

Style-Based Software Architectural Compositions as Domain-Specific Models

Nikunj R. Mehta, Ramakrishna Soma, Nenad Medvidovic

Department of Computer Science

University of Southern California

941 W. 37th Pl., Los Angeles, CA, 90089-0781, USA

{mehta | rsoma | neno}@usc.edu

Abstract

Architectural styles represent composition patterns and constraints at the software architectural level and are targeted at families of systems with shared characteristics. While both style-specific and style-neutral modeling environments for software architectures exist, creation of such environments is expensive and frequently involves reinventing the wheel. This paper describes the rapid design of a style-neutral architectural modeling environment, ViSAC. ViSAC is based on Alfa, a framework for constructing style-based software architectures from architectural primitives, and is obtained by configuring Vanderbilt University's Generic Modeling Environment (GME). Users can define their own styles in ViSAC and, in turn, use them to design software architectures. Moreover, ViSAC supports the hierarchical design of heterogeneous software architectures, i.e., using multiple styles. The rich user interface of GME and support for domain-specific semantics enable interactive design of well-formed styles and architectures.

1. Introduction

Architectural styles codify the recurring design practices and successful organizations of software systems. Styles are the composition patterns of and constraints on the computing, interaction, and data elements of the architectures of similar software systems [26]. A number of styles have emerged in research and industry, such as publish-subscribe, client-server, peer-to-peer, and so on. Styles are considered useful for instituting high-level reuse and bring economy to the design of software architectures [22]. Modern software systems are often composed *hierarchically*, and frequently use *multiple* styles in an architecture [17]. A prominent reason for the increasing use of styles in the design of software architecture is the observation that a property proven about a style holds true in the architectures based on that style [12]. Further, styles provide analytical models for determining properties of software architectures such as performance and reliability [4].

In order to support the growing use of styles in software development, there is a pressing need for software engi-

neering techniques and tools that support systematic style-based architectural design. Past research has helped establish the utility of architectural styles for designing large-scale software systems by providing style-specific environments for architectural design (e.g., DRADEL [18]) and style-based architectural description languages (e.g., Weaves [11]). The creation of such techniques and associated tools is highly expensive and frequently involves reinventing the wheel. Moreover, such approaches do not take into account the basic similarities among architectural styles. More recent research has focused on style-neutral design environments (e.g., AcmeStudio [27]) and architectural description languages (e.g., xADL [5]). These generic approaches support user-defined styles and, therefore, eliminate the need to create new infrastructure and environments for every new style. Still, such approaches do not recognize the architectural primitives underlying different styles and style-based software architectures (architectures as a shorthand). The knowledge of such primitives could enable an architect to better understand the relationships between architectural styles. Moreover, in the absence of a flexible and compositional foundation for using styles, these approaches significantly limit an architect in designing architectures.

In the context of the Alfa project, our research is aimed at supporting the systematic composition of architectures for network-based systems from architectural primitives [19]. The Alfa framework provides a small set of architectural primitives and formalizes a theory for composing style and multi-style, heterogeneous software architectures [20]. This paper describes the rapid design of a style-neutral architectural modeling environment, ViSAC for Alfa. ViSAC is a domain-specific modeling environment (DSME) obtained by configuring Vanderbilt University's Generic Modeling Environment (GME) [14].

DSMEs, such as Matlab/Simulink [16], have been considered useful for capturing specifications in specific engineering fields [14]. Modeling environments that can be tailored for use in specific domains support the specification of custom meta models for tailoring the environments for use in those domains. ViSAC is specified as a custom meta model for one such configurable modeling environment, GME. This allows us to leverage existing investments in a design environment and enables the rapid

development of a rich user interface for visually modeling styles and architectures. GME is customized through the use of a custom meta model for Alfa, called *xAlfa*. ViSAC takes advantage of semantic analysis features of GME to check the well-formedness of models and, therefore, supports interactive design of styles and architectures.

ViSAC has been successfully applied to styles for network-based systems. In this paper, we illustrate the design of the architecture using hierarchical decomposition and multiple styles. ViSAC interactively helps prevent errors of malformations in models of styles and architectures. Hierarchical composition is supported by the concept of model hierarchy in GME. Moreover, GME’s rich user interface features, such as drag-and-drop, simplify model design. However, contrary to our initial hypothesis, GME’s sub-typing semantics are not suitable for instantiating styles in architectures and lead to extra modeling effort.

