

A Generative Style-driven Framework for Software Architecture Design

Jun KONG

Kang ZHANG

Jing DONG

Guanglei SONG

The University of Texas at Dallas, Richardson TX 75083
{jxk019200, kzhang, jdong, gxs017800}@utdallas.edu

Abstract

Compared with texts, graphs are more intuitive to express comparative and structural information. Many graphical approaches, however, lack a formal basis for precise specifications in the design. This paper proposes a generative style-driven framework for software architecture specification based on a visual language formalism. In this framework, the designer uses graphical notations to define architectural styles. Given the graphical specifications, a visual language generator, i.e. a meta-tool, can automatically generate a specific design environment, which is suitable for the users without any knowledge of formal methods to directly manipulate software architectures by drawing box-line graphs. The graph transformation engine underlying the design environment can automatically validate structural integrity and reveal the hierarchical structure of a user-defined software architecture.

1. Introduction

Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure [18]. Being an important level of description for software abstraction, *software architecture* illustrates static and dynamic aspects of systems, such as the decomposition of a system into sub-systems, the overall organization, the assignment of functionality to components, and the protocol of interaction between components [24]. Supporting architecture-based development and evolution, various *architectural description languages (ADLs)* have been proposed to describe architectures using formal notations.

Though a formal description promises the definition of a software architecture in depth and forms a theoretical foundation for automatic verification and analysis, it requires considerable expertise. On the other hand, a language with a simple and well-

understood syntax, possibly with a graphical representation, provides a better communication between different developers. For example, box-and-line diagrams have been commonly used to describe software architectures. The lack of a formal basis in those diagrams, however, limits their applications. Without a precise semantics, it is not amendable to automatic analysis and transformation. The developer has to rely on personal experience to discover errors and inconsistencies in architectural diagrams. These processes are tedious and error-prone.

Some researchers have proposed formal frameworks to interpret box-and-line drawing with uniform definitions. Abowd *et. al.* [1] proposes a formal model specified through the Z language to interpret architectural styles. An alternative approach is using *graph transformations* [22]. With the capability of formally specifying and verifying syntax and semantics in a visual fashion, graph transformation has gained much attention in recent years [9]. For example, graph transformation is used to interpret UML state charts with formal semantics [16] and construct software development environments [9].

Applying graphical notations to conceptualize and model systems, visual languages frequently use nodes to represent components and edges to represent pre-determined relationships between components. By supplementing a graph transformation system with 1) a start graph, and 2) a distinction of terminal and non-terminal nodes, a *graph grammar* abstracts essential properties shared by a class of diagrams. Used to specify common structural features of software architectures, graph grammars are suitable for defining architectural styles [21].

With a theoretical foundation of graph grammars, we can validate the structural integrity and discover the hierarchical structure of a graph which conceptualizes a software architecture. More specifically, a graph grammar consists of a set of graph transformation rules, which dictate the way of constructing a complete diagram through a variety of nodes, since all possible inter-connections among the nodes have been stated in

the grammar. In other words, any connection in a valid configuration can be eventually derived from a sequence of applications of grammar rules. Conversely, an un-expected connection signals a violation on structural requirements. Therefore, an ill-formed architecture can be detected by parsing its graphical representation.

Though designers can describe a particular system with an arbitrary organization, good designers always tend to reuse some well-established architectural organizations, i.e. architectural styles. In general, a style guides the composition of components constructing an architecture, which satisfies certain topological constraints. It is natural to illustrate an architectural style through a graph grammar while a graph derived from the graph grammar denotes a valid architecture conforming to the style. However, the implementation of a graphical environment, which supports the design and manipulation of architectures of a particular style, is time-consuming and tedious. This issue is addressed by the application of a meta-tool technique [30]. Given the graphical specifications stating architectural styles, a graphical design environment can be automatically generated by a visual language generator [30], i.e. a meta-tool. A user without any knowledge of formal methods can directly manipulate software architectures by drawing box-and-line diagrams within the environment, where the underlying graph transformation engine is capable of verifying structural constraints within architectural styles.

Based on a graph grammar formalism, called the *Reserved Graph Grammar (RGG)* [29], this paper proposes a generative style-driven framework to specify software architectures. In summary, our grammatical approach facilitates the following aspects:

- The meta-tool capability automatically generates a graphical authoring environment for users without knowledge of formal methods; and
- Representing design policies and requirements through graphical notations, our approach is intuitive yet formal.

