IJISCSE
• Vol. 1 • No. 1 • January-June 2010, P. 1-35

From Requirements to Architectural Style Focusing on Connector Properties

Danielle L. DeBraccio[1] & Harry S. Delugach[2]

Abstract

Even though the quality of software requirements and software architecture have a significant impact on the success and quality of a software project, few tools exist to support the transition from requirements to architecture. Many desired qualities in a software project that are expressed in the software requirements can be achieved with software architectural styles. This paper proposes an approach towards an automated approach that can help choose an appropriate architectural style given a set of natural language software requirements. The approach, Automated Architecture Scoring Method (AASM), transforms software requirements into a formal model in conceptual graphs which is analyzed for possible software architectural components, and possible properties of those components. AASM then analyzes those properties and develops a recommendation for an architectural style. This paper focuses on using conceptual graphs to guide architectural style selection, with a particular emphasis on software architectural connector properties.

1. INTRODUCTION

When choosing the best software architecture for a project, a software architect is influenced by many factors including time constraints, the experience of his team, and his own biases towards (or against) certain software architectures. A tool that provides an unbiased software architecture recommendation based solely on the given requirements can serve as a valuable starting point. Tools exist for automated requirements analysis and development environments exist for constructing a software architecture, but few automated approaches exist for choosing between software architectural styles given a set of software requirements [Galster 2006]. The ultimate goal of our research is to develop automated tools to select an architectural style from a given set of natural language software

[1]A-P-T Research, Inc., 4950 Research Dr. NW Huntsville, AL 35805
[2]Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899
Email: [1]ddebraccio@apt-research.com, [2]delugach@cs.uah.edu

requirements. This paper will describe an approach towards that goal with the focus on connector properties to help in choosing an architectural style.

1.1. Software Architecture in the Software Development Process

Many definitions of software architecture exist, although there is no consensus on one single definition ([Kruchten 2006], [Clements 2003]). Bass, Clements and Kazman propose the following general definition of software architecture: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [Bass 2003].

Perry's structural perspective on software architecture was chosen for this paper because the research that most strongly contributes to this paper's focus, [Grüenbacher 2004] and [Bhattacharya 2007], build their work on this perspective: "software architecture is a set of architectural … elements that have a particular form. We distinguish three different classes of architectural elements: processing elements; data elements; and connecting elements" [Perry 1992]. Several other perspectives of software architecture propel the definition beyond the structural perspective: framework models, dynamic models, and process models [Clements 2003]. This paper will explore software architecture from the process model perspective because it "focus[es] on construction of the architecture and the steps or process involved in that construction. From this perspective, architecture is the result of following a process script" [Clements 2003].

A semantic gap exists between requirements and software architecture [Grüenbacher 2004]. Software requirements are typically gathered into natural language and diagrams [Cyre 1997], even though more formal notations have been developed [Grüenbacher 2004]. Software architecture needs to be described in a formal approach [Pressman 2005] that allows the architect to decompose and construct an architecture with components and connectors. Many Architectural Description Languages (ADLs) have been developed for this purpose, and the important ones include Rapide, UniCon, Aesop, Wright, Acme, and UML [Land 2002]. Approaches that attempt to bridge the semantic gap between requirements and architecture are discussed further in sections *Error! Reference source not found*. and 2.2.

1.2. Selecting Architectural Styles

Software architectural styles are important because they enforce quality attributes (e.g., modifiability, usability, scalability, performance, etc.) for a system [Bhattacharya 2007]. "A style guides the architectural design of a system, with the promise of desirable system qualities" [Grüenbacher

2004]. Styles facilitate "high-level reuse and bring economy to the design of software architectures" [Monroe 1996]. "An architectural style is a key design idiom that implicitly captures a large number of design decisions, the rationale behind them, effective compositions of architectural elements, and system qualities that will likely result from the style's use" [Medvidovic 2003].

The challenge of automating architectural style selection is that "the rules guiding the selection and application of a style … are typically semi-formal at best, requiring significant human involvement. Furthermore, multiple architectural styles may appear to be (reasonably) well suited to the problem at hand, requiring additional work to select the most appropriate style" [Grüenbacher 2004]. A way to address this problem is to acknowledge that architectural styles are made of a common set of architectural building blocks [Shaw 1997]. As summarized in [Bhattacharya 2007], Shaw and Clements show [Shaw 1997] that different styles can be compared by analyzing the following common parts:

- Constituent Parts: the components and connectors;
- Control Issues: the flow of control among components (e.g., synchronicity);
- Data Issues: details on how data is processed (e.g., data continuity);
- Control/Data Interaction: the relation between control and data;
- Type of Reasoning: analysis techniques applicable to the style.

A software architecture is comprised of components and connectors [Perry 1992], [Grüenbacher 2004]. The components can be computational components or data components [Perry 1992]. As expressed in [Shaw 1996], "the interactions among components are captured within explicit software connectors (or buses)" [Grüenbacher 2004], and architectural styles "both capture and reflect the key desired properties of the system under construction (e.g., reliability, performance, cost)" [Grüenbacher 2004].

Choosing the most suitable software architectural style(s) can prevent problems later in the software process. Finding a way to automate the style selection should provide a valuable tool to software architects. The approach proposed in this paper is called the Automated Architectural Scoring Method (AASM). AASM builds on existing approaches that transform requirements to architectural building blocks, and also on approaches that analyze these building blocks to determine an architectural style. The approach proposed in this paper includes transforming natural language requirements to an intermediate model; however, the focus of AASM will be detecting and analyzing the architectural building blocks from that intermediate model with an emphasis on architectural connectors, and predicting an architectural style from the results. Section 2 will describe the background research, section 3 will describe the AASM approach, section 4 will present the AASM results of two projects, and section 5 analyzes those results.

2. RELATED WORK

The research goal for this paper is to identify approaches (on which to base automated tools) that support the transformation of requirements to a software architectural style. Previous work includes requirements engineering (Section 2.1), the transformation of requirements to software architecture (Sections *Error! Reference source not found.* and 2.2), and conceptual graphs for modeling requirements and architecture (Sec. 2.4).

2.1. Requirements Engineering

Although this paper does not explore the requirements engineering part of AASM in depth, it builds on two areas from the requirements engineering field: natural language processing (Section 2.1.1) and conceptual graphs, a notation that facilitates natural language processing (Sections 0 and 2.1.2).

2.1.1. Natural Language Processing

Many tools have been developed for requirements engineering, "in particular, requirements management tools (e.g., Rational RequisitePro and Telelogic DOORS) and specification analysis tools" [Ambriola 2003]. The requirements engineering research for this paper is influenced mostly by CIRCE [Ambriola 2003], a specification analysis tool. CIRCE analyzes requirements through natural language processing (NLP). Natural language processing is the "computer analysis and generation of natural language text … In the computer analysis of natural language, the initial task is to translate from a natural language utterance, usually in context, into a formal specification that the system can process further" [McGraw-Hill 2007].

It is important to automate the modeling of natural language software requirements because "79% of requirements documents are couched in unrestricted [natural language]" [Gervasi 2000]. The primary difficulty in transforming natural language requirements into a form that can be manipulated by a machine is the inherent ambiguity and imprecision of natural language.

Conceptual graphs (CGs) are a logic-based graphical notation. CGs complement the science of natural language processing because they can map directly to natural language. “Since conceptual graphs were originally designed as a semantic representation for natural language, they can help form a bridge between computer languages and the natural languages that everyone reads, writes, and speaks” [Sowa 2000]. The remainder of this section provides a very brief and informal introduction to the portion of conceptual graph theory that relates to this paper.

CGs were first introduced by Sowa in [Sowa 1984]. Sowa derived CGs from a syntax specifically developed to express logic in graphical notation.1 CGs provide a graphical way to express first-order logic statements. First-order logic is a system of rules that can (1) express statements of existence (or non-existence), (2) express a relation between statements, and (3) apply a relation to a family of statements. A conceptual graph is a directed graph with two types of symbols: concepts (shown as rectangles) and conceptual relations (shown as ovals or diamonds) [Sowa 2000].

The appearance of a concept in a conceptual graph is an existential assertion that the concept exists. To express that a concept does *not* exist, the concept is surrounded by a rectangle with rounded edges (“negated context”). For example, Figure 1(a) expresses the assertion that a Cat exists, while Figure 1(b) asserts that a Dragon does not exist. Relations are shown as ovals, connected by directed arcs whose direction is defined for the semantics of each relation. Figure 1(c) shows a conceptual graph representation of the sentence: “Elsie the cat sits on a mat.” In Figure 1(c), the concept type Cat is shown with a referent to a specific individual named Elsie.