The rest of this paper is organized as follows. Section 2 briefly describes Alfa and its primitives. Section 3 discusses the GME paradigm for Alfa including its syntax and semantics. Section 4 illustrates the use of ViSAC in the design of an example hierarchical, multi-style architecture. Section 5 discusses the strengths and limitations of our approach. Section 6 discusses related work in the areas of domain-specific modeling, visual languages, and software architectures. Finally, Section 7 presents some conclusions of our approach and pointers to future work.

2. Alfa

Alfa, an *assembly language for software architectures*, is used to compose architectural styles and style-based architectures from architectural primitives [19].

Alfa’s primitives (written in **bold constant-width** in this paper) and their interrelationships have been formalized in a composition theory [20]. This composition theory supports uniform modeling of styles and style-based architectures using architectural primitives, and enables data type checking in architectures. In this theory, styles provide templates, i.e., parameterized types, that are instantiated in architectural elements.

Datum is the type of data items (used synonymously with control signals in this paper) exchanged in software architectures. In Alfa, both software components and connectors, i.e., **particles**, can be composed hierarchically. A **particle** is the locus of computing, and comprises **input** and **output** ports for interacting with its environment, which are, in turn, organized into disjoint partitions called **interfaces**. The behavior of a **particle** is expressed in terms of the relationships among events at its **input** and **output** ports. Each port defines a **datum** that it can transfer.

The means of interaction between **particles** are **ducts**, **relays**, and **birelays**. A **duct** contains two ends, which are either **input** or **output** ports. Every **duct** is a FIFO queue that provides two functions—**holds** and **loses**—to determine its behavior. A **relay** is a special

particle with pre-defined behavior to provides multi-point interaction and contains multiple **inputs** and **outputs**. Every **input** and **output** belongs to a unique **interface**. Another specialized **particle**, **birelay**, is available to support bidirectional communication. A **birelay** provides **twoway** interfaces for bidirectional communication, each of which is a combination of an **input** and an **output**. A **birelay** may contain multiple **initiator twoways** (where the bidirectional communication originates) and **terminator twoways** (where the bidirectional communication ends).

To support hierarchical composition, a **particle** may be composed from other **particle**, **duct**, **relay**, and **birelay** primitives. **Inputs** and **outputs** of a **particle** may be mapped to similar ports of its internal **particles**, **relays**, and **birelays**.

We use a graphical metaphor, shown in Figure 1, to partially render compositions of Alfa’s primitives. Each element of the graphical metaphor corresponds to an Alfa primitive. *Function* primitives (e.g. **holds** and **loses**) are represented as textual attributes of their *form* primitives (e.g., **duct**). Hierarchical composition of a **particle** is rendered as geometric containment of shapes corresponding to its parts. The mapping of a **particle’s** port to a port of its internal elements is recorded as an association between the ports. Finally, allowed **datums** are shown as undirected lines between ports and **datums**. This metaphor provides an informal basis for Alfa compositions.

3. A Domain-Specific Modeling Environment for Alfa

In designing the modeling environment for Alfa, called ViSAC, we employed three common strategies: *conceptual simplicity, explicitness, and liveness* [4]. Conceptual simplicity means that, the language features directly map the user’s perception of the corresponding notion in the domain. An example of conceptual simplicity in our notation is the use of geometric containment to denote the aggregation relationship between the containing **particle** and the contained primitives. Another example is using topological connections—**ducts**—to represent paths of interaction. Explicitness means that the relationships are showed explicitly within the visual language. Using connections to show port mappings is an example of how this

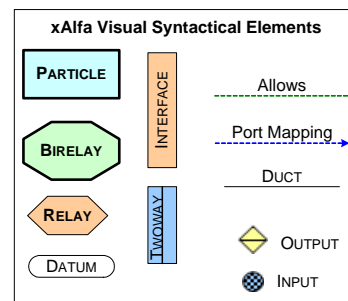


Figure 1. Graphical metaphor for Alfa

example, *datum reference* refers to *datum*. Inheritance relationships (shown as a triangle) are used to indicate concrete implementations of abstract kinds. The use of abstract meta model objects reduces the repetition of relationships across concrete objects derived from them. For example, **particle**, **relay**, and **birelay** can each be modeled as concrete *constituents*, such that their common relationships can be housed in a *constituent*.