The rest of the paper is organized as follows. Section 2 introduces the Reserved Graph Grammar and its language generator. Section 3 demonstrates how to specify architectural styles. Section 4 presents the design of software architecture. Section 5 reviews related work, followed by conclusions and future work in Section 6.

2. A generative framework

This section first introduces the Reserved Graph Grammar (RGG) [29] being the underlying formalism for the specification of software architectures, and then briefly overviews the toolset for generating a graphic authoring environment.

2.1. Reserved Graph Grammars

Designers of complex systems typically use graphical methods as conceptual devices to organize their design space. Compared with text, graphs can represent semantic and structural information more intuitively. In most visual languages, programs are represented as graphs and thus called *visual programs*. Such graphs to be compiled/parsed are called *host graphs*. A visual programming environment includes a visual editor for graphical construction of host graphs representing visual programs and a parser for graph transformation. A visual language may be defined by a graph grammar [22], which consists of a set of rewriting rules called *productions*. Each production consists of two sub-graphs, called the *left graph* and the *right graph*. The nodes of the graphs symbolize some real world objects and edges between nodes represent pre-determined relationships. A sub-graph in the host graph is called a *redex* if it is isomorphic to the left or the right graph.

A graph transformation process is a sequence of applications of productions. Applications can be *L-application* or *R-application*. An L-application (or R-application) is to find a redex that matches the left (or right) graph and replace it with the right (or left) graph of a production. One obstacle restricting applications of graph grammars is that most proposed parsing algorithms are based on context-free formalisms. Many interesting graphs, however, cannot be specified by pure context-free grammars.

As the underlying formalism for specifying software architectures, the Reserved Graph Grammar (RGG) is a context-sensitive graph grammar formalism [29], which allows the left graph to have multiple nodes and thus is expressive in specifying various types of diagrams. The RGG formalism is expressed in a node-edge format, suitable for automatic analysis based on graph grammars. In a RGG, nodes are organized into a two-level hierarchy, where a large rectangle is the first level with embedded small rectangles as the second level called *vertices*. In a node, each vertex is uniquely identified. A node can be viewed as a module, a procedure or a variable etc., depending on the design requirement and granularity. A vertex acts as the connecting point of an edge. Edges

are used to denote communications or relationships between components. In addition to the two-level hierarchy structure of a node, the *marking* technique, which classifies vertices as marked and un-marked ones, addresses the embedding issue, i.e. building connections between a new sub-graph and the surrounding of a replaced sub-graph. Informally, the embedding rule can be stated as the following:

If a vertex in a production is unmarked and has an isomorphic vertex v in the redex of the host graph, then all edges connected to v should be completely inside the redex.

A marked vertex is identified by a unique integer, and preserves its associated edges connected to nodes outside a replaced sub-graph. The embedding rule makes the RGG expressive in system modeling and efficient in parsing graphs.

The RGG also supports syntax-directed computations so that a graph represented through RGG notations can be executable. The RGG is equipped with a deterministic parsing algorithm, called *selection-free parsing algorithm (SFPA)* [29]. Informally, the selection-free property ensures that different orders of applications of productions result in the same result. Zhang *et al.* [29] prove that a failed parsing path indicates an invalid graph, and thus SFPA is efficient with a polynomial parsing complexity by only trying one parsing path.

2.2. Separating architecture design and construction

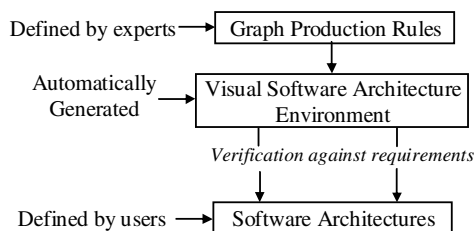


Figure 1. Support for two levels of responsibilities

The grammatical framework supports the separation of the responsibility of architecture design from that of architecture construction, like in the building industry where architects and constructors have separate concerns and responsibilities. This is achieved through a 2-level meta-tool concept. Once a software architect has provided the specification of an architectural style, a visual language with an environment supporting graphical construction of architectures will be automatically generated in the same fashion as generating textual languages using Lex/Yacc. A user of an application domain will be able

to use the visual environment to directly construct software architectures by drawing graphs. A graph transformation engine underlying the environment can verify design requirements. By providing a set of general architectural styles, e.g. “pipe and filter” and “client and server” etc, to the visual language generator, the graphical environment allows users to design an architecture making use of multiple styles. The 2-level meta-tool concept supporting separation of concerns is shown in Figure 1.