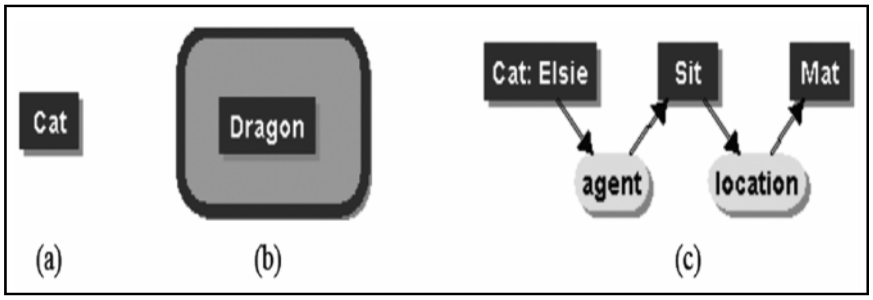


Figure. 1: A CG Concept, Negated Concept and Sample Graph

Elsie is the **agent** of a **Sit** action. **Mat** is the **location** of the **Sit** action.

To add meaning to the conceptual graph, the concepts are organized into a type hierarchy: a lattice that shows which concept types inherit from other concept types. Figure 2 shows a possible type hierarchy for the conceptual graph in Figure 1(c).

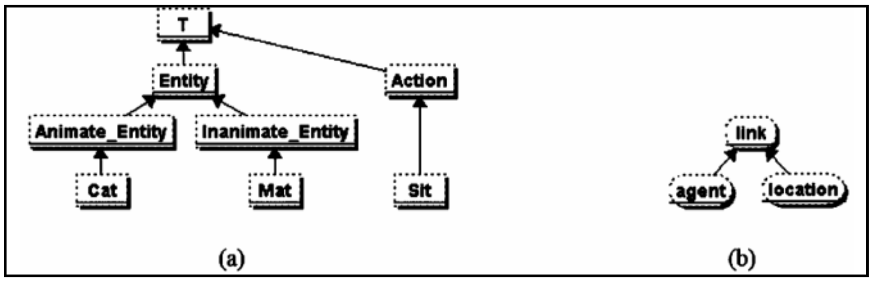


Figure 2: Example of a CG type Hierarchy and CG Relation Hierarchy

In Figure 2, Cat is modeled to be a type of Animate_Entity, which in turn is a type of Entity. The T type is the universal type from which all concept types inherit [Sowa 2000].

In a similar way, the CG relations are categorized into a relation hierarchy. Figure 2 shows a possible relation hierarchy for the conceptual graph in Figure 1(c). It is common practice in relation hierarchies to assign **link** as the "universal relation type" from which all relations inherit.

The way to impose restrictions on how conceptual graphs are formed is to create CG definitions. CG definitions are conceptual graphs that serve as templates for the way the concepts and relations may be connected in a particular model. The CG definition applies the relation(s) depicted in the graph to all concepts of the designated concept type. The designated concept type can be shown with a @forall symbol, as in Figure 3, which asserts that all instances of type Cat are Sitting on a Mat.

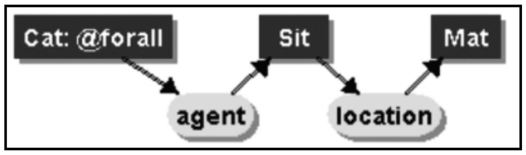


Figure 3: Example of a CG Definition Graph

2.1.2. Requirements and Conceptual Graphs

Research into the transformation of software requirements to conceptual graphs includes work by Ryan, Delugach, and Cyre. In [Ryan 1993], Ryan proposed a requirements collecting tool that aided the transition of requirements from natural language to a formal notation. In [Delugach 1991], [Delugach 1992] and [Delugach 1996], Delugach transformed requirements to conceptual graphs to analyze multiple views of requirements. Conceptual graphs provide "a single knowledge representation capable of capturing the information expressible in several existing requirements notations" [Delugach 1992].

In [Cyre 1997], Cyre explored automating the generation of CGs from natural language requirements. Cyre's goal was to "automatically translate, evaluate, and interpret requirements expressed in natural language … so that they may be integrated and analyzed automatically" [Cyre 1997]. In later research, Glaze transformed requirements conceptual graphs to software architecture in [Glaze 2004], which will be described in Section 2.2.

2.2. Approaches to Predicting Architectural Styles

Before discussing how to predict architectural style from requirements, it is important to mention one approach to transforming requirements directly into an architecture. The ArchE project [Bachmann 2003] is one example of continuing research about constructing a software architecture via the analysis of quality attributes. However, the research that lends itself best to automatic architectural style prediction maps architectural properties to architectural styles.

Approaches that predict architectural styles from specifications include the Concept Scoring Method (CSM) [Glaze 2004], an approach by Bhattacharya and Perry [Bhattacharya 2007], and CBSP (Component-Bus-System-Property) [Grüenbacher 2004]. For CSM, Glaze devised an automated approach to select an architectural style from software requirements. He modeled requirements with conceptual graphs and measured characteristics about the graphs to determine an architectural style; he then used the conceptual graph model and the determined style to manually generate the preliminary software design. Glaze searched for characteristics in the conceptual graphs that would suit the Structured Analysis and Design Technique (SADT) style and the Object-Oriented (OO) style. He then used the resulting metrics to predict the more suitable style.

The approach developed by Bhattacharya and Perry [Bhattacharya 2007] searches *architectural* specifications (not *requirements* specifications) for characteristics of the architectural building blocks described in [Shaw 1997] to predict an architectural style. Their assumption is that a repository of reusable software components exists, and that the component specifications are organized according to their proposed documentation model. For each architectural building block, they search a list of the relevant specifications for the characteristics.

Bhattacharya and Perry then compare the results with a table derived from [Shaw 1997] that maps the architectural building block characteristics to the architectural styles. Table 1 shows just a few of the styles from the table Bhattacharya and Perry use.

Table 1
Comparison of Architectural Styles (Bhattacharya & Perry)

		Style		
		Pipes and Filter	Call based Client Server	Layered –
Constituent Parts	Components	Transducers	Programs	–
	Connectors	Data Stream	Calls or RPC	–
Control Issues	Topology	Linear	Star	Hierarchical
	Synchronicity	Asynchronous	Synchronous	Any
Data Issues	Topology	Linear	Star	Hierarchical
	Continuity	Continuous	Sporadic	Sporadic
Control/ Data Interaction	Isomorphic Shapes	Yes	Yes	Often

CBSP is an approach that uses architectural building blocks similar to those described in Section 1.2 to transform requirements to architectural styles. CBSP, developed by Grünbacher, Egyed, and Medvidovic, is "a lightweight approach intended to provide a systematic way of reconciling requirements and architectures using intermediate models" [Grüenbacher 2004]. In each step, the software architects vote on the topic(s) addressed. The CBSP approach, a flexible five step process, allows iterations between steps. The steps are as follows (the iterations are not shown):

Step 1 – Selection of requirements for next iteration

Step 2 – Architectural classification of requirements

Step 3 – Identification and resolution of classification mismatches

Step 4 – Architectural refinement of requirements

Step 5 – Trade-off choices of architectural elements and styles

In Step 1, the software architects distinguish which requirements are architecturally relevant based on prioritizations set by the stakeholders. In Step 2, the software architects rate each requirement for its suitability for being transformed into each architectural building block and property. In this step, the CBSP architectural building blocks are Data Component, Process Component, Bus/Connector, and (sub)System. The CBSP properties are Data Component Property, Process Component Property, Bus/Connector Property, and (sub)System Property. In Step 3, the architects vote again to exclude requirements that are irrelevant to the architecture. In Step 4, the architects refine the intermediate model by merging and creating new CBSP elements. In Step 5, the properties of the resulting CBSP elements are compared with a table that rates the support provided by each architectural style. The properties in the table are meant to be added as needed by the architects for the particular project. Table 2 is an abbreviated version of the CBSP table from [Grüenbacher 2004].

Table 2
Comparison of Architectural Styles (CBSP)

		Style		
		Pipes and Filter	Client Server	Layered
Data Component	*aggregated*	—	++	+
	persistent	o	++	o
Processing Component	service provide/ consume only	o	++	o
	loose coupling	++	+	—
Connector/ bus	*synchronous*	—	++	++
	asynchronous	++	—	—
	local	+	—	++
	distributed	+	++	—
(sub)System	*efficient*	—	o	o
	scalable	+	+	—

Legend: ++ *extensive support* + *some support* o *neutral* — *no support*

The approach in this paper can partially automate the CBSP approach. The CBSP approach was chosen mainly because it supports the transformation of software requirements to software architectural styles. Also, CBSP supports creating a design that may include multiple software architectural styles, a flexibility that software engineers often need in large projects. The reason CBSP has this flexibility is because it transforms requirements to architectural components; the properties of the architectural components can then be analyzed by architectural style.

