cases, an accepted prototype has to be re-implemented in a conventional programming language.

Rapid prototyping of domain-specific software requires a systematic software development methodology and user-friendly tools. Being both executable and easy to use, visual languages and their automatic generation mechanisms are highly suitable for software prototyping. As visual tools are becoming increasingly popular for non-computing professionals and end-users, visual languages would potentially challenge traditional specification languages for rapid software prototyping due to their intuitiveness and ease-of-use.

Visual programming refers to a process in which the user specifies a program in a two (or higher) dimensional fashion [14]. Visual languages include languages where programming is done by manipulation of visual objects, languages designed to facilitate algorithm and program animation, and languages for use in computer-oriented applications such as software engineering and database design and access [3]. Different from CASE toolsets, visual languages are executable specification languages that are defined by graph grammars and syntax-directed rewriting rules, apart from being at high level [19]. This means that prototyped domain software can also be rigorously verified according to the provided specification. We will call the visual languages designed for rapid software prototyping as *visual executable prototyping languages* (VEPLs). This paper presents an automatic language-generation mechanism as a meta-tool for the fast generation and reuse of domain-specific language modules, using a graph grammar-based approach to the specification and verification of domain software prototypes.

Automatic generation of VEPLs involves the generation of a visual programming environment with a graphical user interface tailored for the VEPL users. A VEPL generation system enforces the definition of graph grammars and syntax-directed rewriting rules according to domain requirements, before a VEPL is generated. The generated VEPL, therefore, can be used to rapidly prototype high quality domain software. The execution trace and measurements of a domain prototype written in the VEPL may be specified using syntax-directed rewriting rules so that the prototype can be quantitatively and functionally analyzed. Iterative and incremental prototyping is naturally supported by the two-step process. The domain software could be easily re-prototyped by visual programming in the VEPL, or, if structural (syntactical) or behavioral (semantic) changes are required, the VEPL could be re-generated by specifying the grammars and rewriting rules with new and evolving features.

The thrust of this methodology is the meta-tool capability, i.e. a software engineer has a VEPL generator at his/her disposal for generating any required VEPL or re-generating a modified VEPL whenever needed. This methodology is explained in the next section. Section 3 gives a detailed prototyping example using this methodology. Related work

is compared in Section 4, followed by the conclusions in Section 5.

2 Automatic Generation of Visual Executable Prototyping Languages

2.1 Design Criteria

Tools and formalisms have been created for automatically generating visual languages. Most of them are specialized in certain aspects of visual language generation, e.g., for user interface generation [10], and for parsing [4][13]. Little work has been done in automatically generating desired visual languages from rigorous specifications. In a visual programming environment, users must be able to interactively construct and manipulate reusable components in the visual language. The graphical requirements of a visual language include defining the reusable visual components of the language and the control and dataflow relationships that must be maintained when these components are connected together. The editing operations themselves are event-driven, and appropriate interpretations of mouse and keyboard events must be provided.

When considering the design of a visual software prototyping framework that supports the whole process of automatic generation of VEPLs, we need to consider the following design criteria:

- It is necessary for the VEPL generation framework to be an executable programming environment so that VEPLs can be automatically generated and the software engineer is free from the general tasks in the construction of a new VEPL. He/she needs only to provide domain-related specifications for the desired VEPL using an automated toolset.
- The toolset should support syntactic and semantic specifications for effectively designing VEPLs and efficiently parsing and executing prototypes constructed as visual programs in a VEPL.
- It is desirable for the framework to be customizable to a wide spectrum of VEPLs, and to support an iterative and incremental process of prototyping.

2.2. The Meta-Tool Concept

Software engineers of complex software systems typically use diagramming methods, such as UML, as conceptual devices to conceptualize their architectural design. Compared with text, graphs can represent semantic and structural information more intuitively. As shown in Figure 1, we provide a structural specification and verification mechanism in the form of graphical production rules for software engineers. Once a software engineer has provided the specification through production rules, a VEPL with its visual programming environment will be automatically generated in the same fashion as generating textual languages using Lex/Yacc. A programmer (or user), without any knowledge of the prototyping language syntax or the detailed specification, will be able to use the programming environment to construct prototypes visually on the graphical editor of the tool and run the prototypes to verify their requirements. Once the content of each software component (i.e. node of the graph) is provided, the system is able to generate the final prototypical and executable programs.