Therefore, more disciplined software development is encouraged by the 2-level process, i.e. the meta-tool [30] used by software architects and the visual design environment by domain users. The development can be iterative and incremental. Thus, our grammatical approach advocates a sound yet flexible design practice.

Development of visual languages and their supporting environments is time-consuming and tedious. Figure 2 presents a set of graphical tools, which support the generation of a graphical authoring environment for specifying software architectures built on the foundation of the RGG. Components and connectors are visually specified through the *visual object tool*, and are stored in a repository, which supports the reuse of existing designs. Based on the vocabulary of architectural elements, i.e. types of components and connectors, the *rule specification tool* is used to specify the static structural constraints in terms of a graph grammar. In other words, a graph grammar prescribes a family of architectures with common properties building on style-specific components and connectors. Having defined all required specifications, a *software architecture environment (SAE)* is automatically generated, and supports the graphical authoring of software architectures.

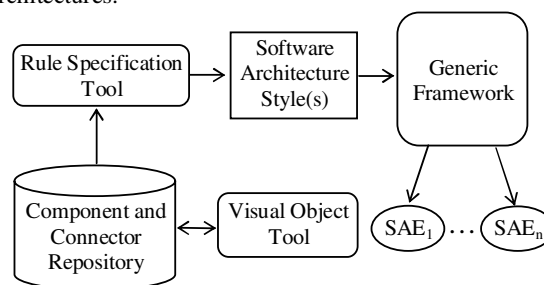


Figure 2. Toolset for generating an environment

In summary, our approach can generate from grammatical specifications a domain-specific SAE, which allows users to intuitively construct software architectures without expertise as required by most formal approaches and is capable of performing structural verification.

3. Designing architectural styles

Having introduced the Reserved Graph Grammar and its 2-level authoring framework, this section explains the definition from building blocks, i.e. components and connectors, to high level specifications, i.e. architectural styles.

Architectural styles fall in two broad categories [11]. The *idioms and patterns* refer to global organizational structures, i.e. pipe-filter style. The *reference models* include system organizations that prescribe specific configurations of components and interactions for specific application areas. A common pattern can be easily communicated and understood among a broad group of people while a reference model can be more efficient in a specific domain by enforcing domain-specific constraints. The following description illustrates the specification of a set of common patterns guiding the composition of a real-time system, and a reference model can be prescribed following the same principle.

3.1. Components and connectors

The building blocks of an architectural description language are: 1) components, 2) connectors, and 3) architectural configurations [18]. Components denote various units of computation, and connectors model interactions among components. A configuration representing an architectural structure is specified by connecting components and connectors. In our approach, nodes are used to represent components, and edges model connectors between components. A graph grammar “glues” various components and connectors into a meaningful architecture.

As described above, a node in the RGG is organized in a 2-level hierarchy. The name of a node denotes the type of a component, while vertices are used to define *port*¹ referring to the provided and required functionalities. Connectors used to model interactions among components [18] are specified in the same fashion as components. Vertices in a connector node denote roles of communication protocols.

3.2. Architectural styles

Having a vocabulary of architectural elements, i.e. component and connector types, a graph grammar specifies an architectural style, which denotes

constraints on configurations of architectural elements [20]. A graph grammar is made up of a set of productions, and each production illustrates the composition of sub-systems from the right graph to the left graph. All possible inter-connections between individual components have been defined in the graph grammar. Any legitimate connection can be derived from a sequence of applications of grammar rules. Conversely, an un-expected link signals a violation on the graph grammar. Therefore, parsing a host graph representing an architecture can validate the structural integrity. The parsing process is a sequence of R-applications, which is modeled as recognize-select-execute. This process is continued until no production can be applied. If the host graph is eventually transformed into an initial graph, the parsing process is successful and the host graph is considered to represent a valid architecture satisfying the structural requirements enforced by the graph grammar.