AASM adds automation to the CBSP approach, but we do not believe than any approach should bypass or eliminate the architect's experience and expertise. That is why the AASM is presented in Section 3 within the context of CBSP because it ensures the involvement of the software architect.

3. THE AASM APPROACH

To select a software architectural style from software requirements, the approach should be an iterative process ([Grüenbacher 2004], [Nuseibeh

2001]) that analyzes both functional and quality requirements. AASM does not yet handle quality requirements, but the approach provides the groundwork for future work as discussed in [DeBraccio 2007]. Architectural refinement is an iterative process because "designing architectures is necessarily an iterative activity … it is impossible to get it completely right the first time" [Bengtsson 1999]. Software developers have found that "the architecture design evolves with the detailed design and often new requirements, requiring architectural changes, become apparent when working on detailed design" [Gurp 2002].

AASM will focus on the partial automation of a small part of a larger iterative process that helps software developers to select an architectural style (or an architecture of multiple styles) from software requirements. AASM can be applied in the context of the CBSP approach. AASM can be described in the following steps:

(a) Model the software requirements into an intermediate formal model (in conceptual graphs).

(b) Find architectural constraints.

(c) Find candidates for the CBSP architectural building blocks: Data components, Process components, Connectors, and (sub)Systems.

(d) Find candidate properties of CBSP architectural building blocks.

(e) Determine the Architectural Style Prediction Scores.

AASM fits into the CBSP approach as shown in Table 3.

Table 3
Steps of AASM within the CBSP Context

Origin	*Step*	*Main Supporting References*
AASM Step A	Model requirements into conceptual graphs	[Ambriola 2003]
AASM Step B	Find architectural constraints	[Svetinovic 2003], [Eden 2004]
AASM Step C	Find candidate architectural building blocks	[Grüenbacher 2004], [Glaze 2004]
CBSP Step 1	Selection of requirements for next iteration	▨
AASM Step D	Find candidate architectural properties	[Grüenbacher 2004], [Bhattacharya 2007]
CBSP Step 2	Architectural classification of requirements	▨

Table 3 Contd...

Table 3 Contd...

CBSP Step 3	Identification and resolution of classification mismatches	▨
CBSP Step 4	Architectural refinement of requirements	▨
AASM Step E	Determine the Architectural Style Prediction Scores	[Grüenbacher 2004], [Bhattacharya 2007]
CBSP Step 5	Trade-off choices of architectural elements and styles	▨

Automating Step A requires a linguistic CASE tool such as CIRCE [Ambriola 2003]. The work from [Svetinovic 2003], and the hypothesis for a formal approach to categorizing design statements of different abstraction levels from [Eden 2004] could support a partial automation of Step B. Step C can be an automated way to search the requirements conceptual graphs from CBSP Step 1 for candidate architectural building blocks. AASM metrics and metrics from Bhattacharya and Perry's approach [Bhattacharya 2007] support the automation of finding some candidate properties of the architectural components for Step D. Step E depends on the style comparison tables from [Grüenbacher 2004] and [Bhattacharya 2007] described in Section 2.2; automating Step E mainly involves calculations of scores.

3.1. Step A: Model the Software Requirements in CGs

The requirements were modeled in CGs because 1) CGs are not biased for or against any particular architectural style, and 2) CGs allow for modeling requirements as a close approximation of natural language requirements, and potential exists for the automation of modeling natural language requirements ([Cyre 1997], [Lenat 1995], [Ambriola 2003]).

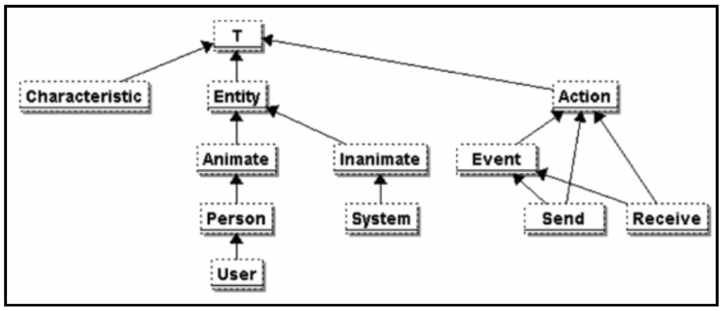


Figure 4: AASM Type Hierarchy

The relation hierarchy for AASM (shown in Appendix A of [DeBraccio 2007]) is a small set of relations that are commonly used in conceptual graph modeling such as **attr** (attribute), **agt** (agent), and **obj** (object). The AASM type hierarchy as shown in Figure 4 was kept minimal to simulate the lightweight ontology in CIRCE [Ambriola 2003].

The three basic types in the AASM type hierarchy are the **Entity** type, the **Action** type, and the **Characteristic** type. Similarly to CIRCE, AASM includes a **System** type; in AASM, this type inherits from the **Entity** type. AASM also includes some of CIRCE's basic actions such as the **Send** and **Receive** types that inherit from the **Action** type. CIRCE lets the user distinguish between a condition, an event, and an action. In AASM, an **Event** is a subtype of the **Action** type so that the **Event** types are identified as Process component candidates (see [DeBraccio 2007]). A type for conditions was not included in the AASM type hierarchy because conditions can be modeled as **Event** types in sequential order (see Section 0).

The CG definition in Figure 5 shows a portion of the AASM Action Type CG Definition from Appendix B of [DeBraccio 2007]. Figure 5 shows the basic schema for how the **Action** type concepts were modeled in this paper.

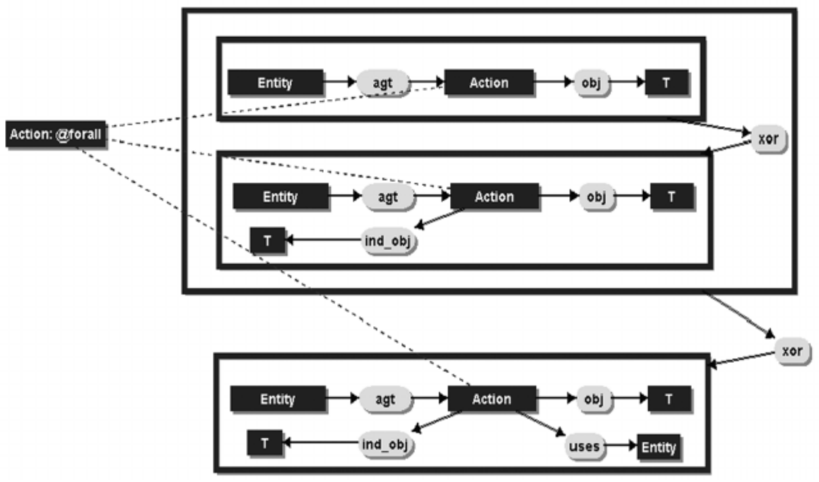


Figure 5: Part of the AASM Action Type CG Definition

In AASM, the set of requirements that are the output of Step A will be assumed to be as unambiguous as a tool such as CIRCE could make them. It is further assumed that the **Entity**, **Action**, and **Characteristic** types are already defined in the type hierarchy.

3.2. Identifying the Architectural Requirements

AASM identifies architectural requirements in both Steps B and C. Although we assume that Step A significantly reduces ambiguity in requirements, it is not absolutely necessary that all ambiguity be eliminated, since only some of the requirements are involved in identifying the architecture [Kozaczynski 2002].

Architectural requirements are the requirements that "have the most impact on the architecture" [Kozaczynski 2002]. In [Firesmith 2006], Firesmith states that architectural requirements can be classified into three categories: quality requirements, architecturally-significant requirements, and architecture constraints. "Quality requirements specify a minimum level of a quality factor such as … availability, … configurability, … portability, reliability, … scalability" [Firesmith 2006] etc. Architecturally-significant requirements "are functional, data, and interface requirements that implicitly have a significant impact on the architecture" [Firesmith 2006]. An architecture constraint "is mandated on the architects as if it were a normal requirement" [Firesmith 2006]. Firesmith does not give further explanation of what an architectural constraint is, except that it is a decision not made by the architect but nonetheless impacts the architecture.

AASM is not able to distinguish who made the decision behind each requirement, but it is able to detect requirements that impose some kind of constraint on the architecture. In this paper, we do not show how AASM detects quality requirements, but we do show how it detects architectural constraints (Section 3.3) and architecturally-significant requirements (Section 3.4).