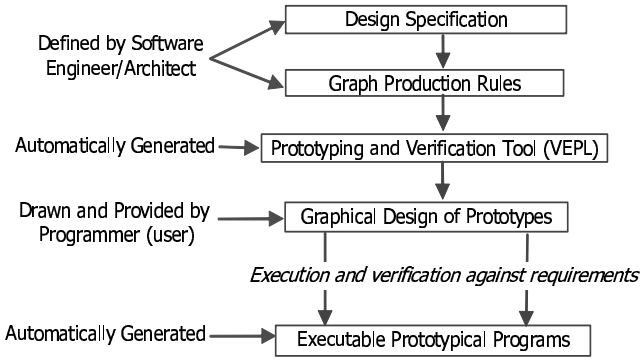


Figure 1: The meta-tool concept

Therefore, more disciplined software development is encouraged by the 2-level process, i.e. the meta-tool used by the software engineer and the visual prototypical tool by the programmer. The development can be iterative and incremental. Thus, the grammar-based specification approach advocates a sound yet flexible design practice.

The VEPL generation framework concept is shown in Figure 2, which illustrates four groups of specifications, to be provided by a software engineer through three graphical tools. *Visual component specifications* define a set of visual objects (using the Visual Object Tool) representing software components to be used in the domain VEPL. The VOT can access software components from the component repository by either reusing existing components or constructing new components using the generated VEPLs. They can also be reused in generating VEPLs. *Control and simulation specifications* provide a formal method for describing the interactive behavior of different parts of the VEPL and how the VEPL will be executed (including how its execution to be visualized). They also define how the generated VEPL environment interacts with the programmer. *Formal method specifications* provide grammar rules that determine the syntax and semantics of a VEPL and can reason or transform a diagram for debugging, animation, or other purposes. Control and parsing specifications are provided using the Control and Parsing Tool. With the visual objects, control and parsing rules specified, the framework

is automatically customized into the visual programming environment for the desired VEPL. Most of the above functionality except the final code generation has been implemented.

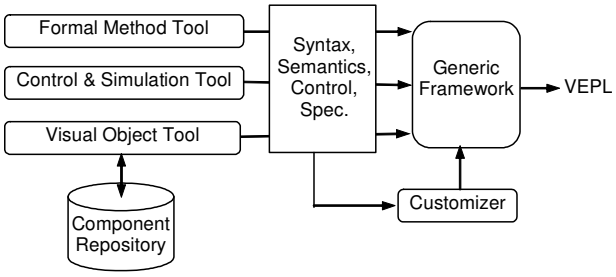


Figure 2: VEPL generation framework and tools

2.3. Specification and Verification Through Graph Grammars

This section introduces the formal method approach in a graph grammar formalism. As the specification language, a VEPL must be expressive and generic enough in specifying structural and semantic properties. The multi-dimensionality of visual languages makes it difficult to build formal grammars and compilers for them. Attempts at developing visual grammars using textual grammars as models have not been very successful; many existing visual language formalisms cannot be specified and parsed effectively and efficiently with existing grammars.

Graph grammars provide a theoretical foundation for graphical languages [16]. A graph grammar consists of a set of rules, which dictates the way of constructing a complete graph from a variety of nodes. A graph grammar specifies all possible legal inter-connections between individual components, i.e. any link in a valid graph can be eventually derived from a sequence of applications of grammar rules. Conversely, an un-expected link signals a violation on the graph grammar. A graph grammar can be used to "glue" various components into a complete system.

As a context-sensitive graph grammar formalism, the Reserved Graph Grammar (RGG) is expressive and also efficient in parsing [18]. It is thus used as the underlying formalism in the VEPL generation framework. In a RGG, a graph rewriting rule as shown in Figure 3, also called a *production*, where DTGS and VDB abbreviate "Dallas TollGate System" and "Vehicle Database" respectively, as elaborated in Section 3. A production has two graphs called *left graph* and *right graph* (. It can be applied to another graph (called *host graph*) in the form of an *L-application* or *R-application*. A *redex* is a sub-graph in the host graph which is isomorphic to the right graph in an R-application or to the left graph in an L-application. A production's L-application to a host graph is to find in the host graph a *redex* and replace it with the right graph of the production. A visual language is defined by all possible graphs that have only terminal labels and can be derived using L-applications from an initial graph (i.e. λ). An R-application is a reverse replacement (i.e. from the right graph to the left graph) that is used to parse a graph.