The pipe-filter style is made up of pipes and filters. A filter having a set of input and output ports reads streams of data on its input and produces streams of data on its output. A pipe having a source role and a sink one transforms information from the output of one filter to the input of another one. Building on pipe and filters, an architecture committing to the pipe-filter style must respect some structural constraints, e.g. a source role needs to attach to an input port and a sink role to an output port.

In order to investigate whether a user-defined architecture observes configuration constraints of a style, a graphical representation of an architecture friendly to end-users needs to be automatically transformed into a node-edge diagram, which is suitable for the RGG graph transformation engine analyzing structural integrity. In general, a node in the RGG indicates a component (a connector) and the vertices within the node represent ports (roles). For example, a pipe is represented by a node labeled *Pipe*, which has two vertices named *Sink* and *SRC* denoting the sink and source roles respectively (the vertices should be named with a clear meaning and vertex labels within the same node are distinct from each other). In order to denote a filter with an arbitrary number of ports, we represent a filter through a graph instead of a single node as the following:

- A node labeled *Filter* with two vertices *I* and *O* represents the filter component.
- Nodes labeled *I_Port* with two vertices *P* and *F* indicate the input ports within a filter component. An edge connecting the *F* vertex of an *I_Port* node and the *I* vertex of a *Filter* node explores the belonging relationship between an input port and a filter. The other vertex *P*

¹ ADLs may differ in the terminology of ports. For example, an interface in UniCon [27] is a *player*.

attaches to the *SRC* vertex of a *Pipe* node representing a data flow.

- The output ports are processed in the same fashion as the input ports.

Figure 3(a) shows an architecture of the pipe-filter style. According to the above principle, a corresponding internal graphical representation used by a RGG parser is automatically generated as shown in Figure 3(b).

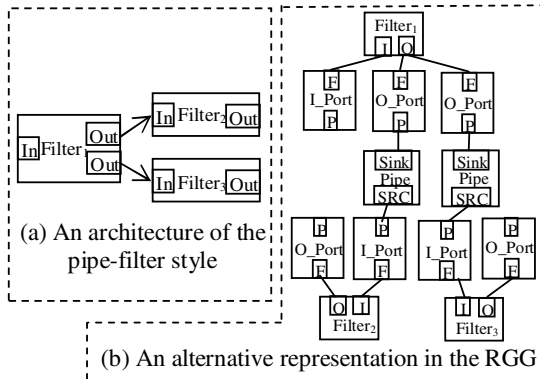


Figure 3. Graphical representations of an architecture

The necessary connectivity among components is stated through the right graph of a production, i.e. the application condition, which a host graph must fulfill. Application conditions, however, cannot express the condition which cannot be present for a production to be applicable. Therefore, the *negative application condition* [10] is introduced. For example, Production 1 in Figure 4 uses the negative application condition, expressed by a rectangle crossed by a line, to define the condition that an initial graph is derived from a single component *Filter* without any other component.

Figure 4 gives a grammatical specification prescribing a pipe-filter style. In particular, Production 1 demonstrates that an initial graph denoted by λ is abstracted from a *Filter* node. Production 2 abstracts two filters into one filter. Production 3 states a data flow between a pair of pipes.

The client-server style has two types of components, i.e. the server and the client. The server has the powerful capability to provide services to clients. A dispatcher being a connector (represented by a node labeled *Dis* as shown in Figure 5) serves to dispatch requested services to appropriate clients. Figure 5 presents the rules constructing a class of architectures committing the client-server style.

Components in an event-based style, represented by *Object* nodes, interact with each other through event broadcast, i.e. the occurrence of an event can invoke methods in components. The connector *distributor* takes announced events and transforms them into

method invocation. Since an *Object* can associate to an arbitrary number of events and methods, we represent an object through a graph in the same fashion as a filter. Figure 6 illustrates a graph grammar defining the structural properties shared by architectures within the event-based style.