Attribute specification. The third step of designing this meta model involves the identification of attributes of the meta model objects. Three kinds of attributes are available: enumerations, integers and fields. For example, **holds** and **loses** are recorded as attributes of a **duct**. Whereas **holds** is specified as an integer, **loses** is specified as an enumeration. The enumeration choices are specified according to Alfa's composition theory.

Constraint Specification. Specification of semantic rules of well-formedness in xAlfa enables checking styles and architectures for malformations. These semantic rules are defined in Alfa's composition theory [20] and mapped to GME's constraint specification notation, MCL. MCL constraints, limit meta model objects and their relationships. GME also defines a library of GME objects available during constraint checking. As examples, constraints for two rules of Alfa are shown below.

Rule:

Ports should be partitioned into disjoint groups of **interfaces** and **twoways**

MCL constraint on port:

```
self.memberOfSets(Interface)->size = 1 or
self.memberOfSets(TwoWay)->size = 1
```

Rule:

All allowed **datums** of ports are defined in the style.

MCL constraint on DatumReference:

```
let style =
self.refersTo().parent().oclAsType(Style) in
let models = style.models() in
models->select(m:Model | m.referenceParts(
DatumReference).includes(self))->notEmpty()
```

To increase the liveness of ViSAC, the triggering of constraint verification is aimed at providing the earliest possible feedback to the architect. For example, the first rule is configured to be checked automatically when a model is closed, and when a port is included in or excluded from an **interface** (a set). Whenever semantic violations of xAlfa occur, ViSAC produces a listing of errors with pre-specified descriptions along with information about debugging the source of errors. Thus, if an architect creates a port without assigning it to an **interface**, ViSAC notifies the architect immediately of the error. The scope of some rules may go beyond an individual modeling object, e.g., the first second above. Hence, such rules cannot be automatically checked. All the rules, including those associated with user events, are enforced upon user command.

Aspect Specification. The final step in specifying xAlfa is identifying the aspect(s) associated with each meta model object. *Aspects* are the means of controlling visibility of model objects, by showing only a subset of model objects at a time. xAlfa defines three aspects: *interface*,

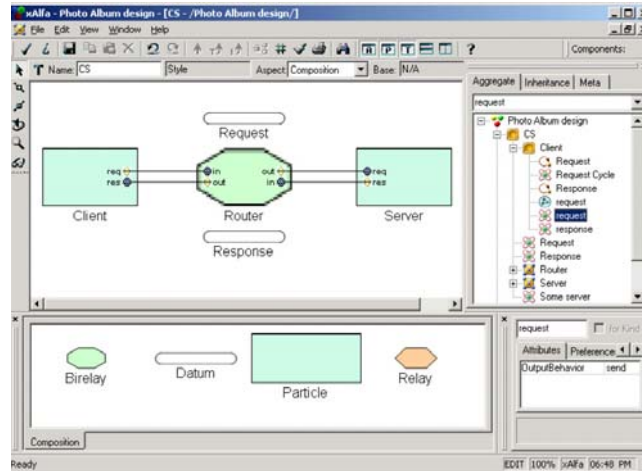


Figure 3. ViSAC showing client-server style

composition, and *constraint*. The interface aspect is used for defining ports, grouping them into **interfaces** and **twoways**, and for specifying their allowed **datums**. The hierarchical composition of **particles** and the specification of **ducts** and port mappings, is performed in the composition aspect. Finally, stylistic constraints and architectural behavior are defined in the constraint aspect.

Once the xAlfa paradigm is specified completely, the meta modeling interpreter produces a configuration for GME that supports modeling of styles and architectures in Alfa. Loaded into GME, this configuration produces ViSAC, as shown in Figure 3, customized for Alfa.

4. Modeling Styles and Architectures

To illustrate our approach of designing styles and architectures, we consider the architecture of an Internet-based photo album (PA) system shown informally as a box-and-line diagram in Figure 4.

PA comprises two types of clients: *image manager* for managing images, and *image viewer* for viewing images. Further, an *album server* stores and retrieves images, and, optionally, enhances images being stored. Multiple clients may be connected to an album server over a *network*.