3.3. Step B: Identify Constraints

For AASM to consider a requirement as having an architectural constraint, the candidate requirement graph has to include a concept of type **System**. For these candidate requirements, there are two ways to detect an architectural constraint: 1) when there is a concept definition or other universal assertion in the candidate requirement (Section 3.3.1) or 2) the portion of the graph for the requirement contains a negated concept (Section 3.3.2). If a requirements graph meets one of the two criteria listed above, but does not include a **System** concept, AASM still considers it a constraint, but not an architectural constraint.

3.3.1. CG Definitions as Constraints

Distinguishing architecture from design impacts the first way AASM detects architectural constraints. Software requirements can be classified into different abstraction levels (business-level, domain-level, product-level, design-level, code-level) [Svetinovic 2003]. "Architecture-level specification lies conceptually between domain and product-level specifications" [Svetinovic 2003]. We have not identified any previous work that attempts to formally distinguish between architectural and design requirements.

However, in [Eden 2004], Eden, Hirshfeld and Kazman proposed a hypothesis to formally describe the distinction between architectural and design *statements*. They divided statements into three abstraction levels, listed here from most abstract to least abstract: "non-local" statements, "local and intensional" statements, and "extensional" statements. They placed architectural styles and design principles in the "non-local" category, design patterns such as the "Gang of Four" catalog [GoF 1995] in the "local and intensional" category, and class diagrams and program documentation in the "extensional" category.

Eden, Hirshfeld and Kazman had intended the hypothesis to apply to architecture and design statements; in this paper, part of the hypothesis will be applied to requirements that have been modeled into conceptual graphs. The goal in this paper is to distinguish between requirements that, once transformed into formal statements, can be categorized as either the "non-local" category or the "local and intensional" category. For brevity, these two categories will be called **non-local** and **local** in this paper. Informally, Eden, Hirshfeld and Kazman observed that "local statements are usually in the form: 'there exists an entity (or set of entities) that satisfies this condition', whereas non-local statements are in the form 'for all entities [that satisfy condition *1*], condition *2* applies'" [Eden 2004].

The architectural constraints that would likely impact the software architecture as a whole can be transformed to **non-local** design statements. If a requirement containing an architectural constraint is modeled in a conceptual graph as a **non-local** statement, it takes the form of a CG definition.

3.3.2. Negated Concepts as Constraints

As mentioned in Section 0, when a concept in a conceptual graph is negated, it means that the concept does not exist. Since software requirements are intended to be incorporated into a software architecture, a negated software requirement concept in a graph containing a concept of type **System** means that the concept *can never* exist in the resulting software system's architecture. Therefore, AASM interprets negated concepts as architectural constraints.

3.4. Step C: Identify the Connector Components

One challenge of modeling architecture in conceptual graphs is that there are no built-in architectural constructs. Therefore, AASM includes how to detect candidates for the CBSP architectural building blocks (also called architectural components). This paper will focus on describing connector component candidates because we have found that they are better for predicting architecture than the components [DeBraccio 2007] [Bhattacharya 2007]. The AASM techniques for finding the other architectural component candidates (i.e., the Data Component, the Process Component, and the System Component) are described in [DeBraccio 2007]. We now describe what architectural connector components are, and how AASM detects a connector component candidate.

Perry and Wolf describe connectors as "the glue that holds the different pieces of the architecture together" [Perry 1992]. As Fielding notes in [Fielding 2000], a more precise definition is provided by Shaw and Clements: "A connector is a mechanism that mediates communication, coordination, or cooperation among components … Examples include shared representations, remote procedure calls, message-passing protocols, data streams, and transaction streams" [Shaw 1997]. Bhattacharya and Perry add to the list of connector examples with "static calls, dynamic calls, … batch data, signals … and direct data access" [Bhattacharya 2007].

In conceptual graphs, we assume that action type concepts and CG actors imply a need for some kind of communication between the concepts they connect. Generally, a candidate connector component can be identified when a system or a part of a system performs an action on any concept outside the system. The type of each connector component will not be explicitly shown in the conceptual graphs; only the *need* for a connector component will be detected. The other type of connector component that AASM detects connects events; this connector is modeled in conceptual graphs by an ordering relation such as the **follows** relation.

AASM will detect a need for a connector component when a system or a concept within one system performs an action on a different system or on one of the different system's concepts. An action can be performed either by a CG actor (not discussed here) or an **Action** type concept. AASM searches each **Action** type concept for direct and indirect objects that are a part of a different system than the concept that initiated the action. In a conceptual graph, systems can contain other systems, so connector components can be found between sub-systems. See the following example from [Grüenbacher 2004] for connector components:

Requirement: "Manipulated spreadsheet data must be stored on the file system."

CBSP Connector Component: "Connector enabling interaction between UI and persistency components."

An ambiguity in the requirement is in identifying who or what stores the data. The entity might be a user of the Cargo Router system interface, or the Cargo Router system might be expected to automatically provide that service. The CBSP provides a process to gather that information from the stakeholders. AASM will assume that ambiguity was resolved in Step A. Figure 6 shows one way the refined requirement can be modeled.

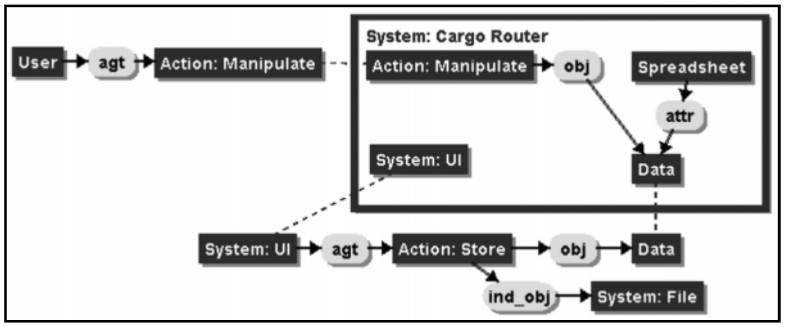


Figure 6: CG Model of a Requirement with Connector Components

The connector component is not explicitly shown in the conceptual graph, but it can be automatically detected that one is needed between the **Cargo Router** system and the **File** system. The **UI** entity which is a sub-system of the **Cargo Router** system performs an indirect action on the **File** system through the **Store** action. AASM detects that the receiving concept of the **ind_obj** relation from the **Store** action does not belong to the **UI** sub-system, and therefore flags that a connector was needed between the **UI** sub-system and the **File** system.

3.5. Step D: Find Candidate Properties For Connectors

As mentioned above, candidate properties are only sought for connector architectural elements. AASM focuses on the connector properties because usually the properties that uniquely identify the style are the data and control issues [Bhattacharya 2007]. Both data issues (Section 3.5.3) and control issues (Section 3.5.2) can be considered connector properties. Another reason this paper focuses on connector properties is because the connector properties are more clearly defined in supporting references.

The potential candidate properties for buses include: locality (Section 3.5.1), synchronicity (Section 3.5.2), and data continuity (Section 3.5.3).

Locality was chosen because it was a connector property in [Grüenbacher 2004]. Data continuity was chosen because it was used in [Bhattacharya 2007] to help determine an architectural style. Synchronicity was chosen because it was both a connector property in [Grüenbacher 2004] and it was used in [Bhattacharya 2007] to help determine an architectural style. In both [Shaw 1997] and [Bhattacharya 2007], the synchronicity property was considered a control issue and the data continuity property was considered a data issue.

3.5.1. Locality

In AASM, the locality of a connector can potentially be either distributed or local. A distributed connector connects concepts from different systems or different collection types (different collection types imply different sub-systems). A connector that connects sub-systems of same types of different names that are a part of a larger system is identified as a local connector.

3.5.2. Synchronicity

"Synchronicity is the nature of the dependence of the component's action upon each other's control state" [Bhattacharya 2007]. In [Shaw 1997], Shaw and Clements describe the different synchronicity types as Sequential (or lockstep), Synchronous, Asynchronous, and Opportunistic. When the synchronicity of a system is Sequential, the components execute in order; execution of one component cannot begin until its predecessor finishes [Shaw 1997]. In Synchronous systems, components coordinate control "regularly and often" [Shaw 1997]. In Asynchronous systems, components coordinate occasionally, but the flow of control is mostly unpredictable [Shaw 1997]. In Opportunistic systems, the components act as "autonomous agents [working] completely independently from each other in parallel" [Shaw 1997].

