Exploring Quality-driven Object-oriented Materializations for Software Architectures

J. Andrés Díaz Pace[1,2], Alejandra C. Diez[1], Marcelo R. Campo[1,2]
[1]ISISTAN Research Institute, Faculty of Sciences, UNICEN University,
Campus Universitario (B7001BBO), Tandil, Buenos Aires, Argentina
[2]CONICET, Argentina
e-mail: [adiaz, adiez, mcampo]@exa.unicen.edu.ar

Abstract

Design activities are critical in the development of quality software. Along this line, architecture-based design has been regarded as the right context for analyzing system-wide quality attributes and making principled design decisions therein. However, the materialization of architectural models into object-oriented structures where the decisions made at the architectural level can be reflected and implemented has not been satisfactory bridged yet. One of the reasons for this problem is that, given an architectural model and a set of quality drivers, there exist usually multiple different ways of mapping this input to object-oriented terms. Furthermore, the whole process involves considerable design background and experience. In this context, this paper describes a tool approach, called ArchMatE, to assist developers in the exploration of materialization alternatives.

1. Introduction

More and more, design activities are becoming critical in software development as they largely influence the quality of the final product [4]. A clear evidence of this trend is that today's software systems typically have to deal with many quality requirements (e.g., modifiability, portability, interoperability, availability, to name a few) whose realization is closely related to design-level solutions [5]. Furthermore, the growing complexity of these systems requires of high-level models to reflect the principal design decisions and also analyze them. In response to this situation, many researchers are starting to pay a great attention to software architectures as means to manage both quality attributes and system's complexity [4, 6].

At its essence, the software architecture level of design focuses on the gross system's organization in function of the quality attributes affecting the system. An important aspect of architectural models is that they prescribe
the organizational structure of the system being developed, mostly independent of particular implementation technologies. However, one of the challenges of the architecture-based approach is the *gap between architectural models and object-oriented structures*. On one side, architectural models provide suitable abstractions for the exploration of various design alternatives, considering different tradeoffs among quality attributes, although these models do not always lead to an object-oriented computational solution. On the other side, the object paradigm provides a nice implementation approach, but it shows certain inability to address quality-attribute issues in a broad sense. As pointed out by [28], this paradigm is influenced by some intrinsic characteristics of object abstractions (e.g., information hiding,

encapsulation, polymorphism) oriented to promote reusability or flexibility, rather than treating quality attributes as explicit design concerns.

Therefore, the transition from architectural designs to object-oriented counterparts results, in practice, a technically difficult problem that requires extensive design knowledge. In particular, a central issue in this transition is how to break the tradeoffs imposed by a functional and quality-oriented decomposition of a system versus a pure object-oriented decomposition of the same system. Certainly, given an architectural model and a set of quality drivers, there exist often multiple different ways of mapping this input to object-oriented terms. At this point, we believe that the apparent mismatch between architectures and objects does not deny the fact that the object paradigm is a very convenient technology to represent software architectures. We have termed this special relationship between architectures and objects as *object-oriented materialization of software architectures* [10].

Besides, along with the advances in software architectures, there is nowadays a pressing need for cost-effective design and prototyping tools supporting the development of architecture-based software [18, 20]. Although several approaches to derive implementations from architectural models based on functional requirements exist, very few of these techniques take quality attributes into account [7, 19, 27, 29, 31]. In particular, our previous experiences in the subject of proto-frameworks [10, 12] led us to envision an environment to provide this kind of automated support. Along this line, we present a tool approach that facilitates the exploration of object-oriented designs within a materialization process, concentrating on the main quality attributes associated with the original architectural model. Basically, the approach relies on a corpus of design knowledge about architectural structures, stylistic features associated to these structures, quality-attribute issues and object-oriented mechanisms, in order to identify links to different materialization strategies. On the basis of this knowledge, a tool called ArchMatE (ARCHitecture MATerialization Explorer) is capable of assisting the developer by sketching a collection of object-oriented classes that approximately fit the requirements of the architectural model under consideration, together with the developer's preferences.

The rest of the work is organized around 5 sections as follows. Section 2 gives some background about architecture-based design and object-oriented design. Section 3 provides a conceptual description of our approach for the exploration of object-oriented materializations, illustrated with a pipe-and-filter case-study. In section 4, we introduce the ArchMatE tool and describe its implementation. Section 5 discusses related work. Finally, section 6 analyzes lines of future research and rounds up the conclusions of the paper.

2. From Software Architectures to Objects

The software architecture community has promoted the construction of designs out of module-scale abstractions (e.g., components, connectors, responsibilities, mechanisms, etc.). At its essence, architectural abstractions rely on the understanding of the architectural means used to fulfill the main functional and quality requirements of the problem [4]. According to [21], these abstractions are primarily focused on: high-level system specifications, rich abstractions for interaction between architectural elements, and analysis of emergent quality-attribute properties.

Normally, architectural designs arise from the combination of a number of patterns of various sorts, called architectural styles [24], which are descriptions of typical ways of structuring software systems. Understanding and applying predefined styles in the development of architectural models is expected to save time and improve quality, when compared to searching for specific solutions on its own [9]. Basically, an architectural style defines a collection of related systems, providing a coherent vocabulary of design elements and rules for their composition. Furthermore, each style represents a package of design decisions with predetermined implications on quality-attribute issues. This can be reused across different application domains, although permitting some tailoring to fit specific developer's needs. Examples of well-known architectural styles are layers, blackboard, pipes and filters, objects and broker, among others [24]. In addition, some rules of thumb for choosing appropriate styles have been proposed [25]. As example, Figure 1 depicts a system in a pipe-and-filter style.

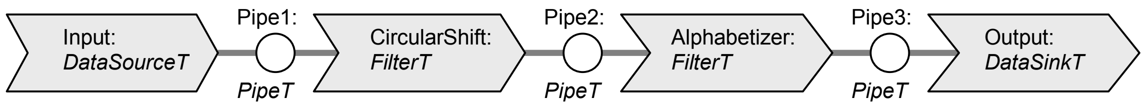

Figure 1. A typical pipe-and-filter system

Regarding the representation of architectural designs, a variety of special-purpose notations have surged during the last years. These languages, called ADLs (Architectural Description Languages) (see [18] for a survey), are mostly focused on the description of systems on the basis of components, connectors and interactions among them. The components represent the primary locus of computation, while the connectors act as mediators in communication and coordination activities among components. Besides specification, the ADLs give support for different kinds of analyses on the architectural information, and come with generative facilities that permit rapid development of systems.

From a different perspective, the object paradigm conceives a system as a collection of objects interacting via message sending [14]. At first glance, the object abstractions are often more related to detailed design or implementation than architectural abstractions. In terms of design, the way responsibilities are assigned to groups of objects is going to determine the flexibility/reusability of the resulting system. For this reason, we can say that the target quality attribute of the paradigm is principally modifiability, albeit ramifications to other quality attributes may exist. Nonetheless, despite this apparent limitation, the paradigm is equipped with abstractions that allow developers to implement useful object structures and collaborations, as is the case of design patterns and frameworks [13, 14]. A framework infrastructure for a given domain can be extended by the developer to get specific applications in such a domain. To illustrate this point, Figure 2 shows a small framework (sometimes called *framelet* [23]) for a pipe-and-filter style, implemented in terms of push-like filters arranged in a pipeline.