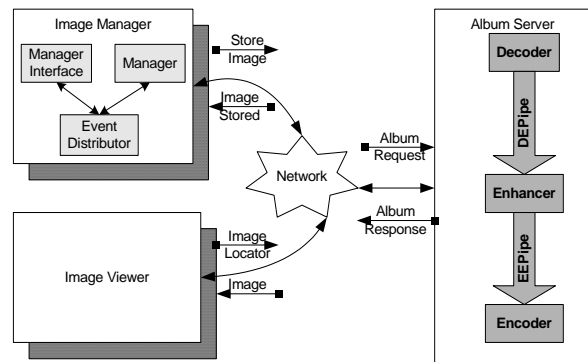


Figure 4. Architecture of photograph album software

Image viewer provides a visual interface for choosing an image to be displayed, and for displaying a chosen image. Image manager clients consist of a *manager* and a *manager interface* that communicate via *events*. Internal components of an image manager are used to select images to be stored on the server, as well as to select enhancements to be performed on a selected image. The album server also internally contains image *decoding*, *enhancing*, and *encoding* filters connected via *pipes*.

PA employs three different architectural styles—client-server (CS), in the top-level architecture; pipeline (P), in the album server; and event-based integration (EBI), in the image manager client. PA also requires hierarchical composition of its elements. The composition of PA’s architecture using Alfa’s primitives consists of two phases: style composition and architecture composition. In the first phase, we define the three styles used in PA. Each of the styles is recorded as a root-level model. For example, the pipeline style consists of two types of **particles** – *pipe* and *filter* as shown in Figure 5a. Moreover, the ports of *pipe* and *filter* are connected using five different **ducts**. Further, a *pipe* is decomposed into two **relays** – *opener* and *forwarder*, and the ports of a pipe are mapped to ports of these internal **relays**. Finally, pipe’s ports are divided into two **interfaces** – *source* and *sink*, as shown Figure 5b depicting the interface aspect of a *pipe*. This aspect also identifies the allowed **datums** of each port. If no **interface** is associated with a port, then ViSAC automatically generates an

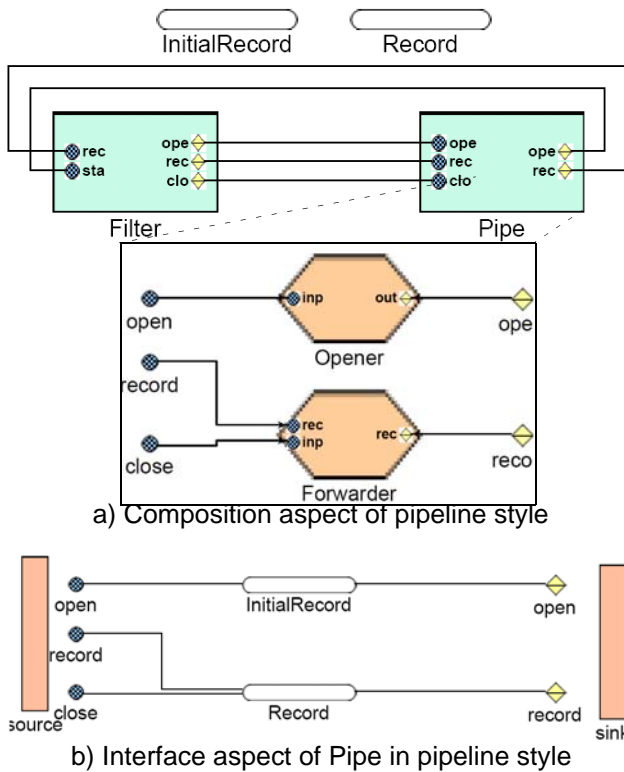


Figure 5. Pipeline style model

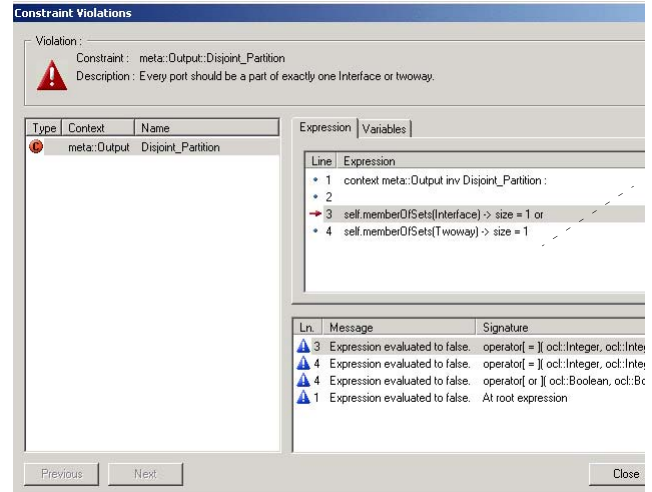


Figure 6. Example constraint violation in ViSAC

error message shown in Figure 6 due to the violation of the rule on disjoint partition of ports described earlier.