In [Bhattacharya 2007], Bhattacharya and Perry devised an algorithm to detect these four synchronicity types. As mentioned in Section 2.2, Bhattacharya and Perry proposed a way to organize architectural specification documents to facilitate searching the documents for architectural properties. Bhattacharya's and Perry's synchronicity algorithm searches a list of input and output event specifications of the Service Event Specifications as defined in their documentation model. The algorithm determines that the synchronicity is Sequential if, in the process of searching the list of software components, the output events of one component match the input events of the next component; Synchronous if at any point in searching the list, the output events of all preceding components match the input event of the next component; Opportunistic if the input events of a component all match the output events within the same component and do not match the output events of any other component; otherwise, if no other synchronicity type is detected, the synchronicity is Asynchronous [Bhattacharya 2007].

Bhattacharya and Perry detected synchronicity by analyzing the behavior of events with respect to each other. Similarly, AASM will detect synchronicity in conceptual graphs by analyzing how events relate to each other. As stated in [DeBraccio 2007], AASM identifies an event type as a process component. The events are identified as Sequential if an ordering relation, such as the CG **follows** relation, is between the events. If more than one event is connected to another event by an ordering relation, then the synchronicity is Synchronous. All other potential connectors described in Section 3.4 are considered Asynchronous. The opportunistic connectors are handled in an alternate analysis (see Section 4.5).

A blurred line exists between process components and connectors when identifying events. In [Mehta 2000], some types of connectors are events themselves. It is not always clear whether the event type modeled in a requirements conceptual graph should be an event connector component or an event process component. This paper will focus on the way in which the events interact with each other; in other words, the property of the connector between the events.

3.5.3. Data Continuity

"Continuity is a measure of the flow of data through the system" [Bhattacharya 2007]. Bhattacharya and Perry proposed two classifications of data continuity: Sporadic and Continuous. "While in a continuous flow system, new data is available at all times, in a sporadic flow system, new data is generated at specific intervals" [Bhattacharya 2007]. Since data continuity is a data issue, an architectural data component must be involved in the portion of the model that is being analyzed.

Sporadic data would occur when a **User** type gives information to the system. This is detected when a process is initiated by a **User** type that acts on a data component. Continuous data is detected when data components are transmitted via connectors from one main "input" to one main "output" with little user interaction between (measured relative to the whole graph) at a continuous rate.

3.6. Step E: Architectural Style Prediction Scoring

To determine the most applicable architectural style, Table 4 was modified from the CBSP approach, to focus on the connector properties. The CBSP connector properties from the example in [Grüenbacher 2004] that were included in the table were Local, Distributed, Synchronous, and

Asynchronous. As shown in Table 4 three rows were added for the three additional properties that were explored in [Bhattacharya 2007]: Sequential (a synchronicity issue), and Sporadic and Continuous (both data continuity issues).

Table 4
Connector Property Support for Architectural Styles

		Client-Server	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
Locality	Local	—	++	-	++	+
	Distributed	++	++	++	—	+
	Sequential	—	—	—	++	—
Synchronicity	Synchronous	++	—	+	++	—
	Asynchronous	—	++	++	—	++
Data	Sporadic	++	++	++	++	—
Continuity	Continuous	—	—	—	—	++

Legend: ++ extensive support + some support - marginal support — no support

In [Bhattacharya 2007], support for the three additional properties among various styles was expressed in basically a "yes" or "no" answer, but the properties were not rated for partial support. For this paper, the "yes" rankings of support were interpreted as 'Extensive Support' and the "no" rankings were interpreted as 'No Support.' Future work will explore how to assign each style the partial support rankings for the three additional properties (see [DeBraccio 2007]).

Neither [Bhattacharya 2007] nor [Grüenbacher 2004] provided any information on C2 support for these three properties. The C2 support rankings for data continuity were estimated based on the research of Taylor et al., the authors that introduced the C2 architecture in [Taylor 1996]. C2 was designed (but not restricted) to support applications with a Graphical User Interface (GUI) [Taylor 1996]. In GUI applications, "both users and the application perform actions concurrently and at arbitrary times" [Taylor 1996]. Because the actions taken at arbitrary times suggest sporadic data flow, we assumed C2 provides 'Extensive Support' for Sporadic data. Since the definition from Section 3.5.3 suggests that the data continuity would either be Sporadic or Continuous, but not both, we assumed that any one architectural style would probably only support one property or the other. Consequently, the C2 support for Continuous data was estimated as 'No Support.' Future work may explore multiple-style architectures [DeBraccio 2007] in which the two types of data continuity can co-exist in the same architecture. In this paper, C2 was assumed to provide 'No Support' for the Sequential property because "while the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components" [Medvidovic 1997].

[Bhattacharya 2007] and [Grüenbacher 2004] conflict regarding the support that the Layered style provides for asynchronous connectors: in [Grüenbacher 2004], the support is considered the lowest ranking, but in [Bhattacharya 2007], the Layered style is considered capable of supporting any synchronicity property, including the Asynchronous property. For this paper, the support ranking from [Grüenbacher 2004] was chosen over the support ranking from [Bhattacharya 2007]. [Grüenbacher 2004] provides more of a granularity in support rankings, and recognizing partial support for properties provides a finer distinction between the styles.

To determine an architectural style from the numbers of connector properties, 0 was first converted to Table 3.1. Instead of using plus and minus signs to convey the support ranking, numbers (0 to 3) were used, the highest number being assigned to 'Extensive Support,' and the lowest value being assigned to 'No Support.'

Table 3.1
Number-based Connector Property Support for Architectural Styles

		Client-Server	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
Locality	Local	0	3	1	3	2
	Distributed	3	3	3	0	2
	Sequential	0	0	0	3	0
Synchronicity	Synchronous	3	0	2	3	0
	Asynchronous	0	3	3	0	3
Data	Sporadic	3	3	3	3	0
Continuity	Continuous	0	0	0	0	3

Legend: 3 extensive support 2 some support 1 marginal support 0 no support

To calculate the Architectural Style Prediction Score (***ASPS***), each entry in Table 3.1 was treated as a multiplier for the corresponding number of connector properties (to be tallied from the results table). The connector properties scores were then added up per architectural style.

The following formula summarizes these steps:

$$\text{ASPS} = \sum_{i=1}^{INCP} CPSR \times ncp$$

- ***TNCP***: The total number of connector properties. In this paper, AASM searches for seven connector properties: Local, Distributed, Sequential, Synchronous, Asynchronous (or Opportunistic), Sporadic, and Continuous.

- ***CPSR***: The connector property support ranking for property ***i***; the values are from Table 3.1.
- ***ncp***: The number of times each connector property ***i*** was counted.

The resulting scores are then compared; whichever style receives the highest score is the recommended style for the software architecture.

3.7. AASM Examples

This section illustrates AASM Steps A through D with several examples. The examples show how AASM behaves 1) when architectural components are detected in requirements (Section 3.7.2), 2) when architectural constraints are detected in requirements (Section 3.7.3), and 3) when no architectural constraints or components are detected (Section 3.7.1). The requirements were taken from the Scheduler project discussed in Section 4. The full set of Scheduler software requirements are listed in Appendix F of [DeBraccio 2007], the conceptual graphs for the Scheduler requirements are in Appendix I of [DeBraccio 2007], and the AASM results for the Scheduler are in Appendix J of [DeBraccio 2007].

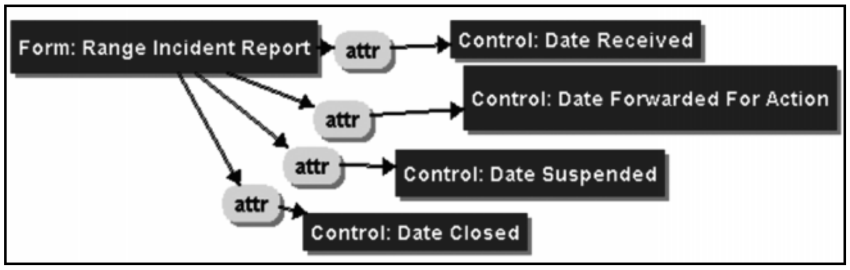
3.7.1. Detecting a Non-Architectural Requirement

The conceptual graph model of the following requirement was found to have no architectural constraints and no architectural components.

Scheduler Requirement #9:

Incident report records should show, date received; date forwarded for action, date suspended; and date closed (incident report history).

The "Incident report" refers to the "Range Incident Report" form in the Scheduler system. 0 shows how the requirement was modeled into a conceptual graph.



CG Model of Scheduler Requirement #9

AASM does not detect a system or a sub-system because none of the CG concepts is a System type. AASM does not detect a Process component because none of the CG concepts is an **Action** type. The **Control** concept types inherit from the **Entity** type. They qualify as data attributes of the **Range Incident Report** form. However, AASM does not detect them as *architectural* data components since there is no Process component that acts on the data. Table 3.2 shows the AASM table entry for Scheduler Requirement #9.