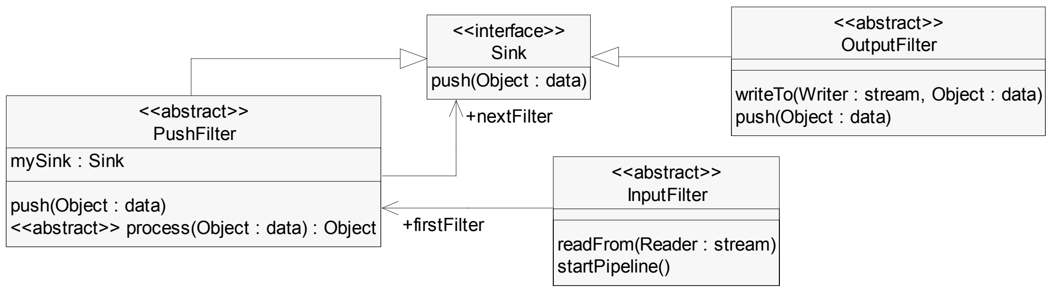

Figure 2. A framelet implementing a pipe-and-filter style

When it comes to the specification of object-oriented designs, the most popular visual modeling language standard is the UML [30]. UML provides several kinds of diagrams to describe different viewpoints of a system, and can also give directions to translate these models into code.

2.1. Towards a Linkage between Architectural and Object Models

From the above overview, we can see that both architectural and object abstractions tend to consider the following aspects: i) a vocabulary of constituent elements, ii) a set of rules to assemble partial fragments out of these elements, iii) a language appropriate to describe designs, iv) a rationale to explain the advantages/disadvantages of design decisions, and v) a collection of techniques to analyse the resulting designs. However, the two types of abstractions have different roles and capabilities within the design process.

The first observation is about granularity. Architectural models deal with the composition of modules and interactions among them in order to form systems. Instead of that, object models concentrate on flexible, reusable, and to some extent modular, implementations for a system, assuming that an architectural design for it has been outlined. Therefore, architectural mechanisms are good at abstracting away from implementation details, and serve to expose the quality properties that are most critical for the system's success, whereas object mechanisms are more suitable to address variability in implementation settings.

The second observation is about quality attributes. As many authors have mentioned, object models are likely to present difficulties to address the principle of separation of concerns [17, 22]. In particular, many of these concerns arise as a consequence of quality-attribute factors (e.g., concurrency, distribution, real-time constraints, location control, persistence and failure recovery, among others). Here, architectural models can provide a better context to identify relevant concerns and reason them at the very conception of the system architecture.

The appropriateness of software architectures to engineer software quality enables one to structure the design process in two phases, as outlined in [10, 12]. Initially, we can start with an architectural model, capturing the underlying organization of the system and the tradeoffs imposed by quality attributes in terms of predefined architectural styles. Then, using the architectural guidance, we can convert this high-level model into a lower-level object model (e.g., a framework or a final application). Eventually, design patterns may be applied to obtain more flexible object structures, leading to different implementation alternatives. Put it in other way, we are doing an *object-oriented materialization of a software architecture*.

Having subscribed to the materialization approach, a direct outcome of these ideas is how to use adequate design knowledge to guide the developer in the transformation of architectural designs into concrete implementations. For example, depending on the characteristics of the selected architectural model, some of the techniques to obtain an object-oriented representation of that model may include: direct mapping, constraint relaxation or constraint strengthening [10]. Although interesting in theory, it is quite hard in practice to derive object models by applying these techniques without considerable design background and experience. This is so because, even taking a small set of quality drivers for the materialization, there are many alternative ways to get a feasible combination of object-oriented fragments that satisfies the original architectural model. Hence, some kind of *tool support is required in order to keep the exploration of object-oriented design alternatives manageable*. When analyzing the main requirements for automating this exploration, we can distinguish the following levels:

- A categorization of architectural abstractions, taking into account the structural, behavioral and quality characteristics most relevant to each style - *architectural level*
- A collection of design rules to derive object models from architectural descriptions. These rules, although incomplete, should be based on the underlying structure of these

descriptions, but also consider context-specific information and quality-attribute issues - *mapping level*

- A number of techniques to cope with variability in object structures (e.g., design patterns, aspects, etc.), as architectural abstractions do not always have a direct correspondence with individual objects - *object level*

3. Automating the Exploration of Object-oriented Designs

During the design of any system, the developer usually deals with a *hierarchy of design decisions* that potentially impact several quality attributes of this system. Related to this hierarchy of decisions is the concept of *design space* [28], a multi-dimensional space where the developer searches for, selects, evaluates and combines partial solutions. Naturally, this line of reasoning applies to object-oriented materializations as well. That is, the developer can take advantage of particular architectural styles to identify ranges of design families, incorporating also quality-attribute properties, and afterwards use design patterns and frameworks to solve specific design situations within a given style. This way, we can approximate the exploration of object-oriented designs for well-defined architectural models.

At first, a naïve approach for automating the exploration process could be the generation of valid object-oriented configurations for a given architectural model through a standard search, comparing the configurations by means of some analysis technique. It is quite obvious that this initiative has few chances of success, due to the computational burden that such a search would imply. Nonetheless, we can certainly formulate design-oriented heuristics to traverse the design space, in order to narrow the bundle of possible designs to a subset of promising solutions. These heuristics can work reasonably well, if the alternatives are being generated on the basis of principled choices made by the developer. Therefore, assuming a design scope based on particular architectural styles along with predetermined quality-attribute goals, we can define a sort of "*materialization strategies*" to be followed by an automated tool in the exploration of object-oriented solutions. In this context, we propose an assistance schema that comprises the activities enumerated below (see Figure 3).

1. The developer is asked to define an architectural model, choosing the desired levels of quality attributes for this system.
2. A special design engine is in charge of generating an initial object model for that architectural model, applying a number of materialization strategies. To do so, the engine is based on a corpus of architectural information, focused on architectural styles and including some advice on their contributions to quality-attribute issues.
3. Once a preliminary solution reflecting the developer's preferences is identified by the engine, the developer can set the stage for a more detailed exploration of solutions. This would enable the engine to suggest a number of transformations and derive further object models as needed.
4. Finally, functionality is assigned to the resulting object model, according to the design goals and the characteristics of the problem domain.

Note that the use of architectural styles as foundation of the approach gives the developer a collection of component types together with a description of the interaction patterns among them. Nonetheless, these component types will have functionality to the extent necessary to implement the patterns of interaction. Depending on the style and the domain, the specific number of components and their responsibilities will vary according to the functionality to be implemented by the system.