As the second step, required **datums**, **particles**, ports, and **interfaces** in PA are instantiated from stylistic templates. Figure 7 shows two levels of hierarchy of PA. The top level is composed using the client-server style (Figure 7a). *Album server* is composed using P (Figure 7b) and *image manager* using EBI (not shown) styles. In turn, every particle is decomposed into its parts, including ports, **interfaces**, and allowed **datums**, as shown in Figure 7c.

We had originally hypothesized that the instantiation of stylistic templates could be simplified through the use of GME’s sub-typing and instantiation features. For example, *store image* datum is obtained by sub-typing the *request* datum. However, in general this hypothesis could not be supported. This is due to the semantics of sub-typing that results in duplication of all the parts of the model being sub-typed, none of which can be deleted. Such sub-typing is not appropriate for **particle** templates. For example, the *encoder* component only requires a *read* interface, but instantiating *filter* would give it both a *read* and a *write* interface. Moreover, it is not possible to duplicate parts of a model in its sub types. This is required, for example, when defining two **interfaces** from the same **interface** template. As a result, the instantiation of templates in an architecture is more involved, and requires creating 1) a new **particle**, and 2) instances of port templates being instantiated, and 3) partitioning port instances into **interfaces** and **twoways**. The templates of **particles**, **interfaces**, and **twoways** are then inferred from the templates of their ports.

5. Discussion

ViSAC is used for modeling architectural styles and software architectures. We have used ViSAC for modeling twenty different styles for network-based systems and architectures using them. We find that model hierarchies in