Nodes in RGGs are organized into a two-level hierarchy. A large rectangle (Figure 3) is the first level with embedded small rectangles as the second level called *vertices*. In addition to the two-level hierarchy structure of a node, the *marking* mechanism, which avoids ambiguity and thus accelerates parsing, makes the RGG expressive yet much faster in execution than other graph grammar formalisms [18]. A marked vertex is identified by a unique integer attached to the vertex label. If a vertex in a right graph is marked, it is allowed to be connected, in a host graph, to any node outside of the redex that matches the right graph. The marked vertex preserves its associated edges connected to the outside of the redex during parsing. For example, the marked vertex *T* in node VDB may be connected to multiple TollGate nodes (as in Figure 7), which can be parsed successfully. Parsing a TollGate connected to multiple VDBs will fail, however, since vertex *V* in TollGate is unmarked.

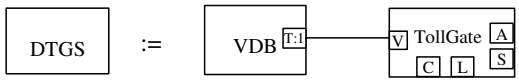


Figure 3: A production rule in the RGG notation

The RGG supports syntax-directed computations by associating data and operators to nodes in productions in terms of *attributes* and *actions*. An attribute expresses a piece of data related to the component represented by a node, and can be retrieved and evaluated in the process of parsing. Different actions can be performed on different attributes of the redex of the production to achieve the desired execution effects. Writing an action code is like writing a standard exception handler in Java by treating each attribute as an object. For example, to retrieve the payment information for a vehicle and update the vehicle's information in the database system, we will attach the following action code to the production in Figure 3:

```
Action(AAMGraph g) {
        If (Car.Authorized == true) {
           TollGate.ConnectDataBase();
           DataBase.UpdateInfo(Car);
         }
```

Therefore, a graph representing a software prototype in a VEPL can be executable, and thus be able to simulate and verify the system requirements. Another straightforward and obvious use of action codes is to provide the actual program segment and test case generation code corresponding to each production. In other words, the generated proto-

type needs not be disposed after being determined to conform to the specification. The VEPL environment can automatically construct, from the satisfied prototype, the final program code in a high-level language, such as C++.

The RGG is equipped with a deterministic parsing algorithm, called *selection-free parsing algorithm* (*SFPA*) [18]. Informally, the selection-free property ensures that different orders of applications of productions result in the same result. Zhang *et al.* [18] proves that a failed parsing path indicates an invalid graph (i.e. prototype), and thus SFPA is efficient with a polynomial parsing complexity by only trying one parsing path. This theoretical result allows the RGG to be used in practical specification and execution of visual prototyping languages.

Software verification is to check the conformance of developed programs to their specification. A graph grammar specification assumes a graph homomorphism for every graph (i.e. a prototype instance) that could be derived from the specified grammatical rules. Graph grammars have been successfully applied to the verification of inheritance relationships between classes [9], and the detection and elimination of dead code [17]. A graph, i.e. a program, in a visual language can be seen as a result derived by applying graph rewriting rules, i.e. L-applications as described earlier. The L-application steps can be animated during parsing so that the user can easily understand how the verification process proceeds, and thus preserve his/her "mental map" rather than be presented with the final verification result. The user may also visually detect any specification errors during the parsing process.

3 A Prototyping Example

The RGG, serving as the underlying formalism of domain VEPLs, defines a VEPL in terms of node-edge diagrams. A node in the RGG represents a module in a prototype. A vertex in a node represents an interface of the module to other modules or the outside. The link between two vertices represents the control and dataflow between the two modules. System designers specify the syntax and semantics of the modules, interfaces, control and dataflow via the RGG production rules. The meta-tool framework can automatically generate a graph editor together with the parser for the VEPL according to the specified production rules. Prototypes written in the VEPL could be drawn manipulated, verified, and executed by a none-software-specialist with the domain knowledge of the system requirements.

Briefly there are three major steps in generating a domain-specific VEPL within the meta-tool framework.

1. Identify possible modules of the system by extracting and analyzing information from system requirements;
2. Specify the production rules and syntax-directed computations associated with the rules (as action code); and
3. Automatically generate the domain specific VEPL.