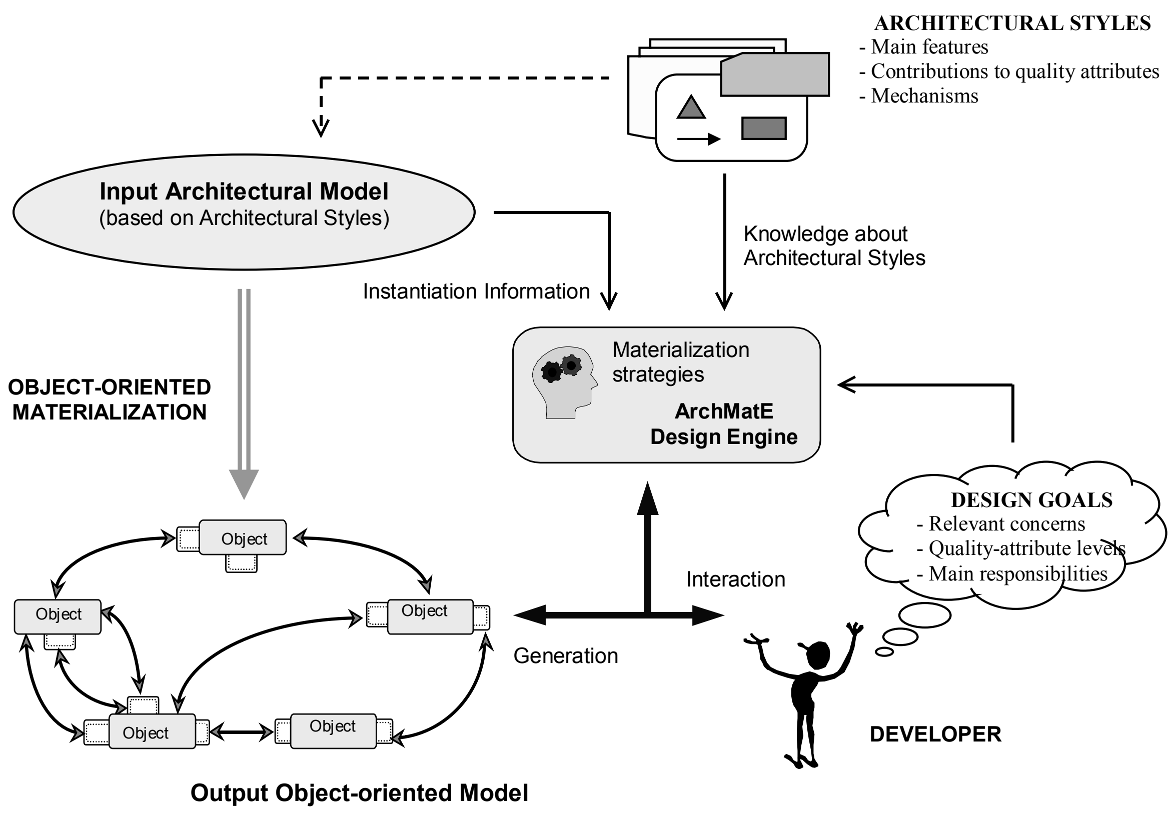
ARCHITECTURAL STYLES
- Main features
- Contributions to quality attributes
- Mechanisms
Input Architectural Model (based on Architectural Styles)
Instantiation Information
Knowledge about Architectural Styles
Materialization strategies
ArchMatE Design Engine
OBJECT-ORIENTED MATERIALIZATION
DESIGN GOALS
- Relevant concerns
- Quality-attribute levels
- Main responsibilities
Object
Interaction
Generation
DEVELOPER
Output Object-oriented Model

Figure 3. Assistive schema to explore the materialization of architectural models

3.1. Details of the Approach

For the implementation of the assistance schema, the approach requires of several assets: i) an ADL to specify architectural models, amenable for automation, ii) a framework to organize the design knowledge about architectural styles, iii) strategies for visiting architectural models and generating object-oriented skeletons, iv) a rule-based engine to put the precedent things together.

The starting point has to do with the description of architectural models. These models are represented as typed graph structures, where nodes represent components or connectors and arcs refer to attachments among them. In order to account for semantic properties, the elements in the graph can be annotated with different properties (e.g., protocol information, data types, assigned priorities, logging enabled, etc.). Furthermore, any architectural model is expected to adhere to the rules of a predetermined architectural style (e.g., a pipes-and- filters, blackboard, client-server, etc.). In terms of the graph, such an architectural style will typically define a set of types for components, connectors and properties, along with rules that govern how elements of those types may be composed. Some of the benefits of having stylized architectural models include: support for analysis, reuse and code generation [15]. The importance of these aspects will be visible in subsequent parts of the approach.

The idea is that the architectural model given as input to the approach should be characterized as an “instantiation” of a particular “base” style. Specifically, we have chosen the Acme ADL [1] as the language to specify both styles and general architectures (codified as “families” and “systems” using the Acme vocabulary). To get a flavor of Acme specifications, Figure 4 shows an architectural description of a pipe-and-filter style, basically we have: filter components that receive data and transform data, and pipe connectors that transfer data between filters. Note that the constitutive elements of this “base” style are declared as types (see lines 8, 11, 14, 15, 16, 22, 38, 44 in Figure 4). Additionally, the specification is augmented with predefined properties in order to capture extra-structural information. In our example, we have: concurrency, protocol, customization, error handling and quality (see lines 1-5, 9, 12, 17, 19, 23-30, 32, 35,39, 41, 45, 50 in Figure 4). Then, when it comes to the specification of an input architectural model (to be materialized), the types of the “base” style can be “instantiated” by creating and linking different instances of pipes and filters. Likewise, all the properties defined by the “base” style are inherited by its instantiations. A typical input architecture for the KWIC system [22], whose design is actually based on the pipe-and-filter family provided in Figure 4, is the one introduced previously in Figure 1. Although a graphical representation of this architecture was preferred for clarification purposes, this representation can be certainly translated into an Acme textual form.

```
1.  ->  Property Type Concurrency_T = enum {singleThreaded,multiThreaded};
2.  ->  Property Type Protocol_T = enum {pull,push,pull_push};
3.  ->  Property Type Customization_T = Set{Parameter_T};
4.  ->  Property Type ErrorHandler_T = Record [name : string; comment : string; ];
5.  ->  Property Type QualityMeasure_T = enum {high,low,medium};
6.  ->
7.  ->  Family PipeAndFilterFam = {
8.  ->      Port Type InputPortT = {
9.  ->          Properties { dataType : string; };
10. ->      };
11. ->      Port Type OutputPortT = {
12. ->          Properties { dataType : string; };
13. ->      };
14. ->      Role Type SourceT;
15. ->      Role Type SinkT;
16. ->      Component Type DataSinkT = {
17. ->          Properties { protocol : Protocol_T; };
18. ->          Port input : InputPortT = new InputPortT extended with {
19. ->              Properties { dataType : string; };
20. ->          };
21. ->      }; // End DataSinkT
22. ->      Component Type FilterT = {
23. ->          Properties {
24. ->              protocol : Protocol_T;
25. ->              customizable : Customization_T;
26. ->              sharedData : boolean;
27. ->              logging : boolean;
28. ->              errorHandling : ErrorHandler_T;
29. ->              priority : QualityMeasure_T;
30. ->          };
31. ->          Port input : InputPortT = new InputPortT extended with {
32. ->              Properties { dataType : string; };
33. ->          };
34. ->          Port output : OutputPortT = new OutputPortT extended with {
35. ->              Properties { dataType : string; };
36. ->          };
37. ->      }; // End FilterT
38. ->      Component Type DataSourceT = {
39. ->          Properties { protocol : Protocol_T; };
40. ->          Port output : OutputPortT = new OutputPortT extended with {
41. ->              Properties { dataType : string; };
42. ->          };
43. ->      }; // End DataSourceT
44. ->      Connector Type PipeT = {
45. ->          Properties { dataType : string; };
46. ->          Role src : SourceT = new SourceT;
47. ->          Role snk : SinkT = new SinkT;
48. ->      }; // End PipeT
49. ->
50. ->          Properties { concurrency : Concurrency_T; };
51. ->          Attachments { };
52. ->  }; // End PipeAndFilterFam
```