Table 3.2
AASM Table Entry for Scheduler Requirement #9

Req't ID	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
9	not identified	not identified	*(Local data* attributes: Control: Date Received, Control: Date Forwarded For Action, Control: Date Suspended, Control: Date Closed)	not identified	not identified	not identified

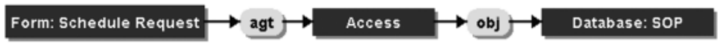
3.7.2. Detecting Architectural Components

AASM detected architectural components in the conceptual graph model of Scheduler Requirement #13.

Scheduler Requirement #13:

The SR form shall integrate with the SOP database.

The "SR" form is the **Schedule Request** form for the Scheduler system. "Integrate" was interpreted as an **Access** action type in the CG model shown in 0.



CG Model of Scheduler Requirement #13

AASM does not detect a system or a sub-system because none of the CG concepts is a System type. AASM detects the **Access** action type as a Process component. The **Database: SOP** concept is detected as a data component because it is an **Entity** type concept and the **Access** process component acts on it.

AASM detected a connector was needed between the Schedule Request Form and the SOP Database because the two CG types, **Form** and **Database**, are from different system types: the **Form** inherits from the **UserInterface** type, which is a different system type from the **Database** type. The fact that the **Form** and **Database** types are from different system types also means that the connector property is detected to be distributed. Table 3.3 shows the AASM table entry for Scheduler Requirement #13.

Table 3.3
AASM Table Entry for Scheduler Requirement #13

Req't ID	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
13	not identified	not identified	Database: SOP	Access	Form: Schedule Request & Database: SOP	Distributed

3.7.3. Detecting an Architectural Constraint

This section provides examples for the two ways AASM detects an architectural constraint as described from Section 3.3.2: by a CG definition (Section 3.7.3.1) and by negated concepts (Section 3.7.3.2).

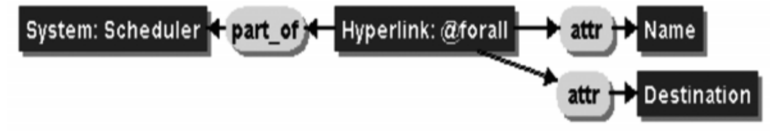
3.7.3.1. Detecting a CG Definition

AASM detected a CG definition in the conceptual graph model of Scheduler Requirement #52.

Scheduler Requirement #52:

Clear shortcuts through the system should be defined.

As shown in 0, when Requirement #52 was modeled into a conceptual graph, "Clear shortcuts" was interpreted as the **Hyperlink** type, and "defined" was interpreted as defining the hyperlink by assigning it **Name** and **Destination** attributes.



CG Model of Scheduler Requirement #52

The system in Requirement #52 is the Scheduler system. AASM detects this requirement as an architectural constraint because the Hyperlink concept is labeled with the @forall symbol, meaning that the attributes and other properties of the Hyperlink concept apply to all Hyperlink types in the model. Table 3.4 shows the AASM table entry for Scheduler Requirement #52.

Table 3.4
AASM Table Entry for Scheduler Requirement #52

Req't ID	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
52	Arch. Constraint: for all Hyperlinks	Scheduler	(Local data attributes: Name, Destination)	not identified	not identified	not identified

3.7.3.2 Detecting Negated Concepts

AASM detected negated concepts in the conceptual graph model of Scheduler Requirement #50.

Scheduler Requirement #50:

SR work flow shall allow TD to go forward and or go and get needed

information without having to get out and restart the form again.

Figure 7 shows that the "SR" acronym was interpreted to be a Schedule Request Form type. The "TD" acronym was interpreted to be a Test Director User type. The "work flow" was interpreted to be an interrupt initiated by the Scheduler system.

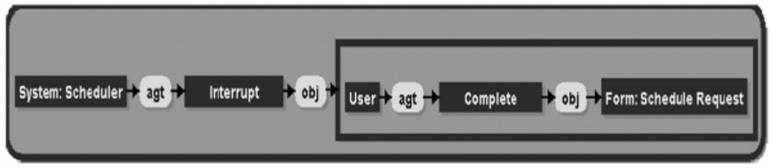


Figure 7: CG Model of Scheduler Requirement #50

The constraint was an architectural constraint because the Scheduler system was included in the requirement, but AASM detected no architectural components from the conceptual graph in 0 because all the concepts were negated. Table 3.5 shows the AASM table entry for Scheduler Requirement #50.

Table 3.5
AASM Table Entry for Scheduler Requirement #50

Req't ID	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
50	Arch. Constraint: Negated concepts involving a system	not identified	not identified	not identified	not identified	not identified

4. RESULTS

To validate the AASM, the process was run on two sets of software requirements. The first set of software requirements (from [Ambriola 1997]) describes how the CIRCE system works. Those requirements were chosen because since they had already been validated by the CIRCE process, they should already be relatively free of ambiguities. The second, larger set of software requirements was taken from an industrial software project which will be called Scheduler in this paper. The Scheduler requirements were taken directly from the final draft of the requirements document for Scheduler version 1.0; no formal process was taken to purge ambiguities from the Scheduler requirements once they were taken from the requirements document.

In Step A of AASM, the requirements were modeled in conceptual graphs. Steps B and C (identifying the architectural requirements) were performed in one stroke: architectural requirements are the requirements that have an architectural constraint or they have at least one architectural component. In Step D, the architectural connectors were analyzed to determine properties. In Step E, the connector properties were compared to predict a plausible overall architectural style.

4.1. CIRCE Results

The twelve CIRCE software requirements from [Ambriola 1997] and the conceptual graphs that model those requirements are documented in [DeBraccio 2007], as well as the tailored CIRCE type hierarchy. Table 4.1 is an excerpt from the AASM analysis for CIRCE. The table lists the architectural constraints, architectural components, and the connector properties that were found in the CIRCE conceptual graphs for the first three CIRCE software requirements.

Table 4.1
Excerpt of CIRCE Requirements Analysis

Req #	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
1	not identified	System: CIRCE, System: Web Interface, System: View Module, System: Cico, System: View Selector	not identified	not identified	not identified	not identified
2	not identified	Information: User Reqs	Information: User Reqs	Receive	System:Web Interface & Information: User Reqs	[Asynchronous or Opportunistic] Distributed
	not identified	Information: Glossary	Information: Glossary	Receive	System: Web Interface & Information: Glossary	[Asynchronous or Opportunistic], Distributed
3	not identified	Information: User Reqs	not identified	not identified	not identified	not identified

4.2. Scheduler Results

The eighty-eight Scheduler software requirements, the Scheduler type and relation hierarchies, CG definitions, conceptual graph models, and AASM results are documented in [DeBraccio 2007]. Table 4.2 is an excerpt of the AASM results for Scheduler.

Table 4.2
Excerpt of Scheduler Requirements Analysis

Req #	*Constraint*	*(sub) System*	*Data*	*Process*	*Connector*	*Connector Properties*
75	not identified	Form: Firing Program	Form: Schedule Request	Auto-populate	Form: Schedule Request & Form: Firing Program	[Asynchrono us or Opportunistic], Local
77	not identified	Schedule r	Query	Clear	System: Scheduler & Form: Firing Program	[Asynchrono us or Opportunistic], Local
71	not identified	Database : Tube	Form: Schedule Request	Auto-populate	Form: Schedule Request & Database: Tube	[Asynchrono us or Opportunistic], Distributed

4.3. Requirement and Connector Totals

Table 4.3 lists the total numbers of requirements and the numbers of potential architectural requirements for each project. As mentioned in Section 3.2, the architectural requirements could be architecturally-significant requirements or architectural constraints. The architecturally-significant requirements are the requirements in which AASM found at least one architectural component candidate. Some Scheduler requirements qualified as both an architectural constraint and an architecturally-significant requirement.

Table 4.3
Numbers of Architectural Requirements and Total Requirements

	CIRCE	*Scheduler*
Numbers of Architectural Constraints	0	8
Numbers of Architecturally-Significant Requirements	12	63
Numbers of Architecturally-Significant Requirements with at least one Connector	9	53
Total Requirements	12	88

Table 4.4 shows the numbers of connector properties for each project. A single requirement can have more than one connector, and a single connector can have more than one property.

Table 4.4
Numbers of Connector Properties

		CIRCE	*Scheduler*
Locality	Local	0	24
	Distributed	19	8
Synchronicity	Sequential	2	15
	Synchronous	0	0
	Asynchronous	23	62
Data	Sporadic	2	22
Continuity	Continuous	0	0
Total Connector Properties		46	131

4.4. Architectural Style Results with Asynchronous Connectors

The architectural style that has the highest *ASPS* is the recommended style for that project.