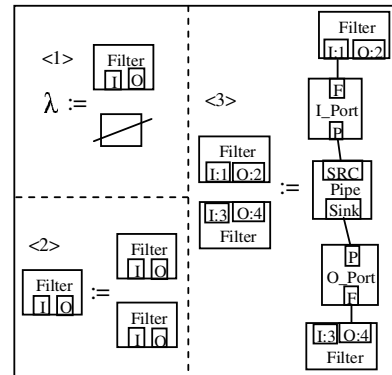


Figure 4. A graph grammar defining the pipe-filter style

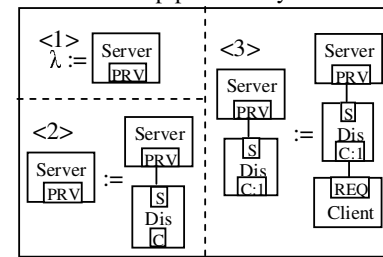


Figure 5. A grammatical specification prescribing a server-client style

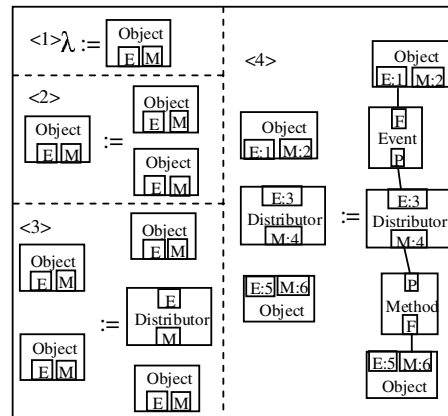


Figure 6. A graph grammar defining the event-based style

4. Designing an Architecture

Supporting a set of general architectural styles, a graph transformation engine underlying a generated graphical environment can validate user defined architectures against architectural styles. The following section goes through a toll gate example to explain designing a system using multiple general architectural styles.

4.1. Toll-Gates

In a road traffic pricing system, drivers of authorized vehicles are charged at toll gates 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. The registration of an authorized vehicle having an *ezpay* includes owner's personal data (such as name, date of birth, driver license number, bank account number and vehicle registration number).

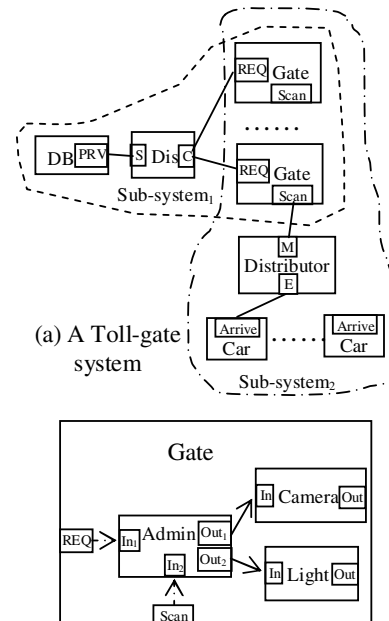
Each toll gate has a sensor that reads *ezpay*. The information read is stored by the system and used to debit the respective accounts. When an authorized vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorized vehicle passes through it, a yellow light is turned on and a camera takes a photo of the vehicle's license plate.

4.2. Designing a Toll-gate system

Based on a set of general architectural style defined through graph grammars, a visual authoring environment can be automatically generated. In the authoring environment, users without any knowledge of formal methods can define software architectures by drawing graphs. The structural integrity of such graphs can be validated by a graph grammar parser. In the environment, users can design a system or sub-system by choosing an appropriate style and customize components and connectors inherited from a vocabulary within the style.

A toll-gate system is made up of a database system and several gates. The database stores customers' information, which can be updated and retrieved upon requests coming from the gates. Such a communication model is implemented as the sub-system₁ in Figure 7(a) using a server-client style. The node labeled *DB* inherited from the component type *Server* represents a database, and nodes labeled *Gate* inherited from the type *Client* denote toll gates. The node *Dis* represents a connector sending requests to the database and dispatching replies to appropriate toll gates.

A toll gate needs to scan the identification of an arriving vehicle. The sub-system₂ applies the event-based style to implement the interactions between toll gates and vehicles (The node *Gate* is a common component in the sub-system₁ and sub-system₂, and it is inherited from both the *Client* type in the client-server style and the *Object* type in the event-based style). In particular, the vertex named *Arrive* in the *Car* node is an event port invoking the scanning operation denoted by the *Scan* vertex in the *Gate* node.



(b) The architecture of a gate

Figure 7. An architecture of the Toll-gate system

We can further elaborate the design of the toll gate using a pipe-filter style as shown in Figure 7(b). Three components constructing a gate are inherited from the filter type, and are denoted by nodes *Admin*, *Camera* and *Light* representing an administrator, a camera and a signal light. Directed edges in the Figure 7(b) denote pipes connecting different filters.