Figure 4. Fragment of an Acme specification of a pipe-and-filter style, extended with properties

The next step is the incorporation of design knowledge into the engine. As underlying framework for architectural styles, aiming at progressively providing a vocabulary of Acme families and architectural features thereof, we adopted the categorization proposed by [25] extended with quality-attribute information. Besides, we added to this classification a collection of architectural mechanisms (e.g., data conversion, error handling, logging, customization or persistence), as general-purpose concerns that may be plugged into base styles.

On the basis of this categorization, it is possible to analyze well-known architectural styles and recognize the major variants within each style. The style's variants should address those aspects of the architectural level that have to do with the details of how components and connectors can be alternatively arranged. For instance, to continue with our pipe-and-filter style, the available topologies for pipes and filters may admit different pull/push interactions, single- or multi-threaded schemas, and varied data types. In principle, there is a 1-to-N relationship between a style variant and its materializations at the object-oriented level, although some variants evidently direct the developer towards specific implementations (e.g., a configuration of push filters running on separate threads may derive into an implementation based on active objects.).

In order to identify materialization strategies for them, we have considered the following items for each variant: main characteristics of the variant, their contributions to quality-attribute issues, strategies to build an object-oriented implementation, and arguments for and against the available alternatives. Table 1 summarizes the analysis of the pipe-and-filter style. Additionally, Figure 5 shows a possible object-oriented implementation for the KWIC system, based on the *FilterPattern* variant. This implementation was built on top of the pipe-and-filter framelet given in Figure 2.

Table 1. Design knowledge about the pipe-and-filter style

Variant	Main Features	Implementation	Advantages	Disadvantages
Filter Pattern	- Single-threaded - Linear topology - Some communication protocol	Based on an appropriate combination of abstract classes and delegation	- Moderate coupling among components. A given data source/sink does not know about its upstream/downstream components (flexibility) - Transference of data between filters is quite fast (performance)	- The topology is restricted to linear (scalability) - The whole computation schema is fixed (modifiability)
Publisher/ Subscriber	- Single-threaded - Arbitrary topology - Some communication protocol	Based on an event notification mechanism (e.g., Observer pattern)	- Low coupling among components. Each data source potentially has a list of filters monitoring it, and conversely, each filter is observed by either data sinks and/or other filters (flexibility) - Components can be used easily in different contexts (reusability) - Topology may change (scalability)	- Transference among filters is rather slow (performance)
Multi-threaded	- Multi-threaded - Arbitrary topology - Some communication protocol	Based on pipe mechanisms to synchronize active filters (e.g., Producer-consumer pattern)	- Low coupling among filters (flexibility) - Filters that produce/consume data do not need to wait for production/consumption of data, except if the corresponding pipes are blocked (performance) - Pipes are explicitly modeled, thus sophisticated communication protocols can be implemented (communication)	- Some overhead due to synchronization policies (performance)

So, from a more technical point of view, how can we assist developers in moving from an input architectural specification based on a single style (see Figures 4 and 1) to an object implementation (see Figure 5), using pre-compiled knowledge about that style (see Table 1)? As stated in [21], it is possible to define object-oriented architectural styles that exhibit (some of) the characteristics of their corresponding architectural abstractions, using the facilities provided by object-oriented design (e.g., inheritance, information hiding, abstract and hook methods, polymorphism, etc.). Establishing a direct relationship between architectural and object abstractions may be a feasible exercise, although in many situations this mapping could be of little utility, as it may not reflect those design drivers the developer is primarily concerned with. In this context, *the transformation of architectural structures to object-oriented ones depends mainly on what quality attributes are at work rather that on functionality issues*. In fact, this is the very essence of our materialization strategies, they are instruments to characterize related object-oriented design decisions targeted to achieve a desired level of some quality of interest, not just implementation recipes. Then, the effects of materialization strategies on quality attributes can be operationalized, for instance using the NFR framework [27]. At the end, these strategies may or may not be expressed through design patterns, or perhaps admit slightly different implementations.

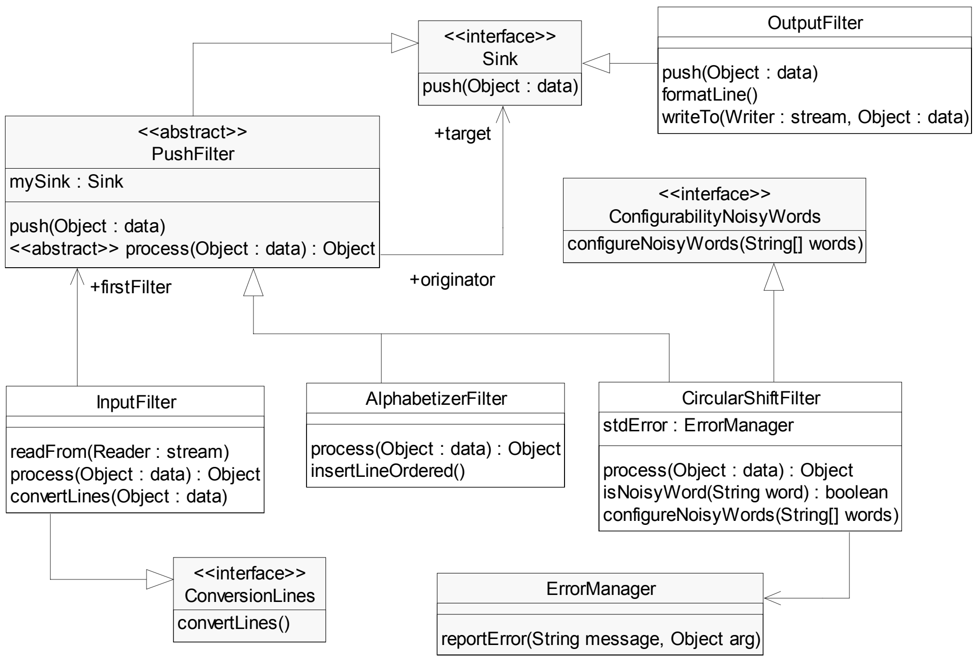
<<interface>>
Sink
push(Object : data)
OutputFilter
push(Object : data)
formatLine()
writeTo(Writer : stream, Object : data)
<<abstract>>
PushFilter
mySink : Sink
push(Object : data)
<<abstract>> process(Object : data) : Object
+target
+originator
+firstFilter
<<interface>>
ConfigurabilityNoisyWords
configureNoisyWords(String[] words)
InputFilter
readFrom(Reader : stream)
process(Object : data) : Object
convertLines(Object : data)
AlphabetizerFilter
process(Object : data) : Object
insertLineOrdered()
CircularShiftFilter
stdError : ErrorManager
process(Object : data) : Object
isNoisyWord(String word) : boolean
configureNoisyWords(String[] words)
<<interface>>
ConversionLines
convertLines()
ErrorManager
reportError(String message, Object arg)

