Predicting Emergent Properties of Component Based Systems

Sutirtha Bhattacharya, Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
Austin, TX-78712
sutirtha.bhattacharya@intel.com
perry@ece.utexas.edu

Abstract

Software Product Lines (SPL), Component Based Software Engineering (CBSE) and Commercial Off the Shelf (COTS) components provide a rich supporting base for creating software architectures. Further, they promise significant improvements in the quality of software configurations that can be composed from pre-built components. Software architectural styles provide a way for achieving a desired coherence for such component-based architectures. This is because the different architectural styles enforce different quality attributes for a system. If the architectural style of an emergent system could be predicted in advance, the System Architect could make necessary changes to ensure that the quality attributes dictated by the system requirements were satisfied before the actual system was deployed. In this paper we propose a model for predicting architectural styles, and hence the quality attributes, based on use cases that need to be satisfied by a system configuration. Our technique can be used to determine stylistic conformance and hence indicate the presence or absence of architectural drift.

1. Introduction

Software architecture styles represent a cogent form of codification [1, 2, 3] of critical aspects to which an architecture is expected to conform. They differ from patterns in that patterns are the result of a discovery process, not a constraint process. Of course, patterns may play an important role in the creation and specification of a style: commonly occurring patterns provide a useful basis for codification. Part of the confusion comes from the fact that styles can be viewed both prescriptively (i.e., as a complex constraint that must be satisfied) and descriptively (i.e., as a description of what exists).

In 1997 Mary Shaw and Paul Clements proposed a feature-based classification of architectural styles [3]. They proposed that different architectural styles can be discriminated among each other by focusing on the following feature categories.

- Constituent Parts i.e. the components and connectors.
- Control Factors i.e. the flow of control among components.
- Data Factors i.e. details on how data is processed
- Control/Data Interaction i.e. the relation between control and data.

Even after years of software engineering research, the relationship between software components and architectural styles hasn't been adequately explored. This, in fact, is surprising given the attention Component Based Software Engineering has received in the recent past. However, if we explore the motivation of these two disciplines, we would realize that the relationship may not be obvious.

The focus of CBSE is to build software systems using pre-existing (including COTS) components thus reducing software costs and delivery time. Attention has mostly been directed towards understanding and resolving integration issues between the various components and establishing a common vocabulary for facilitating integration. The focus of Software Architecture, on the other hand, has been on the initial structure and constraints of complex software systems.

The critical question is: when designing software systems from components, should we leave the emerging architectural styles of a software system to pure chance or should we investigate the component characteristics that need to be understood, to enforce an architectural style by choice. Since different architectural styles support distinct sets of quality attributes, the benefit of evaluating components for suitability to an architectural style is obvious, as the

desired quality attributes for a system are often dictated by the system requirements. Quality attributes are essentially the benchmarks that describe a system's intended properties in the context of the environment for which it was built. It includes system characteristics such as performance, security, availability, usability etc. The ability to determine the architectural style for a system configuration will help us predict whether the desired quality attributes will be satisfied by the system prior to actual deployment.

In this paper we propose a model for documenting component specifications and demonstrate how we can reason over the specifications to determine the emergent architectural style. We analyze the different feature categories proposed by Shaw and Clements and identify the component attributes that would help determine the architectural style of a system configuration. Section 2 of the paper provides the background for our proposal while in Section 3 we perform the feature category analysis. Section 4 outlines the steps for style determination. Section 5 presents a case study conducted while in Section 6 we document related work. Section 7 concludes the paper.

2. Background

The context for the proposed research is outlined in this section. We start with the assumption that there exists a component repository in which software components relevant for a particular application domain have been specified using our asset specification model (briefly explained here) against our architectural specification. The System Architect identifies a deployment use-case or usage scenario (consisting of a list of services that needs to be delivered by the system) that needs to be implemented using pre-built components. For identifying the configuration of components that are needed to satisfy the use case, the Architect queries the repository for the available components that can potentially be used to satisfy the targeted scenario. The architectural style related reasoning that we are proposing will be done on the set of components returned by the component repository based on the System Architect's query. The envisioned reasoning capabilities will facilitate i) determining whether the set of components returned by the repository conform to any specific architectural style, ii) identifying a set of components that conform to a desired architectural style and hence support the desired set of quality attributes.

Before we begin, we briefly explain with the help of UML diagrams, our specification approach which will be leveraged for the style related reasoning. Our specification model captures an architecture in terms of architectural elements. These elements are essentially the components and connectors that are relevant for the application domain and enable functional partitioning. Figure 1 elaborates architectural element specification.

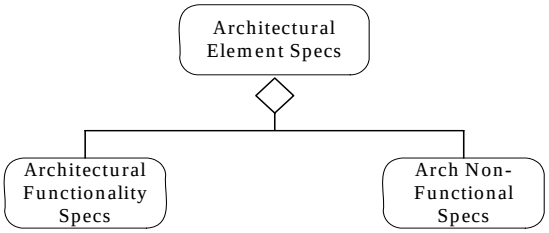

Figure 1: Architectural Element Specification

A key aspect of our model, the separation of the functional specs from the non functional specs, is elaborated in Figure 2 and Figure 3.