Based on graph grammars specifying a set of architectural styles, users can implement sub-systems with an appropriate style and incrementally glue sub-systems into a complete architecture. The structural integrity of each sub-system can be verified through a graph parser. For example, Figure 8(a) shows an architecture violating the pipe-filter style, which requires that one pipe should connect an output port to an input port between a pair of filters. The violation can be attacked by the RGG parser. Figure 8(b) gives the corresponding representation in the form of the RGG. Since the graph in Figure 8(b) can never match

to any production in Figure 4, the architecture cannot be abstracted to an initial graph. A failed parsing indicates an invalid software architecture.

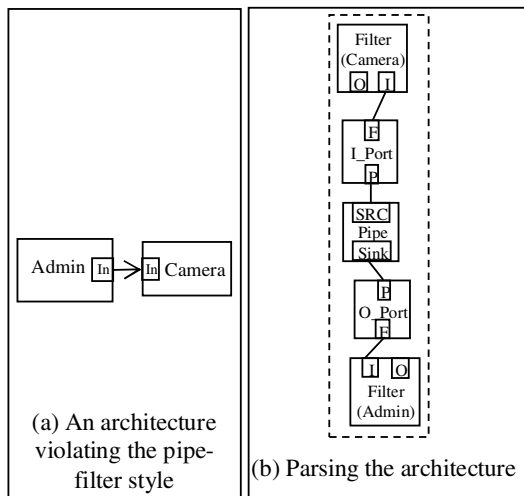


Figure 8. Checking an architecture

5. Related work

Many ADLs, such as Wright [2] and Rapide [17], have been proposed to model and analyze software architectures. Based on formal models, those languages allow users to define software architectures without ambiguity, and thus are suitable for automatic reasoning on architectures. There is, however, a mismatch between the abstraction level at which users usually model the software architectures and the abstraction level at which users should work with these languages [3]. In order to model software architectures using those languages in their current forms, users need expertise with a solid technique background.

With the meta-tool capability, our approach can overcome the problem by automatically generating a style-specific design environment. Then, users without any formal method knowledge can directly specify and manipulate software architectures in terms of box-and-line drawings. Based on well-established theoretical foundations, a graph transformation engine underlying the environment can verify the structure of user-defined architectures. Therefore, our approach is visual yet formal.

The Unified Modeling Language (UML) [5] provides a family of design notations to model various aspects of systems. Being a general purpose modeling language, the UML has also been applied to defining software architectures. Medvidovic *et. al.* systematically presents two strategies to model

software architectures in the UML [19]. Focusing on the usability of concepts, Garlan *et. al.* proposes several alternatives to map concepts from ADLs to the UML [12]. Both works exhaustively discuss the strength and weakness of each method. Emphasizing on the analysis and validation of designed models, Baresi *et. al.* uses the UML notations to specify the static aspect of structural styles paired with graph transformation for modeling dynamic reconfiguration [3]. Selonen and Xu apply UML profiles to software architecture design process and software architecture description [23]. Such profiles, called *architectural profiles*, define the structural and behavioral constraints and rules of the architecture under design, and are used to drive, check and automate the software architecture design process and the creation of all architectural views.

In general, class diagrams provide a declarative approach to defining instances of an architectural style. On the other hand, the Reserved Graph Grammar proposes a grammatical approach to specifying architectures in a constructive and incremental fashion. Though the declarative approach is easier to understand, the constructive and incremental method is more suitable for analysis. Furthermore, parsing an architecture can reveal the hierarchical structure of the architecture.

Some researchers apply the typed graph approach to define architectural styles. For example, Wermelinger and Fiadeiro [28] use typed graphs to specify all possible connections between components. Briefly, a typed graph $\langle G, \tau \rangle$ is a graph G equipped with a morphism $\tau: G \rightarrow TG$, where TG is a fixed graph, i.e. the type graph [7]. The typed graph approach also leads to a declarative fashion as the UML. We argue that graph grammars are more expressive in specifying architectural styles than typed graphs by associating attributes to nodes.