Table 4.5
Architectural Style Prediction Scores with Asynchronous Connectors

	Client-Server	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
CIRCE ASPS	63	134	132	12	107
Scheduler ASPS	90	348	300	183	250

4.5. Architectural Style Results with Opportunistic Connectors

An alternative analysis of the AASM connector properties is to consider the Asynchronous connectors as Opportunistic connectors instead. Interpreting the connector properties found by AASM as Opportunistic instead of Asynchronous would be a defendable viewpoint, since conceptual graphs lend themselves to stating the existence of concepts and groups of concepts that exist in parallel with each other. In [Bhattacharya 2007], the only style of the five explored in this paper that is shown to support Opportunistic connectors is the Layered Style. Table 4.6 shows a possible interpretation of how these five styles may support the Opportunistic connector property.

Table 4.6
Opportunistic Connector Property support for Architectural Styles

		Client-Server	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
Synchronicity	Opportunistic	0	3	0	3	0

Legend: 3 extensive support 2 some support 1 marginal support 0 no support

No values for C2 were given in [Bhattacharya 2007] or [Grüenbacher 2004] for Opportunistic support, so the C2 support rankings were estimated in this paper based on the research of Taylor et al. Since C2 was described as supporting multiple threads in "multi-user and concurrent applications" [Taylor 1996], C2 was assigned a high support ranking in Table 4.6.

Table 4.7 can be compared to Table 4.5; Table 4.7 shows the alternate ASP scores if all the Asynchronous connectors in both projects were considered instead as Opportunistic connectors.

Table 4.7
Architectural Style Prediction Scores with Opportunistic Connectors

	Client-Server	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
CIRCE ASPS	63	134	63	81	38
Scheduler ASPS	90	363	114	369	64

5. DISCUSSION

This section will discuss 1) the analysis challenges encountered during the AASM process (Section 5.1), 2) the numbers of requirements (Section 5.2), 3) the results of the analysis of asynchronous connectors (Section 5.3.1) and opportunistic connectors (Section 5.3.2), and 3) the actual styles of the projects (Section 5.4).

5.1. Analysis Challenges

During the analysis of each project, challenges were encountered in the Software Requirements and in Conceptual Graph Modeling. The main challenges regarding the software requirements were ambiguous requirements, redundant requirements, and tasks that were bundled into the software requirements that were not actually meant to be modeled into the software project. The method in this paper needed the requirements to be unambiguous enough to identify potential architectural components and potential connector properties; whether or not the requirements were unambiguous enough for software design tasks to commence is outside this paper's scope. If redundant requirements were modeled into the conceptual graphs, any architectural components that may have been detected were not double-counted. Also, when the requirements were modeled into conceptual graphs, no attempt was made to distinguish between the "real" requirements and any pre-requirement tasks (meant for the customer(s) to perform prior to requirements gathering) or pre-design tasks (meant for the software developers to perform prior to modeling the software).

Conceptual Graph Modeling challenges included the need to add to the conceptual graph definitions and to the type hierarchy, inaccurate or ambiguous modeling into conceptual graphs, and the fact that there are many different ways to model a software requirement into conceptual graphs.

Part of the difficulty is the fact that conceptual graphs are limited when attempting to model complicated expressions of time. As mentioned in Section 0, conceptual graph notation does provide some symbols for modeling temporal statements. Modeling temporal statements was important, particularly for the Scheduler, a system whose central purpose involved relating activities to time. In many cases, the problem was overcome with the use of an ordering relation like follows (see Scheduler Requirements #20, #34, #38, and #85 in [DeBraccio 2007]), with the use of the **past** relation (see Scheduler Requirements #2 and #10 in [DeBraccio 2007]). Some requirements, though, were like Scheduler Requirement #50 (discussed in Section 3.7.3.2). Scheduler Requirement #50 included different verb tenses within the same sentence, so it had to be interpreted broadly to model the meaning.

5.1.1. CIRCE Challenges

No challenges were encountered regarding the CIRCE requirements because none of these particular requirements were found to be ambiguous (that is, in some way that might hinder identifying candidates for architectural components).

A conceptual graph modeling challenge was that many CIRCE entity concepts seemed to fit into both the 'Data' component category and the 'System' category. This is important because AAMS's method of detecting connectors depends on whether a component was a system or not. In the results table, these CG concepts were treated as candidates for both data components and system components (see Appendix E in [DeBraccio 2007] for requirements 2, 6, 8, 10 and 11).

5.1.2. Scheduler Challenges

Several ambiguous requirements were found in the Scheduler requirements during the AASM analysis. The ambiguous requirements often resulted in undefined connector properties.

This was because the entity type that initiates the action on the data is not specific enough to determine which property should be chosen. Theoretically, these ambiguities would have been purged if the software requirements had gone through the same filtering process as the CIRCE software requirements.

Redundant requirements that suggest an architectural component means that the architectural component is listed twice in the results table; however, the redundant components are not counted twice in the results. We recognize that some requirements would have been detected as constraints or as having architectural components if they had been modeled in conceptual graphs more accurately.

5.2. Numbers of Requirements

In Table 4.3, the number of architectural requirements with respect to the total number of requirements is usually expected to be small [Grüenbacher 2004]. The fact that AASM discovered most of the requirements as an architectural requirement means that further research is needed for AASM to distinguish between architectural-level requirements and design-level requirements. However, AASM provided a high coverage of the conceptual graphs, so a large percentage of the graphs contributed to the prediction of a suitable architectural style. The requirements that contributed to the style prediction score were the architecturally-significant requirements with at least one connector. That means 75% of the CIRCE requirements and over 60% of the Scheduler requirements contributed to the style prediction score in Section 4.4.

5.3. Analysis of Architectural Style Results

The ASPS scores within a particular project such as those in Table 4.5 are meant to be interpreted relative to each other, and the scores for one project are not meant to be directly compared with the scores of another project. Section 5.3.1 discusses the Asynchronous Connector results and Section 5.3.2 discusses the Opportunistic Connector results.

5.3.1. Analysis with Asynchronous Connectors

In Table 4.5, both projects ranked C2 as the highest, Event-Based as the second highest, and Pipe-and-Filter as the third highest. The reason is probably because only the connector properties are analyzed. The C2 style happens to support several different connector properties very well. Both projects contained many potential candidates for asynchronous connectors and sporadic connectors, and both the C2 and Event-Based styles support both those types of connectors well. The Pipe-and-Filter style was ranked third because it provides support for asynchronous connectors.

For the purposes of AASM, the scores for the C2 and Event-Based styles were nearly identical for the CIRCE project. Both styles not only received high scores, but they received similar scores, meaning that both are a recommended choice for CIRCE. The style prediction scores could be more reliable if the properties for the other architectural components are analyzed in future work.

5.3.2. Analysis with Opportunistic Connectors

In Table 4.7, both the layered style and the C2 style are favored above the other styles for both projects. The fluctuation in the scores for the Event-Based, Layered, and Pipe-and-Filter styles shows that in this stage of AASM's development, AASM is very sensitive to different interpretations of the conceptual graph model; ideally, the ASP scores would not fluctuate as much among different interpretations. The sensitivity of AASM indicates that further research is needed in how to distinguish between Opportunistic and Asynchronous connectors in conceptual graphs.

5.4. Styles Implemented

Information on the actual architectural style(s) in which CIRCE was implemented was not available for this paper. Scheduler 1.0 was developed into a web-based database application which was implemented with both a layered style and a client-server style. It would be interesting to see in future work, when the properties for the other architectural components are analyzed, if the Scheduler AASM results would more closely favor the actual chosen styles.

6. CONCLUSION

In the context of the human-intensive CBSP process, AASM could be a valuable support tool because it would serve as a voter free of human bias. In addition, AASM contributes to the following research areas:

AASM identifies some types of architecture-level requirements, a topic of needed research ([Svetinovic 2003], [Firesmith 2006]).

AASM provides the potential for tool support in the transition from software requirements to software architecture, described as a needed research area in [Galster 2006].

AASM provides the groundwork for an automated tool that can help a software architect construct a multiple-style architecture. Tool support for multiple-style architecture recommendations was described as future work for CBSP in [Grüenbacher 2004]. An important future goal for AASM would be the realization of this potential.

In summary, AASM is an approach towards selecting an architectural style from a given set of natural language software requirements with the focus on automating the selection of styles.

No fully automated tools exist to achieve this goal; AASM provides groundwork for future research in this area. As discussed in [DeBraccio 2007], AASM provides a starting point for a variety of research areas that target bridging the gap between software requirements and software architecture.

REFERENCES

[1] [Galster 2006] M. Galster, A. Eberlein, and M. Moussavi, "Transition from Requirements to Architecture: A Review and Future Perspective," in *7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2006, pp. 9-16.