Figure 5. Materialization of KWIC system using a pipe-and-filter framelet (generated by ArchMatE)

Having these concepts in mind, and recalling the support given by stylistic architectural descriptions, we think that a suitable approach to accomplish the materialization is generative programming [11]. Briefly, this approach is focused on mechanisms to enable the automatic production of software from a high-level specification. An advantage of a generative approach is that permits a clear separation between the high-level specification and the implementation model used to support this specification. Normally, there is a code generator that represents the configuration knowledge in a generative model. This code generator is responsible for defining how specific combinations of features in the high-level specification are translated to a set of classes and methods within a particular implementation model. In terms of our

approach, the architectural descriptions act as high-level specification, the corpus of design knowledge provides a set of style-dependent but domain-independent features, the code generator corresponds with our design engine, and finally, the output is represented by a collection of framelets implementing different style variants. More details for the implementation of this kind of generators are described in the next section.
To date, the information we have gathered about a few styles, variants and strategies comes from experimental studies carried out by our research group, plus a some evidences regarding object-oriented implementations of architectural styles reported in [6, 21]. This kind of categorizations intends to be a vehicle to analyze different architectural styles, so that the main relationships among architectural abstractions, quality-attribute issues, and object-oriented implementation techniques can be articulated into a number of materialization strategies. Certainly, the compilation of all this knowledge is frequently a time-consuming task, however, we argue that once provided, it can support a semi-automated engine to explore materializations quite efficiently.

4. ArchMatE: A Rule-based Engine for Materializing Architectural Styles

To validate our ideas about materialization, a prototype Java tool called ArchMatE (ARCHitecture MATerialization Explorer) has been developed. Basically, the tool supports the definition of architectural styles, the instantiation of systems on top of these styles in Acme, and the specification of quality-attribute properties respect to the materialization of these systems. Quality-attribute properties may refer to modifiability, reusability, scalability, or performance aspects of components, connectors or sub-systems. After that, there is a rule-based engine responsible for the generation of object-oriented skeletons, as realization of the system given as input. This engine is actually a code generator implemented using Javalog [3], an integration framework between Prolog and Java. This engine is able to evaluate a corpus of mapping rules and generate different solutions for a particular architectural setting, following predefined materialization strategies. Figure 6 outlines the main classes of the ArchMatE environment.
At first, the architectural description of the input system is edited with AcmeStudio [1]. This description is captured by the *ArchitecturalDesign* class, internally represented using the AcmeLib toolkit [1]. Basically, the standard operation mode of ArchMatE relies on a singleton class *RuleBasedEngine* to work over instances of *ArchitecturalDesign* and produce instances of the *ObjectOrientedDesign* class. In addition, any instance of *ArchitecturalDesign* is required to belong to a predefined Acme family, so that this stylistic information can be used by the *RuleBasedEngine* to carry out the materialization. Regarding quality-attribute preferences, they are also configured by the developer in *ArchitecturalDesign*.
Once a particular architectural style is identified, there are three classes where design knowledge and materialization strategies are hooked, namely: *RuleBasedEngine, ArchitecturalStyleFeatures,* and *ArchitecturalStyleBuilder* (one or more subclasses of each base class per style). Within the *ArchitecturalStyleFeatures* class, we should locate those features derived from our categorization of the style (like the ones for the pipe-and-filter style given in Table 1). Each subclass of *ArchitecturalStyleBuilder,* in turn*,* aims to encapsulate a separate strategy for generating object-oriented models from the actual architectural configuration of components and connectors, based upon the characteristics of its associated *ArchitecturalStyleFeatures*. Here, the class *RuleBasedEngine* is the factory responsible for selecting appropriate subclasses of builders and features for a particular style.