Formalizing graphs through set theories, Dean and Cordy [8] apply the diagrammatic syntax to express software architectures, and use it for pattern matching. Their work focuses on exploiting the composition of software architectures. Taentzer *et. al.* [26] uses the distributed graph transformation to specify dynamic changes in distributed systems. The changes are organized in a two-level hierarchy. One is related to the change in a local node and the other to the structure of the distributed system itself. This work emphasizes on modeling dynamic changes of distributed systems rather than specifications of structural composition.

Métayer [21] presents a formalism for the definition of software architectures in terms of graphs. Dynamic evolution is defined independently by a coordinator. Métayer's approach only allows a single node to be replaced with a sub-graph, and thus is

limited to those graphs, which can be specified by context-free graph grammars. On the other hand, our approach is more expressive in specifying software architectures by allowing arbitrary graphs in both left and right graphs of a production. Furthermore, our methodology is supported by a set of tools. Using the Reserved graph grammar as the underlying formalism, a visual language generator [30] can automatically generate an application-specific design environment. The environment is able to verify the structural properties of software architectures in terms of graphs.

Based on the theoretic foundation of term rewriting systems, Inverardi and Wolf [13] apply the *Chemical Abstract Machine (CHAM)* [4] model to the architectural description and analysis. Briefly, software systems are viewed as chemicals whose reactions are controlled by explicitly stated rules. Wermelinger [27] further proposes two CHAMs, i.e. the creation CHAM and the evolution CHAM, to define the architectural style and the reconfiguration respectively.

Karsai *et al.* [25, 15] propose the Model-Integrated Computing (MIC) to address essential needs of embedded software development. The MIC uses domain-specific modeling languages, which are tailored to the needs of a particular domain, to represent static and dynamic properties of a system. Similar to the meta-tool capability of our approach, the MIC supports to program a meta-programmable generic modeling environment into a domain-specific environment, which only allows the creation of models complying with the abstract syntax of a modeling language. Instead of using the UML class diagram as the meta-model to define the abstract syntax of a domain-specific modeling language, we apply a rule-based paradigm, i.e. a RGG grammar, to define a language. Supported by a formal basis of graph grammars, the rule-based specification is more suitable for reasoning and verification.

Our work is also inspired by the development of the Aesop system, the effort of adapting the principles and technology of generic software development environment to provide style-specific architectural support [11]. The Aesop system defines a style-specific vocabulary of design elements by specifying subtypes of the seven basic architectural classes. Then, designers have to over-load the methods of these subtypes to support stylistic constraints. Taking the advantage of conciseness and intuitiveness of graph transformation, our approach supports a high level specification of architectural styles through graph grammars.

6. Conclusions and future work

The grammatical approach is promising in providing an intuitive yet formal method to the specification of software architectures. Graph grammars are used to define architectural styles, which impose constraints on the interaction among components. With the well-established theoretical foundation, the grammatical approach can automatically validate the structural integrity and reveal the hierarchy of a user-defined software architecture through parsing.

The graph transformation tool can be considered an authoring language generator, which can generate a specific design environment whenever needed. A software engineer without any knowledge of formal methods will be able to use the generated environment to design software architectures by drawing graphical structures. Syntax check can be automatically performed within the environment.

Our future work includes:

- **Formal methods:** This paper presents a grammatical framework to specify software architectures, and illustrates the validation of structural integrity through syntactic checking. Moreover, the framework is open and can incorporate other formal approaches. For example, by viewing evolution as transitions of system states, the *model checking* technique applied in the automatic of large and realistic systems [3, 6] can be integrated with our framework.

- **Spatial Graph Grammar:** Due to the visual nature of visual languages, the spatial information not only contributes to the representation, but also intuitively conveys structural and semantic constraints. For example, an order over a set of objects can be directly specified according to their spatial locations (e.g. the left object has a smaller index than the right). We have proposed a *Spatial Graph Grammar (SGG)* [14] formalism, in which both the connectivity and spatial relationships construct the pre-condition of a graph transformation. We are currently implementing the visual language generator specific to the SGG.

- **Semi-automatic definition of architectural styles:** Writing productions and their action codes to handle reconfiguration is not an easy task, even for a design expert, since it requires a good command of the grammar formalism. It has been the authors' goal to partially automate the production authoring according to users' requirements.