[2] [Grüenbacher 2004] P. Grüenbacher, A. Egyed, and N. Medvidovic, "Reconciling Software Requirements and Architectures with Intermediate Models," *Software and Systems Modeling*, **3,** No. 3, pp. 235-253, August 2004.

[3] [Kruchten 2006] P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present and Future of Software Architecture," *IEEE Software*, **23**, No. 2, pp. 31-39, March-April 2006.

[4] [Clements 2003] P. Clements, et al., *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[5] [Bass 2003] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[6] [Bhattacharya 2007] S. Bhattacharya and D. E. Perry, "Predicting Emergent Properties of Component based Systems," in *6th IEEE International Conference on COTS-based Software Systems*, 2007, pp. 41-50.

[7] [Perry 1992] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, **17**, No. 4, pp. 40-52, October 1992.

[8] [Pressman 2005] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. McGraw-Hill, 2005.

[9] [Nuseibeh 2001] B. Nuseibeh, "Weaving Together Requirements and Architectures," *IEEE Computer*, **34**, No. 3, pp. 115-117, March 2001.

[10] [DeBraccio 2007] D. DeBraccio, "Towards Automating Software Architectural Style Selection from Requirements," M.S. Thesis, *Department of Computer Science*, University of Alabama in Huntsville, 2007.

[11] [Shaw 1996] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[12] [Shaw 1997] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," in *Proceedings of the 21st International Computer Software and Applications Conference*, 1997, pp. 6-13.

[13] [Cyre 1997] W. R. Cyre, "Capture, Integration and Analysis of Digital System Requirements with Conceptual Graphs," *IEEE Transactions on Knowledge and Data Engineering*, **9**, No. 1, pp. 8-23, 1997.

[14] Land 2002] R. Land, "A Brief Survey of Software Architecture," Mälardalen Real-Time Research Centre, *Department of Computer Engineering*, Mälardalen University, Västerås, Sweden, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-57/2002-1-SE, 2002.

[15] [Bengtsson 1999] P. Bengtsson and J. Bosch, "Haemo Dialysis Software Architecture Design Experiences," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 516-526.

[16] [Monroe 1996] R. T. Monroe and D. Garlan, "Style-based Reuse for Software Architectures," in *Proceedings of the 4th International Conference on Software Reuse*, 1996, p. 84.

[17] [Medvidovic 2003] N. Medvidovic, A. Egyed, and P. Grüenbacher, "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery," in *Proceedings of the 2nd International Software Requirements to Architectures Workshop*, 2003, pp. 61-68.

[18] [GoF 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison Wesley, 1995.

[9] [Monroe 1997] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural Styles, Design Patterns and Objects," *IEEE Software*, **14**, No. 1, pp. 43-52, January 1997.

[10] [Ambriola 2003] V. Ambriola and V. Gervasi, "The CIRCE Approach to the Systematic Analysis of NL Requirements," University of Pisa, Dipartimento di Informatica, Italy, Tech. Rep. TR03-05, 2003.

[11] [McGraw-Hill 2007] "Natural Language Processing," in *McGraw-Hill Encyclopedia of Science and Technology*. [Online]. Available: Answers.com Web site, http://www.answers.com/topic/naturallanguage-recognition. [Accessed: January 18, 2007].

[12] [Gervasi 2000] V. Gervasi and B. Nuseibeh, "Lightweight Validation of Natural Language Requirements: A Case Study," in *Proceedings of the 4th International Conference on Requirements Engineering*, 2000, pp. 140-149.

[13] [Lenat 1995] D. B. Lenat, "Cyc: A Large-scale Investment in Knowledge Infrastructure," *Communications of the ACM*, **38**, No. 11, November 1995.

[14] [Hars 1996] A. Hars, "Advancing CASE Productivity by using Natural Language Processing and Computerized Ontologies: The ACAPULCO System," in *Proceedings of the 2nd Americas Conference on Information Systems*, 1996, pp. 898-900.

[15] [Kiyavitskaya 2004] N. Kiyavitskaya, N. Zeni, L. Mich, and J. Mylopoulos, "NLP-based Requirements Modeling: Experiments on the Quality of the Models," Informatica e Telecomunicazioni, University of Trento, Italy, Tech. Rep. DIT-04-005, 2004.

[16] [Sowa 2000] J. F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks Cole Publishing Co., 2000.

[17] [Sowa 1984] J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, 1984.

[18] [Delugach 2006] H. S. Delugach, Ed., "Conceptual Graphs Home Page," July 25, 2006. [Online]. Available: http://www.conceptualgraphs.org.

[19] [Delugach 1991] H. S. Delugach, "Dynamic Assertion and Retraction of Conceptual Graphs," in E. C. Way, Ed., *Proceedings of the 6th Annual Workshop on Conceptual Graphs*, 1991, pp. 15-26.

[20] [Delugach 2003] H. S. Delugach, *CharGer Conceptual Graph Editor*, Huntsville, AL, 2003. [Online]. Available: http://www.sourceforge.net/projects/charger.

[21] [Ryan 1993] K. Ryan and B. Mathews, "Matching Conceptual Graphs as an Aid to Requirements reuse," in *Proceedings of the IEEE Symposium on Requirements Engineering*, 1993, pp. 112-120.

[22] [Delugach 1991] H. S. Delugach, "A Multiple-viewed Approach to Software Requirements," Ph.D. Dissertation, *Department of Computer Science*, University of Virginia, Charlottesville, 1991.

[23] [Delugach 1992] H. S. Delugach, "Specifying Multiple-viewed Software Requirements with Conceptual Graphs," *Journal of Systems and Software*, **19**, No. 3, pp. 207-224, 1992.

[24] [Delugach 1996] H. S. Delugach, "An Approach to Conceptual Feedback in Multiple Viewed Software Requirements Modeling," in *Viewpoints 96: International Workshop on Multiple Perspectives in Software Development*, 1996, pp. 242-246.

[25] [Glaze 2004] G. Glaze, "Automating Software Architecture Selection and Generation," M.S. thesis, Department of Computer Science, University of Alabama in Huntsville, 2004.

[26] [Taylor 1996] R. N. Taylor, et al., "A Component- and Message-based Architectural Style for GUI Software," *IEEE Transactions on Software Engineering*, **22**, No. 6, pp. 390-406, June 1996.

[27] [Bachmann 2003] F. Bachmann, L. Bass, and M. Klein, "Preliminary Design of ArchE: A Software Architecture Design Assistant," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2003-TR-021, 2003.

[28] [Lee 2005] J. Lee and L. Bass, "Elements of a Usability Reasonability Framework," *Software Engineering Institute*, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2005-TN-030, 2005.

[29] [Gurp 2002] J. Gurp, R. Smedinga, and J. Bosch, "Architectural Design Support for Composition & Superimposition," Presented at *35th Annual Hawaii International Conference on System Sciences*, 2002.

[30] [Svetinovic 2003] D. Svetinovic, "Architecture-level Requirements Specification," in *Proceedings of the 2nd International Software Requirements to Architectures Workshop*, 2003, pp. 14-19.

[31] [Eden 2004] A. H. Eden, Y. Hirshfeld, and R. Kazman, "Abstraction Strata in Software Design," *Department of Computer Science*, University of Essex, United Kingdom, Tech. Rep. CSM-411, 2004.

[32] [Kozaczynski 2002] W. Kozaczynski, "Requirements, Architectures and Risks," in *10th Anniversary Joint IEEE International Requirements Engineering Conference,* 2002, pp. 6-7.

[33] [Firesmith 2006] D. G. Firesmith, "Architecture-related Requirements," *Journal of Object Technology*, **5**, No. 2, pp. 61-73, March-April 2006. [Online]. Available: http://www.jot.fm/issues/issue_2006_03/column7.

[34] [Fielding 2000] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. Dissertation, University of California, Irvine, 2000.

[35] [Kazman 1997] R. Kazman, P. Clements, L. Bass, and G. Abowd, "Classifying Architectural Elements as a Foundation for Mechanism Matching," in *Proceedings of COMPSAC-97*, 1997, pp. 14-17.

[36] [Mehta 2000] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp. 178-187.

[37] [Medvidovic 1997] N. Medvidovic and R. N. Taylor, "Exploiting Architectural Style to Develop a Family of Applications," *IEE Proceedings – Software Engineering*, **144**, No. 5-6, pp. 237-248, October-December 1997.

[38] [Ambriola 1997] V. Ambriola and V. Gervasi, "Processing Natural Language Requirements," in *1997 International Conference on Automated Software Engineering,* 1997, pp. 36-45.