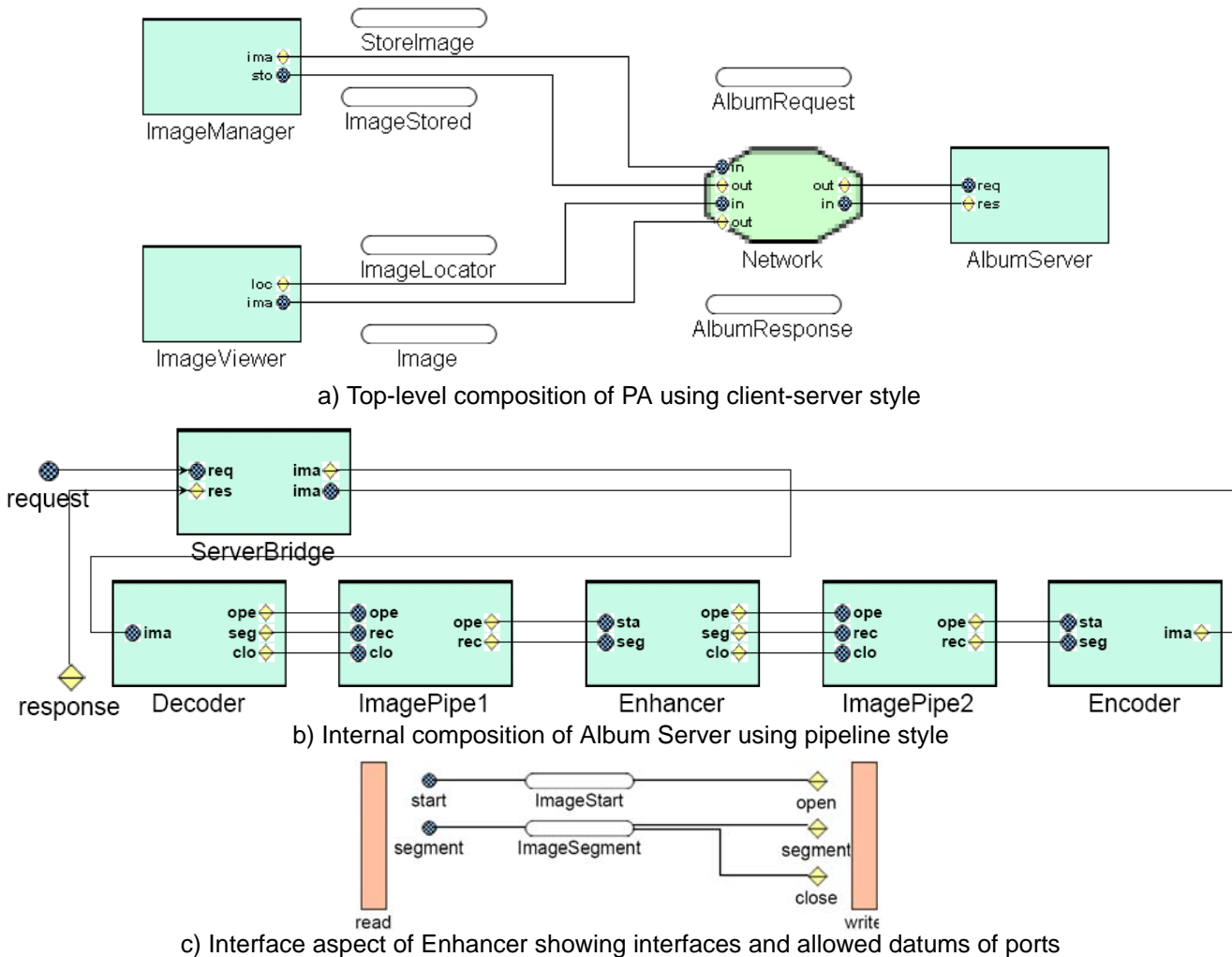


Figure 7. Photo album architectural composition

GME are highly suited for hierarchical composition of style-based architectures. ViSAC's support for systematically combining multiple user-defined styles in architectures is a significant improvement over existing techniques. Moreover, the xAlfa meta model also allows the construction of style-less architectures.

Visual modeling techniques and rich user interface features of GME make architectural composition using ViSAC truly visual. Deep copying and drag-and-drop editing ease the process of visual composition. Further, the use of aspects in GME allows the separation of concerns when modeling styles and architectures. ViSAC uses three different aspects 1) for composition of **particles**, 2) for interface specification, and 3) for specifying constraints and behavior.

GME's sub-typing techniques while useful for extending styles or combining existing styles into new hybrid styles, are inappropriate for instantiating styles in architectures. The mismatch is caused by Alfa's treatment of styles as templates rather than as complete types, to which GME cannot adapt its sub-typing semantics. As a result, much

effort is required for creating style-based architectures. Nevertheless, ViSAC's rich drag-and-drop features ease the modeling of Alfa compositions.

An important benefit of using GME for modeling compositions of architectural primitives is its support for recording rich semantics of visual languages. The result is that well-formedness of style and architectural compositions can be automated within ViSAC. The OCL-based notation used in GME for specifying dynamic semantics closely matches the set and relation-based composition theory of Alfa. In GME, constraint checking can be triggered off by a variety of user events. By correctly choosing events that trigger the checking of a GME constraint, ViSAC interactively helps the user prevent errors. More importantly, by breaking down the task of detecting malformation into tiny checks, each performed at different levels of hierarchy, this approach reduces the overall complexity of validating style and architectural compositions.

Defining the semantics of xAlfa using MCL, in most part proved to be easy and intuitive, mainly due to the

expressiveness of MCL. But we recognized a few shortcomings of GME in this regard. Firstly, the abstract kinds (for e.g., *constituent* in xAlfa) cannot be used in MCL expressions. This leads to much repetition and inelegantly specified constraints. Also, GME informs users of semantic errors through a pop-up dialog. This is in violation of principles of good design feedback, which affirms that it should be as inconspicuous as possible.

Style and architectural models can be composed only partially using visual techniques. An important component of style models are constraint expressions, which are recorded as ASCII text. However, certain stylistic constraints, such as on behavior, and architectural behavior are automata expressions, which could be captured as visual models themselves. We currently record all such expressions as text, leaving room for improvement in the future.

ViSAC can help create valid style-based architectures, but that alone is not sufficient for ensuring conformance of such architectures to their style(s). Style conformance verification requires additional steps to ensure that user-defined stylistic constraints are not violated by architectures using the style. Such rules are evaluated in an external engine, called *alfaac*. Alfaac is the compiler of Alfa models created using ViSAC, and extracts Alfa models from GME through its COM interface. This interface exposes the objects and relationships of xAlfa models through an object-oriented API. Alfaac verifies user-specified stylistic constraints on style-based architectures.

The availability of GME has allowed us to rapidly create a DSME for Alfa. Naturally, certain restrictions of GME have repercussions on its use for architectural modeling. The most important such restriction is that on the subtyping and model inheritance. Such restrictions lead to manual and tedious steps for instantiating architectural elements from style templates. Using GME infrastructure, custom components that manipulate GME models can be integrated to help reduce the effort involved in refining style templates. One such technique that we would like to explore is the use of wizard add-ons for specific tasks such as creating a `particle` from its template.