7. References:

- [1] G. D. Abowd, R. Allen, and D. Garlan, "Formalizing Styles to Understand Descriptions of Software Architecture", *ACM Transactions on Software Engineering and Methodology*, 4(4), pp.319-364, 1995.
- [2] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transaction on Software Engineering and Methodology*, 6(3), pp.213-249, 1997.
- [3] L. Baresi, R. Heckel, S. Thöne, D. Varró, "Modeling and Validation of Service-Oriented Architectures: Application vs. Style", *Proc. ESEC/FSE'03*, pp.68-77, 2003.
- [4] G. Berry and G. Boudol, "The Chemical Abstract Machine", *Theoretical Computer Science*, 96, pp.217-248, 1992.
- [5] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.
- [6] J. S. Bradbury and J. Dingel, "Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems", *Proc. ESEC/FSE'03*, pp. 78-87, 2003.
- [7] A. Corradini, U. Montanari, and F. Rossi, "Graph Processes", *Fundamenta Informaticae*, 26(3-4), pp.241-265, 1996.
- [8] T. R. Dean and J. R. Cordy, "A Syntactic Theory of Software Architecture", *IEEE Transactions on Software Engineering*, 21(4), pp.302-313, 1995.
- [9] H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenburg (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, Vol.2, 1999.
- [10] C. Ermel, M. Rudolf, and G. Taentzer, "The AGG Approach: Language and Environment", *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, Vol.2, pp. 551-604, 1999.
- [11] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Styles in Architectural Design Environments", *Proc. 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pp.175-188, 1994.
- [12] D. Garlan, S. W. Cheng, and A. J. Kompanek, "Reconciling the Needs of Architectural Description With Object-modeling Notations", *Science of Computer Programming*, 44, pp.23-49, 2002.
- [13] P. Inverardi and A. L. Wolf, "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model", *IEEE Transactions on Software Engineering*, 21(4), pp.373-386, 1995.
- [14] J. Kong and K. Zhang, "Spatial Graph Grammars for Graphical User Interfaces", Technical Report, Computer Science Department, The University of Texas at Dallas, October 2003.
- [15] G. Karsai, J. Sztipanovits, A. Ledeczki, and T. Bapty, "Model-Integrated Development of Embedded Software", *Proceedings of the IEEE*, 91(1), pp.145-164, 2003.
- [16] S. Kuske, "A Formal Semantics of UML State Machines Based on Structured Graph Transformation", *Proc. UML 2001*, pp.241-256, 2001.
- [17] D. Luckham, J. Kenney, L. Augustin, J. Vena, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, 21(4), pp.336-355, 1995.
- [18] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, 26(1), pp.70-93, 2000.
- [19] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling Software Architectures in the Unified Modeling Language", *ACM Transactions on Software Engineering and Methodology*, 11(1), pp.2-57, 2002.
- [20] N. R. Mehta and N. Medvidovic, "Composing Architectural Styles From Architectural Primitives", *ESEC/FSE'03*, pp.347-350, 2003.
- [21] D. L. Métayer, "Describing Software Architecture Styles Using Graph Grammars", *IEEE Transactions on Software Engineering*, 24(7), pp.521-533, 1998.
- [22] G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, Vol.1, World Scientific, 1997.
- [23] P. Selonen and J. Xu, "Validating UML Models Against Architectural Profiles", *Proc. ESEC/FSE'03*, pp.58-67, 2003.
- [24] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, 21(4), pp.314-335, 1995.
- [25] J. Sztipanovits and G. Karsai, "Generative Programming for Embedded Systems", *Proc. Generative Programming and Component Engineering*, LNCS2487, pp.32-49, 2002.
- [26] G. Taentzer, M. Geodicke, and T. Meyer, "Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems", *Proc. 6th International Workshop Theory and Application of Graph Transformations*, LNCS 1764, pp.179-193, 1998.
- [27] M. Wermelinger, "Towards a Chemical Model for Software Architecture Reconfiguration", *Proc. 4th*

International Conference on Configurable Distributed Systems, pp.111-118, 1998.

[28] M. Wermelinger and J. L. Fiadeiro, "A Graph Transformation Approach to Software Architecture Reconfiguration", *Science of Computer Programming*, 44, pp.133-155, 2002.

[29] D. Q. Zhang, K. Zhang, and J. Cao, "A Context-Sensitive Graph Grammar Formalism for the Specification of

Visual Languages", *The Computer Journal*, (44)3, pp. 187-200, 2001.

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