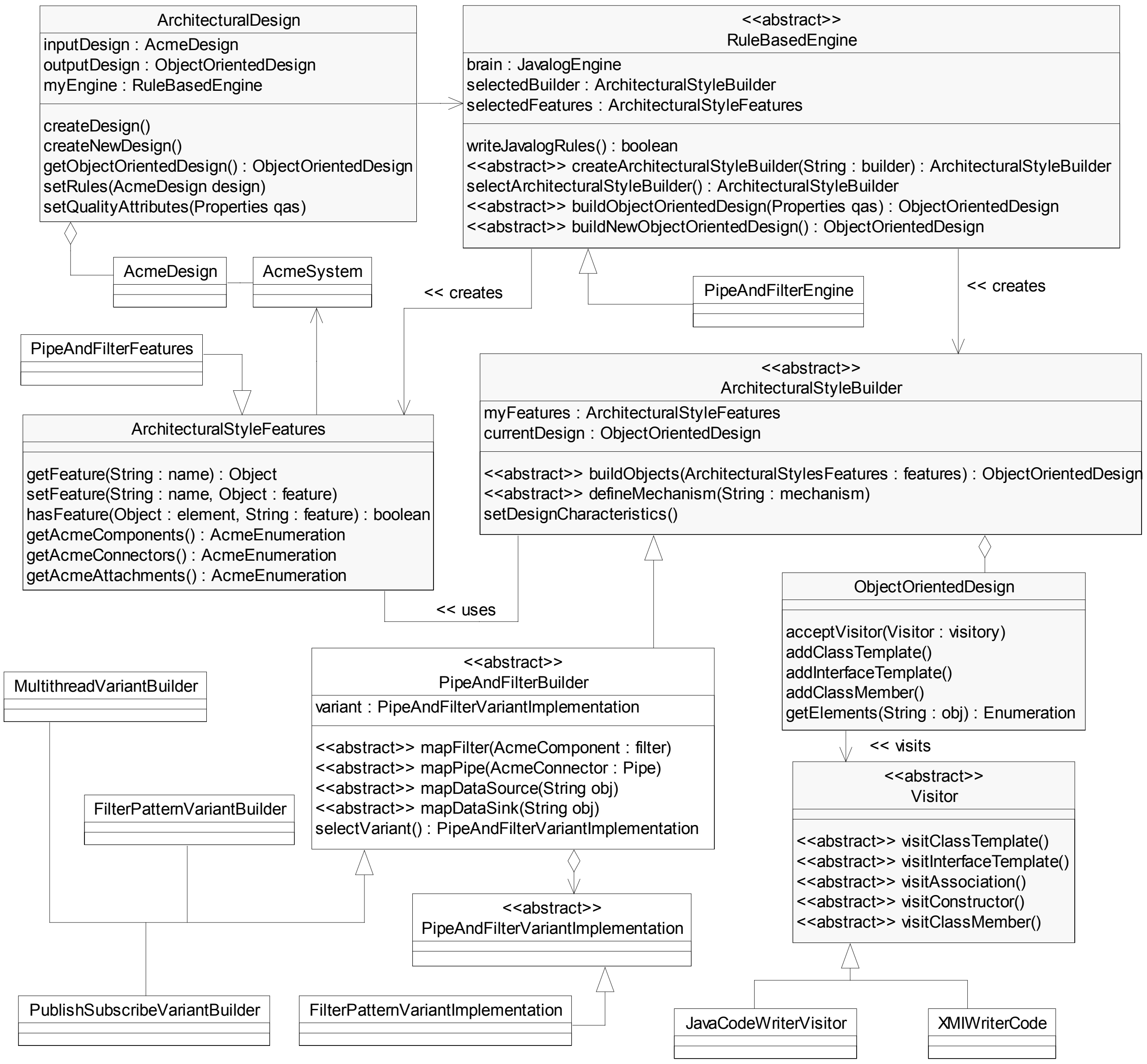
ArchitecturalDesign
inputDesign : AcmeDesign
outputDesign : ObjectOrientedDesign
myEngine : RuleBasedEngine
createDesign()
createNewDesign()
getObjectOrientedDesign() : ObjectOrientedDesign
setRules(AcmeDesign design)
setQualityAttributes(Properties qas)
<<abstract>>
RuleBasedEngine
brain : JavalogEngine
selectedBuilder : ArchitecturalStyleBuilder
selectedFeatures : ArchitecturalStyleFeatures
writeJavalogRules() : boolean
<<abstract>> createArchitecturalStyleBuilder(String : builder) : ArchitecturalStyleBuilder
selectArchitecturalStyleBuilder() : ArchitecturalStyleBuilder
<<abstract>> buildObjectOrientedDesign(Properties qas) : ObjectOrientedDesign
<<abstract>> buildNewObjectOrientedDesign() : ObjectOrientedDesign
AcmeDesign
AcmeSystem
<< creates
PipeAndFilterEngine
<< creates
PipeAndFilterFeatures
<<abstract>>
ArchitecturalStyleBuilder
myFeatures : ArchitecturalStyleFeatures
currentDesign : ObjectOrientedDesign
<<abstract>> buildObjects(ArchitecturalStylesFeatures : features) : ObjectOrientedDesign
<<abstract>> defineMechanism(String : mechanism)
setDesignCharacteristics()
ArchitecturalStyleFeatures
getFeature(String : name) : Object
setFeature(String : name, Object : feature)
hasFeature(Object : element, String : feature) : boolean
getAcmeComponents() : AcmeEnumeration
getAcmeConnectors() : AcmeEnumeration
getAcmeAttachments() : AcmeEnumeration
<< uses
ObjectOrientedDesign
acceptVisitor(Visitor : visitory)
addClassTemplate()
addInterfaceTemplate()
addClassMember()
getElements(String : obj) : Enumeration
MultithreadVariantBuilder
<<abstract>>
PipeAndFilterBuilder
variant : PipeAndFilterVariantImplementation
<<abstract>> mapFilter(AcmeComponent : filter)
<<abstract>> mapPipe(AcmeConnector : Pipe)
<<abstract>> mapDataSource(String obj)
<<abstract>> mapDataSink(String obj)
selectVariant() : PipeAndFilterVariantImplementation
<< visits
<<abstract>>
Visitor
<<abstract>> visitClassTemplate()
<<abstract>> visitInterfaceTemplate()
<<abstract>> visitAssociation()
<<abstract>> visitConstructor()
<<abstract>> visitClassMember()
FilterPatternVariantBuilder
<<abstract>>
PipeAndFilterVariantImplementation
PublishSubscribeVariantBuilder
FilterPatternVariantImplementation
JavaCodeWriterVisitor
XMIWriterCode

Figure 6. Main classes of the ArchMatE environment

In this context, the class *RuleBasedEngine* is at the core of the materialization process, as it is equipped with a Javalog engine able to represent and infer relationships among our design assets. Moreover, this engine has proven to be very good at exploiting the advantages of both object-oriented and logic paradigms. On one hand, the logic paradigm permits the specification of declarative rules for choosing object-oriented variants implemented through subclasses of *ArchitecturalStyleBuilder*, which can be evaluated using the standard Prolog inference mechanisms. On the other hand, after selecting a given variant and its associated builder, the object-oriented paradigm permits to structure well-defined strategies, although a few degrees of customization in their implementation steps are still considered (e.g., through specialization or composition/delegation). This way, the exploratory parts of the materialization process can be written in Prolog, while those parts of the process that are known beforehand or involve significant costs can be programmed in Java. Clearly, those relationships involving quality-attribute issues, architectural configurations and mapping

strategies are candidates to be expressed as logic programs. Conversely, the different implementations of materialization strategies appear more suitable for an object-oriented treatment, ensuring the global characteristics of each variant.
At last, the rule-based engine will select a specific builder, and this builder, after being properly configurated, will generate an object-oriented design. The rules for determining an appropriate builder are specified in a Prolog-like fashion, in terms of feature and structural matching. In the case other design alternatives want to be explored, the developer just asks to the rule-based engine for another solution. This query will cause the selection of another builder or the re-configuration of the current one, to obtain a different materialization, until no more designs satisfying the input are found by the engine. The resulting designs can be later visited to generate Java or XMI code. As additional help, the designs are annotated by the engine with the design rationale followed during the materialization, based on the model given by [8].

```
%% Facts and other helper rules
...
selectMechanisms(...) :- ...
configureMechanism(Builder,Name,Mechanism):- ...

%% Rules for selecting the builder for a given architectural-style variant
selectBuilder(singleThreaded, lineal, [Modifiability, Scalability, Reusability, Performance]) :-
          Modifiability <= 2, Scalability <= 2, Reusability <= 2, Performance <= 3, !,
          selectMechanisms([Modifiability, Reusability, Performance],
                Conversion,Customization,Traceability,singleThreaded,Error,DataSharing),
          send($0, createBuilder, ['FilterPatternVariantBuilder'], MyBuilder),
          configureMechanism(MyBuilder,conversion ,Conversion),
          configureMechanism(MyBuilder,customization ,Customization),
          configureMechanism(MyBuilder,traceability ,Traceability),
          configureMechanism(MyBuilder,error ,Error),
          configureMechanism(MyBuilder,dataSharing ,DataSharing).

selectBuilder(singleThreaded, Topology, [Modifiability, Scalability, Reusability, Performance]) :-
          Modifiability <= 3, Scalability <= 3, Reusability <= 3, Performance <= 1 , !,
          selectMechanisms([Modifiability, Reusability, Performance],
                Conversion,Customization,Traceability,singleThreaded,Error,DataSharing),
          send($0, createBuilder, ['PublishSubscribeVariantBuilder'], Builder),
          configureMechanism(MyBuilder,conversion ,Conversion),
          configureMechanism(MyBuilder,customization ,Customization),
          configureMechanism(MyBuilder,traceability ,Traceability),
          configureMechanism(MyBuilder,error ,Error),
          configureMechanism(MyBuilder,dataSharing ,DataSharing).

selectBuilder(multiThreaded, Topology, [Modifiability, Scalability, Reusability, Performance]) :-
          Modifiability <= 3, Scalability <= 3, Reusability <= 3, Performance <= 3, !,
          selectMechanisms([Modifiability, Reusability, Performance],
                Conversion,Customization,Traceability,multiThreaded,Error,DataSharing),
          synchronizationSharedDataMechanism(Synchronization),
          send($0, createBuilder, ['MultithreadedBuilder'], Builder),
          configureMechanism(MyBuilder,conversion ,Conversion),
          configureMechanism(MyBuilder,customization ,Customization),
          configureMechanism(MyBuilder,traceability ,Traceability),
          configureMechanism(MyBuilder,error ,Error),
          configureMechanism(MyBuilder,dataSharing ,DataSharing),
          configureMechanism(MyBuilder,synchronization, Synchronization).

%% If concurrency is not specified
selectBuilder(Topology, QualityLevels) :- selectBuilder(Concurrency, Topology, QualityLevels).

%% Typical queries - (QualityLevels -> high = 1 medium = 2 low = 3 ignore = 0)
?- selectBuilder(multiThreaded, lineal, [1, 1, 1, 3]).
?- selectBuilder(lineal, [1, 1, 1, 3]).
```