6. Related Work

While not a silver bullet for software [2], pictorial representations of software abstractions are known to greatly aid cognition [13]. Much research effort has been devoted to the study of visual languages (VL). VL is a term generally used to refer to *languages with alphabets consisting of visual representations that are used for human-human and human-computer interactions* [23]. VL research deals with various topics that range from model analysis based on the spatial and topological properties of the representations, to issues related with their efficient definition. VLS are commonly defined in terms of their syntax and semantics [23].

Our work draws greatly from the area of *domain-specific visual languages* (DSVL) [28]. This approach relies on the definition of VLS that correlate closely to the prob-

lem domain. Based on these VLS, domain models are built, which, in turn, can be transformed to other notations, including code. Tools such as GME [14], MetaEdit [21] and Dome [7] employ this approach. These tools use DVSLs to automatically generate DSMEs. These environments provide the capabilities to define a DSVL, and generate environments for domain-specific modeling in a given DSVL. We have chosen to use GME as our tool of choice over the other mentioned tools due to its unique combination of easy availability, its clear extension mechanisms, and close relationship to standard notations (UML and OCL).

Our work parallels UML 2.0 and its related standards in many ways. The four layer meta-modeling approach followed by GME is the same used by OMG to define UML [25]. MCL, the language used to define the semantics of our notation, is compliant with OCL 1.4 [24]. UML 2.0, as a part of its *Superstructure* specification [25], defines the syntax, semantics and the visual notation for representing software architectures (in terms of components, connectors, ports, interfaces and collaborations). Alfa does all of the above while also providing a framework for defining architectural styles and checking style conformance of the composed architectures. While UML aspires to be a “Universal” modeling language [8], the focus of Alfa is strictly on modeling styles and architectures. ViSAC could have been implemented as an extension to UML 2.0, but the lack of easily available techniques for customizing UML rule out any such possibility. However, when such techniques do become available, we will take advantage of them to construct a tool for Alfa.

Finally our work is closely related to the research in software architectural modeling environments. Most such tools tend to be style and notation specific. DRADEL for the C2 architectural style [18] and Weaves and its related toolset [11] are a few examples. ViSAC is style-neutral and aids an architect in defining architectural styles as well as architectures based on these styles. Generic tools such as ArchStudio[1] also support style-neutral architectural description, but do not allow the definition of styles. We also found the tree based user interface of ArchStudio to be cumbersome for hierarchical architectural compositions. The work closest to our own, is the research on AcmeStudio [27]. However, AcmeStudio is not based on any notion of architectural primitives, and does not support behavioral and data type information in architectures. Moreover, it does not support the use of multiple styles at the same level of architectural hierarchy or the use of style-less architectural elements, both supported by ViSAC.

7. Conclusions

This paper has discussed a novel approach for architectural modeling using a domain-specific modeling environment, ViSAC. The main objective of ViSAC is to facilitate the interactive design of style-based software architectures. ViSAC has been rapidly implemented by configuring

GME, a configurable modeling environment. Alfa's composition theory matches the concepts underlying GME: model hierarchy, atoms, connections, sets, references, multiple aspects, and constraints, thus simplifying the design of ViSAC.

Architectural styles and style-based software architectures are both modeled on the basis of a GME meta model, called xAlfa. The xAlfa meta model reifies Alfa's composition theory by defining the syntax and semantics of a visual language for modeling styles and architectures from Alfa's primitives. xAlfa's syntax enables the hierarchical composition of software architectures based on possibly multiple, user-defined styles. In addition to preventing malformed compositions, ViSAC can be easily extended to support the checking of style conformance of architectures according to their style(s). Moreover, we are integrating ViSAC with a compiler for xAlfa architectural models, called alfaac, to check stylistic constraint expressions defined by a style designer. Another important objective of alfaac is to generate source code for architectural compositions.

ViSAC has been used for modeling nearly twenty different styles for network-based systems. This paper illustrated the definition of three different styles and their use in modeling style-based architectures. ViSAC provides interactive support for designing styles and architectures by alerting the user to their malformation. However, the design of style-based architectures tends to be tedious as style templates do not lend themselves easily to GME's sub-typing rules. We are currently engaged in tiding over such differences by designing task-driven wizards that leverage support for customization in GME.