The procedure could be iterative and evolutional in practice, e.g. in the second step one may find some additional modules needed, and then go back to Step 1 and add to the system. After Step 3, one may also go back to Step 2 to modify the rules if some relationships between the modules need to be added or removed. To clearly describe the prototyping process using the RGG, we go through an example in the following sub-sections.

3.1 System Requirements

Generally, system requirements analysis is the first step of the system development. A major problem with the traditional waterfall lifecycle approach is the lack of any guarantee that the resulting product will meet the customer's needs. In most cases the blame falls on the requirements phase of the lifecycle [11]. Often users are not able to provide sufficient and accurate requirements until they observe the behavior of the functional modules.

Many system requirements analysis tools have been developed [5], and software prototyping serves as a tool for system requirements [11]. Requirements analysis is a process that extracts system design elements, such as modules, interfaces of a module, from the customer's formal requirement documents. The requirements for our prototyping example are informally given as the following.

In a road traffic pricing system, drivers of authorized vehicles are charged at tollgates automatically. The tolls are placed at special lanes called green lanes. A driver has to install a device (called an EzPay) inside his/her vehicle's windshield in order to pass a green lane.

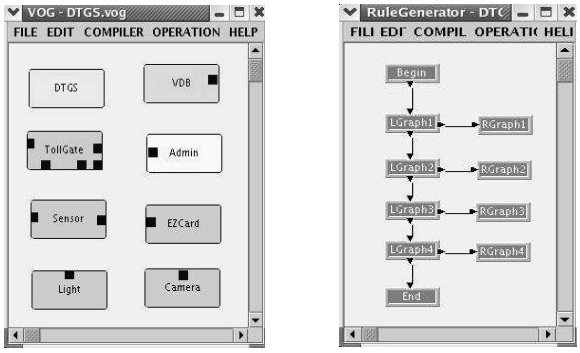
Each tollgate has a sensor that reads EzPay. When an authorized vehicle passes through a green lane, the green light is turned on, and the amount being debited is displayed. If an unauthorized vehicle passes through a green lane, a yellow light is turned on and a camera takes a photo of the vehicle's license plate.

Assume that we will define and generate a VEPL for prototyping the Dallas TollGate System (DTGS) meeting the above requirements.

3.2 Identification of System Modules

The meta-tool framework provides a visual object tool (called VOG) for system modules in the DTGS VEPL to be specified in a container, called *modules pool*. It supports top-down decomposition. As shown in Figure 4(a), eight

system modules have been identified and defined in the VOG, based on the system requirements analysis.



(a) Modules (b) Productions

Figure 4: Visual Object Tool (VOG) and Formal Method Tool (RuleGenerator)

DTGS is the top level module of the system, representing the entire tollgate system. VDB acts as a database system, which manages all vehicles' registration and updated EzPay information. Admin is a user interface for administrators to manage information and interact with the tollgate system, such as query and registration. TollGate provides communication among VDB, Admin, and other modules. According to the information collected from an approaching vehicle at a tollgate, the tollgate queries the database to decide the price and action.

EzCard stores the account information for the registered vehicle. Sensor is the application interface to the EzCard reader in a tollgate. It reads the approaching vehicle's EzCard and sends the read information to TollGate for processing. TollGate updates the vehicle's account information in VDB after processing. Light functions as an indication to the driver to show whether the vehicle has an authorized EzCard. Light and Camera at a tollgate receive commands from the tollgate. Camera takes pictures of unauthorized vehicles.

Except DTGS, all other modules are considered as component-level modules which are self-contained and perform independent functions. If any of these modules are available in the component repository, they can be reused in the DTGS VEPL, rather than being redefined.

The modules may change with the progress of system development, because one may not be able to acquire or extract all the system requirements at the beginning, or customers may require for additional functions. In the following two steps, one can add, remove, or modify any necessary modules in the modules pool, and re-generate the VEPL. Therefore, the automatic generation and re-generation capability of the meta-tool provides an iterative and evolutionary methodology.

3.3 Rule Specification

To allow the meta-tool to generate a domain specific VEPL, one needs to specify a set of production to define the VEPL. Each production consists of a left graph and a right graph. The left graph specifies the modules to be decomposed into the modules defined in the right graph. The connection between two nodes semantically represents a possible control or dataflow between two modules. The meta-tool also allows each production to be associated with a piece of supplemental textual code that defines operations performed on the modules in the production.