Figure 7. Sample of Javalog rules for determining materialization strategies

To better understand the relationships among builders, features and the Javalog engine, let's revisit the pipe-and-filter family of Figure 4 and the corresponding instantiation of the KWIC system presented in Figure 5. Figure 7 sketches how the logic module containing rules for selecting builders within this style would look like. The quality-attribute levels can have values in a low/medium/high/ignore scale (see lines 7, 8, 19, 19, 29, 30 in Figure 7). The *send()* predicate (see lines 11, 22, 34 in Figure 7) is a built-in predicate to invoke methods on Java objects. Let's observe also that each builder can be configured with style-independent mechanisms such as data conversion, error handling, or traceability, among others, depending on the variant selected (see lines 12-16, 23-27, 33, 35-40 in Figure 7). According to the proposed rules, three main builders are available for the engine, namely: *FilterPattern VariantBuilder*, *PublishSubscribe VariantBuilder*, and *MultithreadedVariant Builder*. Although omitted in the figure for clarity reasons, the builders typically rely on several concrete subclases to arrive to an object-oriented design.

4.1. Experimental Results

As proof-of-concept, we have implemented two applications: a KWIC and a Tic-Tac-Toe systems, using two architectural styles: pipes-and-filters and blackboard respectively. Within each style, several variants and object-oriented implementations of these variants were analyzed. This knowledge has been incorporated into ArchMatE, and a number of object-oriented alternatives for the case-studies have been accordingly generated and evaluated.
Although the results gathered from these experiments are quite preliminary, the comparisons of the generated materializations with the solutions reported for the two system in the literature have been encouraging in terms of similarity. To this end, a set of modifiability-related metrics (e.g., non-commented source statements, cyclomatic complexity, stability, abstractness and rippling factor) [16] have been computed on the solutions for the KWIC and Tic-Tac-Toe systems. Interestingly, the evolution of the quality-attribute properties under consideration, as different solutions are being generated by the engine, seems to agree with the developer's expectations for each system. In spite of these facts, a more objective evaluation of the approach to determine how well the materializations correspond to reality is still required. Such an evaluation would involve a more extensive assessment of the materialization strategies, and a larger number of systems and their solutions. This is currently a subject of outgoing research within the ArchMate project.

5. Related Work

Software architectures and frameworks have provided much inspiration for this work. Besides our approach based on proto-frameworks [10, 12], other lines of research have dealt with the transition between architectural and object models.
Two of the most relevant approaches are the C2 model [19] and the ArchJava language [2]. In [19], the authors have developed a family of implementation frameworks for architectural models based on the C2 style. The approach provides a basic object-oriented framework representing the main concepts of this style. Alternative implementations of the same framework have been also derived to address some extra-functional properties required at the application level. The work of [2], on the contrary, presents an extension to Java that seamlessly unifies software architecture with implementation. In terms of materialization, both C2 and ArchJava define particular realization techniques, although the exploration of solutions is not explicitly considered. As the C2 model is more focused on architectural issues concerning graphical interfaces, the mapping rules appear strongly influenced by this context. ArchJava somehow ensures that an implementation conforms to several constraints prescribed by the architecture, although this implementation is not really a derived product of the architectural model. Architectures expressed in ArchJava are usually more concrete than

architectures in other ADLs, and this tends to restrict the ways in which a given architecture can be implemented. Furthermore, little is said in the two approaches about the influence of quality factors in the materialization process.
On the other hand, the use of object-oriented frameworks to represent architectural building blocks can be traced to the notion of framelets [23]. A framelet is a small white-box framework, comprising a set of logically related components and design patterns and interfaces. In ArchMatE, the object-oriented structures resulting from the materialization of each style can be seen actually as framelets, because they capture a cluster of related architectural requirements by means of framework mechanisms. Other object-oriented perspectives of this issue have been also explored in [7, 31].
Besides, our approach bears similarities with domain-specific software architectures (DSSA) [29], and with model-driven architectures (MDA) [26]. A DSSA provides, basically, a software architecture with reference requirements and a domain model, an infrastructure to support it, and a process to instantiate and refine this infrastructure. The main difference with the materialization approach is that we aim to capture architectural abstractions mostly domain-independent and their relationships with quality attributes. In the case of MDA, the work is concerned with the definition of open standards and supporting tools for system modeling and transformation (often using UML profiles), rather that with architectural issues, which seem to be implicit into these models.
More recently, there has been a proposal to generate adaptable software architectures for embedded systems, using the NFR Framework in combination with a base of design knowledge [27]. With this purpose, quality attributes are seen as potentially synergistic or conflicting goals to be achieved during the process of software development. Then, goal graphs with different kind of links are constructed, and design alternatives are linked to graphs according to their partial contributions for or against certain goals. An evaluation procedure calculates the effect of each design decision on the graph. After that, by means of a tool equipped with correlation rules, the developer can select different alternatives from the knowledge base (in this case, regarding adaptability). In our opinion, goal graphs result very useful as modeling instruments, although the codification of knowledge and experience relating tradeoffs with design decisions using methods and correlation rules still needs more elaboration. Besides, the distinction between architectural and object alternatives is not always made clear in the approach.