8. References

- [1] ArchStudio, <http://www.isr.uci.edu/projects/archstudio>
- [2] Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, 20(4), April 1987.
- [3] Burnett, M. M., "Seven programming language issues", Burnett, M. M., Goldberg, A., and Lewis, T. G., (eds.), *Visual Object-Oriented Programming*, Prentice Hall and Manning, Greenwich, CT, 1994.
- [4] Chen, M., Tang, M., and Wang, W., "Software Architecture Analysis - A Case Study", *Proc. COMPSAC '99*, Phoenix, AZ, October 1999.
- [5] Dashofy, E., van der Hoek, A., and Taylor, R. N., "An Infrastructure for the Rapid Development of XML-Based Architectural Description Languages", *Proc. ICSE-24*, May 2002.
- [6] Davis, J., "GME: The Generic Modeling Environment", *Proc. OOPSLA-18*, Anaheim, CA, USA, October 2003.
- [7] Dome, <http://www.htc.honeywell.com/dome>
- [8] Engels, G., Heckel, R., Sauer, S., "UML - A Universal Modeling Language?", M. Nielsen, D. Simpson (eds.), *Proc. ICATPN 2000*, June 2000, Aarhus, Denmark, LNCS 1825, Springer-Verlag, 2000.
- [9] Fielding, R., "Architectural Styles and the Design of Network-Based Software Architectures", Ph. D. Dissertation, University of California at Irvine, 2000.
- [10] GME 3 User's Manual, Version 3.0, Vanderbilt University, March 2003.
- [11] Gorlick, M. M. and Razouk, R. R., "Using Weaves for Software Construction and Analysis", *Proc ICSE 1991*, Los Alamitos, CA, USA
- [12] Jackson, D., "Automatic Analysis of Architectural Style", Unpublished Manuscript, MIT Laboratory for Computer Sciences, Software Design Group.
- [13] Kulpa, Z., "Diagrammatic representation and reasoning" *Machine Graphics & Vision*, 3, 1994.
- [14] Ledeczzi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P., "The Generic Modeling Environment", *Proc. WISP '01*, Budapest, Hungary, May 2001.
- [15] Ledeczzi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G., "Composing Domain-Specific Design Environments", *IEEE Computer*, November 2001.
- [16] Matlab/Simulink, <http://www.mathworks.com>.
- [17] Medvidovic, N., Mikic-Rakic, M., Mehta, N. R., and Malek, S., "Software Architectural Support for Handheld Computing", *IEEE Computer Special Issue on Handheld Computing*, September 2003.
- [18] Medvidovic, N., Rosenblum, D., and Taylor, R. N., "A Language and Environment for Architecture-Based Software Development and Evolution", *Proc. ICSE-21*, Los Angeles, California, USA, May 1999.
- [19] Mehta, N. R. and Medvidovic, N., "Composing architectural styles from architectural primitives", *Proc. ESEC-10/FSE-11*, Helsinki, Finland, September 2003.
- [20] Mehta, N. R. and Medvidovic, N., "Composition of Style-Based Software Architectures from Architectural Primitives", Technical Report USC-CSE-04-503, University of Southern California, 2004.
- [21] MetaEdit, <http://www.metacase.com>
- [22] Monroe, R. T. and Garlan, D., "Style-Based Reuse for Software Architectures", *Proc. ICSR-4*, Orlando, Florida, USA, April 1996.
- [23] Narayanan, N.H. and Hubscher, R., "Visual Language Theory: Towards a Human-Computer Interaction Perspective", Meyer, B. and Marriott, K. eds. *Visual Language Theory*, Springer-Verlag, New York, 1998, pp. 85-127.
- [24] Object Management Group, "Unified Modeling Language Specification, Version 1.4", September 2001.
- [25] Object Management Group, "UML 2.0 Superstructure Specification", August 2003.
- [26] Perry, D. E. and Wolf, A. L., "Foundations for the Study of Software Architectures.", *ACM SIGSOFT Software Engineering Notes*, 17, 40-52, 1992.
- [27] Schemrl, B. and Garlan D., "AcmeStudio: Supporting Style-Centered Architecture Development", Unpublished Manuscript, CMU School of Computer Science.
- [28] Tolvanen, J., Gray, J., and Kelly, S., (eds.), *ACM OOPSLA Workshop on Domain-Specific Visual Languages*, Jyvaskyla, Finland, October 2001.