The Dallas TollGate System (DTGS) can be defined by four production rules using the Formal Method Tool (RuleGenerator) in the framework, as shown in Figure 4(b). The four individual productions can each be defined within the RuleGenerator. As an example, Figure 5 shows Production <1> being defined with a left graph (LGraph1) and right graph (RGRaph1). DTGS represents the whole system, i.e. the top level of the VEPL, and is decomposed into VDB and TollGate. In the left graph of Production <1> in Figure 5, DTGS can be considered as an initial graph, i.e. λ, following the convention of graph grammars.

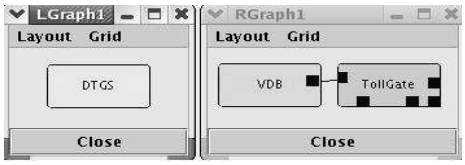


Figure 5: A DTGS VEPL production being constructed

All the four productions and associated action codes are listed in Figure 6. Production <2> defines the connection between TollGate and Admin, implying that administrators operate the tollgate system through Admin interface. Production <3> specifies that TollGate receives the vehicle information from Sensor, which in turn reads from EzCard. Production <4> defines the relationships among TollGate, Camera, and Light, such that Camera and Light both receive commands from TollGate and execute the commands.

In this example, the grammar for DTGS VEPL is in fact context-free since each of the four productions in this example has only one non-terminal node. This is, however, not the limitation of the RGG, a context-sensitive graph grammar formalism, which allows both the left and right graphs to have multiple non-terminal nodes.

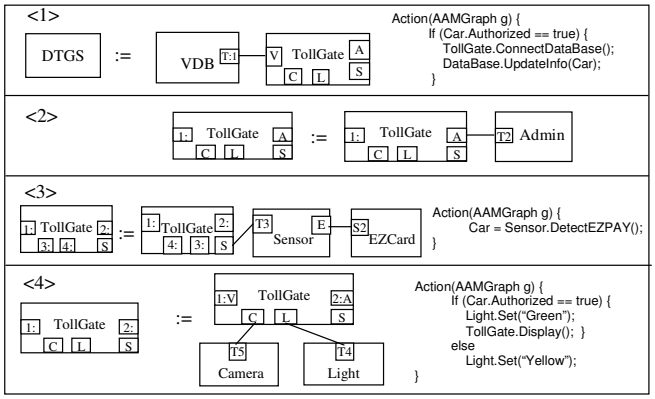


Figure 6: All productions defining DTGS VEPL

According to the above RGG productions defining the DTGS VEPL, the meta-tool framework automatically generates the VEPL programming environment for prototyping DTGS systems.

3.4 Prototyping Using the Generated VEPL

In the automatically generated VEPL programming environment, a domain programmer can draw a DTGS prototype as a graph and verify the configuration of the prototype. He/she can also execute the prototype to animate and observe its behavior.

The prototype has a central database, VDB, which stores the information of all the registered vehicles. Each tollgate has to receive information from VDB for any approaching vehicle and to determine the right action for the vehicle. The VDB-TollGate relation is determined by Production <1> as shown in Figure 6. Any connection between VDB and other modules would result in a violation of the prototyping configuration. Therefore the VEPL programming environment assures that the structure of the prototype meets the requirement.

The generated VEPL programming environment provides programmers a user-friendly interface for rapidly constructing prototypes by direct manipulation on graphic objects. For example, via an intuitive decomposition process, the programmer can select a module to be decomposed and replace it with a sub-graph representing a lower level configuration of the selected module. The relationship between the module and the sub-graph has been defined as a production rule by the software engineer. As shown in a snapshot of the prototyping process in Figure 7, the sub-graphs for decomposition are displayed in the right panel. The top two tollgates in the main drawing canvas are under construction while the bottom one has been constructed.