6. Conclusions and Future Work

In this paper, we have described a tool approach to explore alternative ways of transforming architectural models into object-oriented structures. The contributions of the proposed approach are twofold: i) it considers those quality-attribute settings more relevant to the developer as drivers of the process, ii) it lessens the cognitive complexity of the process (especially in the case of novice or inexperienced developers). Essentially, the mapping process is accomplished by means of materialization strategies, which help the derived object structures retain to some extent the characteristics prescribed by the original architectural model. Furthermore, as the developer is supported with design assistance, he/she is less likely to overlook the various options, variants and details associated with the object-oriented materialization.
Consequently, ArchMatE has been developed as a prototypical design engine to automate the exploration of materialization alternatives. In order to incorporate design knowledge into this engine, it was implemented following a multi-paradigm schema in Javalog. Using Javalog, it

is possible to take advantage of both logical inference mechanisms and conventional object-oriented mechanisms when capturing different kinds of materialization strategies. Given this flexibility, the exploratory parts of the materialization, such as relationships between quality attributes, architectural-style variants and mapping strategies, were expressed in Prolog, while those parts known beforehand, such as implementation of mapping strategies, were expressed in Java.

Currently, ArchMatE has been tested with two architectural styles (pipes-and-filters and blackboard), a few case-studies, and a small sample of quality-attribute concerns. The results of this evaluation have certainly demonstrated the potentialities of the approach, however, some problems and open issues still remain. The composition of partial solutions, either at the architectural or at the object levels, is still little understood, so the engine has to deal with it in an "ad-hoc" manner. Second, this shortcoming also affects the engine's control over the quality-attribute tradeoffs of different generated solutions. Third, the developer cannot make yet decisions in the middle of the generation process, loosing opportunities to incorporate his judgement as the tool is running, in order to early reject inferior solutions in favor of the refinement of more promising alternatives.

Despite the investigation of techniques to cope with the above problems, other interesting lines of work include: the analysis of more architectural styles and associated materialization strategies, the definition of other implementation techniques for each strategy (e.g., AOP [17]), and the topic of code generators. Finally, a long-term goal of this research is the improvement of the assistive capabilities of ArchMatE, and its further integration within the proto-framework design approach.

References

1. Acme Homepage http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html
2. Architectural Reasoning in ArchJava. J. Aldrich, C. Chambers, and D. Notkin. Proceedings of the European Conference on Object-Oriented Programming, June 2002.
3. JavaLog: A framework-based integration of Java and Prolog for agent-oriented programming. A. Amandi, M. Campo, A. Zunino . Computer Languages, Systems and Structures. Elsevier Science. ISSN: 0096-0551. Ed.: R. S. Ledley. 2004.
4. Software Architecture in Practice. L. Bass, P. Clement, and R. Kazman. 2nd Edition. Published by Addison-Wesley. 2003.
5. Identifying Quality-Requirements Conflicts. B. Boehm and H. In. IEEE Software, March, 1996.
6. Software Architecture Design: Evaluation and Transformation. J. Bosch and P Molin. Proc. 1999 IEEE Engineering of Computer Based Systems Symp. (ECBS99), 1999.
7. Odyssey: A Reuse Environment based on Domain Models. R. Braga, C. Werner, and M. Mattoso. Proc. IEEE Symposium on Application-Specific Systems and Software Eng. Technology (ASSET'99), IEEE CS Press, Texas, March 24-27, pp. 50-57, 1999.
8. Reasoning with Design Rationale. J. Burge. D. Brown. In Artificial Intelligence in Design'00, (Ed.) J. S. Gero, Kluwer, Dordrecht. 2000.
9. Pattern-Oriented Software Architecture. A System of Patterns. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal John Wiley & Sons. 1996.
10. Developing Object-oriented Enterprise Quality Frameworks using Proto-frameworks. M. Campo, A. Díaz Pace, M. Zito. Software: Practice and Experience,Vol 32 No. 8, Pp. 837-843. Wiley. 2002.
11. Generative Programming: Methods, Techniques, and Applications. K. Czarnecki. ICSR 2002: 351-352

12. Architecting the Design of Multi-Agent Organizations with Proto-frameworks. A. Díaz Pace, M. Campo, A. Soria. Software Engineering for Multi-Agent Systems II - LNCS 2940. Springer, 2004.
13. Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. Fayad, D. Schmidt, R. Johnson.Wiley Eds., 1999.
14. Design Patterns, Elements of Reusable Object-Oriented Software. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Addison-Wesley. Massachussetts, 1994.
15. ACME: Architectural Description of Component-based Systems. D. Garlan, R. Monroe, D. Wile. Foundations of Component-based Systems. Cambridge Univ. Press, 2000.
16. Structural Analysis for Java. http://www.alphaworks.ibm.com/tech/sa4j
17. Aspect-Oriented Programming. G. Kiczales, J. Lamping, J. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, J. Irwin. Proc. of the European Conference on Object-Oriented Programming (ECOOP), Finlad. Springer-Verlag LNCS 1241. June 1997.
18. A Classification and Comparison Framework for Software Architecture Description Languages. N. Medvidovic, R. Taylor. IEEE Trans. on Soft. Eng., vol.26 no.1, 2000.
19. A Family of Software Architecture Implementation Frameworks. N. Medvidovic, N. Mehta, M. Mikic-Rakic. Proceedings 3rd IFIP WICSA, 2002.
20. Capturing Design Expertise in Customized Software Architecture Design Environments. R. Monroe. Proc. 2nd Int. Software Architecture Workshop, 1996.
21. Architectural Styles, Design Patterns, and Objects. R. Monroe, A. Kompanek, R. Melton, D. Garlan. IEEE Software 14(1): 43-52, 1997.
22. On the Criteria To Be Used in Decomposing Systems into Modules. D. Parnas. Commununications of the ACM 15(12): 1053-1058, 1972.
23. Framelets - Small is Beautiful. W. Pree, K. Koskimies. In: Building Application Frameworks (M.E. Fayad, D.C. Schmidt, R.E. Johnson, ed.), Wiley 1999, 411-414.
24. Software Architecture, Perspectives on an Emerging Discipline. M. Shaw and D. Garlan. Published by Prentice-Hall. 1996.
25. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. M. Shaw and P. Clements. Proc. COMPSAC97, 1997.
26. [Soley00] Model Driven Architecture. R. Soley.OMG. White paper Draft 3.2. 2000
27. Semi-Automatic Generation of Adaptable Architectures. N. Subramanian, L.Chung. Software Engineering Research and Practice 2003: 149-154. 2003
28. Synthesis-Based Software Architecture Design. B. Tekinerdogan and M. Aksit. In Software Architectures and Component Technology: The State of the Art in Research and Practice, M. Aksit (Ed.), Kluwer Academic Publishers, pp. 143 - 173, 2001.
29. Domain Analysis, Domain Modeling, and Domain-Specific Software Architectures. W. Tracz. Proc. of the 4th Int. Conf. on Software Reuse, pp 232-233. Orlando, FL. 1996.
30. UML Homepage http://www.omg.org/technology/uml/
31. Uma Abordagem para a Seleção de Padrões Arquiteturais Baseada em Características de Qualidade. J. Xavier, C. Werner and G. Travassos. Anais XVI Simp. Brasileiro de Engenharia de Software. Porto Alegre: Editora Evangraf Ltda, 2002. v.1. p.52 - 67