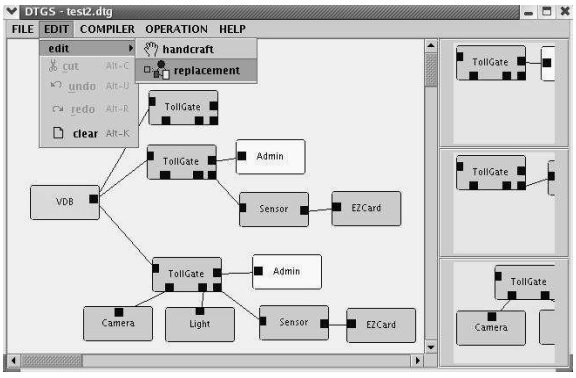


Figure 7: Prototyping in DTGS VEPL

The structural correctness (i.e. syntax) of the constructed tollgate prototype is automatically verified when the graph representing the prototype is parsed. The correct syntax of a constructed prototype does not, however, guarantee the correctness of the system. Executing the prototype possibly with animation of dynamic behavior is facilitated by the RGG parsing mechanism and syntax-directed computations. If the software engineer specifies the action code to perform logging or animation in each production, the generated VEPL system will be able to trace or animate the execution of a constructed DTGS prototype.

4 Related Work

Prototypes must be constructed and modified rapidly, accurately, and cheaply. The most primitive prototyping tools are paper and pencil. Since Computer-Aided Prototyping (CAP) was first coined [12], there have been many computer systems were developed to assist the prototyping process. CAPS [12] uses a specification language, PSDL (Prototype System Description Language), integrated with a set of software tools, including an execution support system, a rewrite system, a syntax-directed editor with graphics capabilities, a software base, a design database, and a design management system. CAPS is used for the rapid generation prototypes by combining dataflow (with real-time constraints) and control flows. Computer systems are described as networks of independent tasks that communicate through buffered data streams. In contrast to our solution, CAPS visualizes and executes the prototyping process based on a textual language, PSDL, and does not utilize formal methods during the process of prototyping. Compared to textual notations, graph formalisms seem to be more abstract, better structured and user-friendlier.

Focusing on prototyping distributed and concurrent programs, Ripple [6] has separate multilevel abstractions, including process-oriented level: the objects of the language are processes and operators that link processes to achieve

some form of control, function-oriented level, and implementation-oriented level. Ripple is formally defined using algebraic semantics [1]. The formal definition of the three abstraction levels allows transformation among the different levels. There are three different sub-languages corresponding to the three abstractions levels and are all textual languages. Similar to Ripple, the RGG also provides two levels of abstractions of prototyping, but has a uniform graph formalism for specifying prototyping languages.

The recently developed SLAM [7] supports an effective use of formal methods in the prototyping process. Equipped with an expressive object-oriented specification language and a development environment, SLAM is able to generate efficient and readable code in a high level object-oriented language such as Java and C++. A SLAM prototype is reusable, but SLAM cannot generate executable code from the prototyping language.

To our knowledge, there has been no prototyping system using graph rewriting or transformation techniques. Graph formalisms have been widely used to improve the productivity of the software development and maintenance process since the late 1960's. Using graphs instead of trees in the data model, Klein *et al*. proposed the IPSEN (Integrated, Incremental, Interactive Project Support ENvironment) Meta Environment for the development of integrated software development environments [8]. IPSEN uses context-free graph grammars in its specifications and language generation tools. Graphical layout annotations and programmable views may be used to define additional diagrammatic representations. For all logical aspects that are outside scope of EBNFs, PROGRES [15] graph transformation systems are used. IPSEN is a software development environment for modeling and implementing software documents and their relationships rather than a prototyping system. Although IPSEN can be used as a requirement engineering tool, it cannot generate executable code.

5 Conclusions

Design and implementation of software systems become increasingly complex. The value of prototyping in software development is well recognized. Bernstein estimated that for every dollar invested in prototyping, one could expect a $1.40 return within the life cycle of the system development [2]. This paper has presented a visual language generation methodology for rapid software prototyping. The major advantages of the grammatical approach can be summarized as the following:

- The meta-tool capability allows executable prototyping languages (VEPLs) to be automatically generated according to the specification of the domain software characteristics. A VEPL can thus be readily created, modified and enhanced whenever the domain needs arise. Thus an iterative and incremental process is supported.
- Any non-disposable prototype conforming to the specifications can be visually constructed through direct manipulation by a domain programmer who needs not to know the detailed specification. Once the prototype is confirmed to meet the domain requirements, the full scale prototype program can be generated.
- Verification is naturally supported. The generated VEPL environment includes a syntax-directed visual editor that is capable of syntactic checking and function verification of any prototype constructed in the VEPL.
- A visual approach to specifying and prototyping domain software is more intuitive than the textual form. A programmer without any knowledge of formal methods would be able to construct well-formed domain software.

A main challenge in the visual language approach is the necessity of graph grammar expertise for writing VEPL grammars. It has been our aim to provide a semi-automatic tool for generating production rules according to sample VEPL prototypes and user-provided hints, and for direct user modification. We also plan to conduct empirical studies to compare our visual approach with traditional approaches based on textual prototyping languages for their efficiency and effectiveness in software prototyping.

6 Acknowledgement

The authors would like to thank the anonymous reviewers for their insightful comments which have helped improving the final presentation of the paper.

7 References

[1] G. Belkhouche, (1996) A formally Specified Prototyping System, *Journal of Systems and Software,* July.

[2] L. Bernstein, (1996) Forward: Importance of Software Prototyping, *Journal of System Integration – Special Issue on Computer Aided Prototyping*, 6(1), 9-14.

[3] M.M. Burnett, (2004) Visual Language Research Bibliography, http://www.cs.orst.edu/~burnett/vpl.html.

[4] G. Costagliola, V. Deufemia, F. Ferrucci, and C. Gravino, (2001) On the pLR Parsability of Visual

Languages, *Proc. 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, Stresa, Italy, 48-49.

[5] J. M. Drake, W. W. Xie, W. T. Tsai, I. A. Zualkernan, (1997) Approach and Case Study of Requirement Analysis Where End Users Take an Active Role, *Proc. 15th International Conference on Software Engineering*, Baltimore, Maryland, 17-23 May, 177-186.

[6] B. Genraci, (1993) *Formal Prototyping for Concurrent Systems*, PhD Dissertation, Tulane University, New Orleans, Louisiana.

[7] A. Herranz, J. Moreno-Navarro, (2003) Rapid Prototyping and Incremental Evolution Using SLAM, *Proc. 14th IEEE International Workshop on Rapid Systems Prototyping (RSP'03)*, 201-209.

[8] P. Klein, M. Nagl, and A. Schürr, (1999) The IPSEN Tools, In: H. Ehrig, *etc al*. (Eds.) *Handbooks of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, Vol.2, 215-264.

[9] J. Kong, K. Zhang, J. Dong, and G.L. Song, (2003) A Graph Grammar Approach to Software Architecture Verification and Transformation, *Proc. 27th Annual International Computer Software and Applications Conference (COMPSAC'03)*, Dallas, USA, 3-6 November, 492-499.

[10] O. Köth and M. Minas, (2002) Structure, Abstraction, and Direct Manipulation in Diagram Editors, In: M. Hegarty, B. Meyer, and N.H. Narayanan (Eds.), *Diagrammatic Representation and Inference*, LNAI 2317, Springer, 290-304.

[11] Luqi, R. Steigerwald, G. Hughes, V. Berzins, (1991) CAPS as a Requirements Engineering Tool, *Proc. Conference on TRI-Ada '91*, San Jose, CA, 75-83.

[12] Luqi and M. Ketabchi, (1988) A Computer-Aided Prototyping System, *IEEE Software*, 66--72.

[13] K. Marriott and B. Meyer, (1996) Formal Classification of Visual Languages, *Proc. International Workshop on Theory of Visual Languages*, Gubbio, Italy.

[14] B.A. Myers, (1990) Taxonomies of Visual Programming and Program Visualisation, *Journal of Visual Language and Computing*, 1, 97-123.

[15] J. Rekers, and A. Schürr, (1997) Defining and Parsing Visual Languages with Layered Graph grammar, *Journal of Visual Languages and Computing*, 8(1), 27-55.

[16] G. Rozenberg, (Ed.), (1997) *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol.1, World Scientific.

[17] I. Stuermer, (2002) A Contribution of Graph Grammar Techniques to the Specification, Verification and Certification of Code Generation Tools, *Electronic Notes in Theoretical Computer Science*, 72(2), http://www.elsevier.nl/locate/entcs/volume72.html.

[18] D-Q Zhang, K. Zhang, and J. Cao, (2001) A Context-Sensitive Graph Grammar Formalism for the Specification of Visual Languages, *The Computer Journal*, 44(3), 186-200.

[19] Zhang, D-Q. Zhang, J. Cao, (2001) Design, Construction, and Application of A Generic Visual Language Generation Environment, *IEEE Transactions on Software Engineering*, 27(4), 289-307.