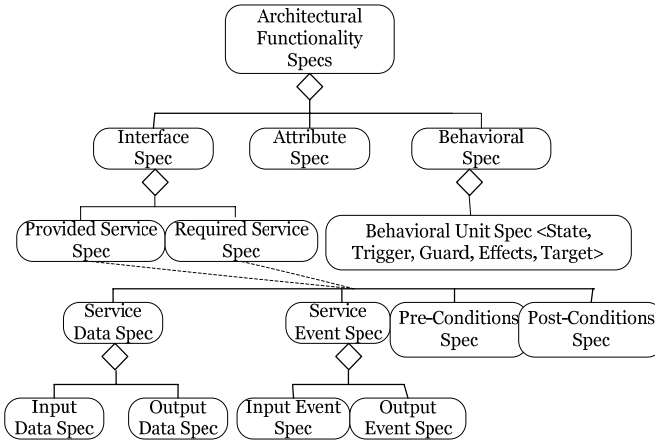

Figure 2: The Architectural Functionality Specs

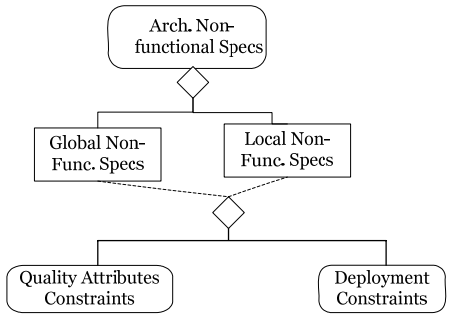

Figure 3: Arch Non-Functional Specifications

In Figure 2, the

- Interface Spec captures the interface information for the services provided.
- Attribute Spec captures the domain data supported by the architectural element.

- Behavioral Spec captures the state transitions supported by the architectural element.

The Architectural Non-Functional Specs, shown in Figure 3, comprises of the Quality Attributes Constraints and Deployment Constraints. These are shown in Figure 4 and 5 respectively.

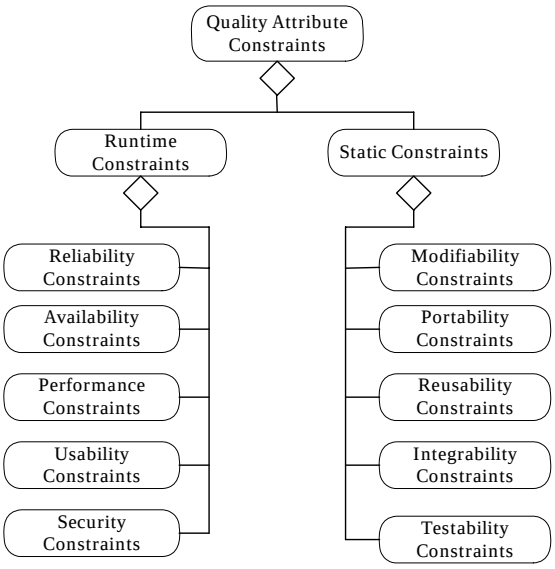

Figure 4: Quality Attribute Constraint

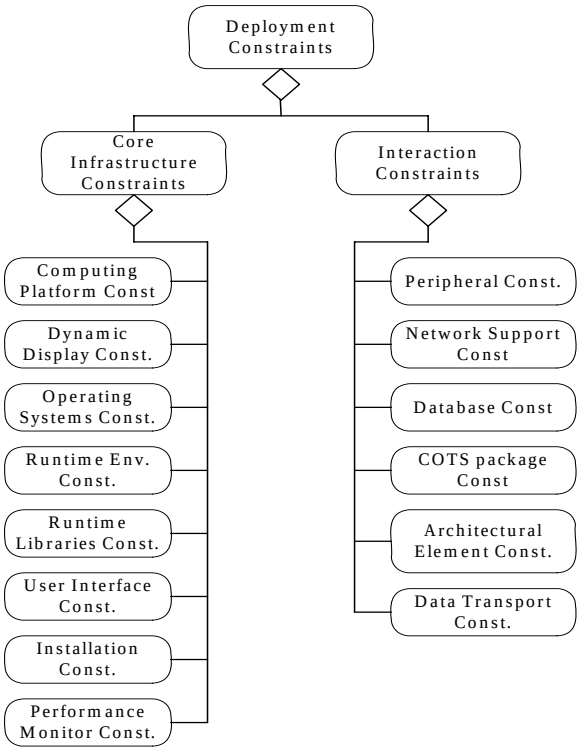

Figure 5: The Deployment Constraints

Each entity in the Quality Attribute constraints and the Deployment Constraints are further characterized by a set of attributes. Since the list of attributes is quite detailed, we do not elaborate them here.

With the above model for architectural element, we next explain the *asset* component specification. Asset components are the software components that have independent existence and are essentially the pre-built components using which a software architecture can be instantiated. The specification of the asset components are shown in Figure 6 below.

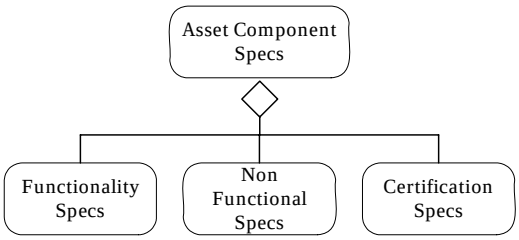

Figure 6: Asset Component Specification

We model the asset components using the same model as the architectural elements so that the asset components can be easily evaluated for an architecture instantiation. We partition our specification exactly as we partitioned our architectural model - Functionality Specs [details of which are similar to the Architectural Functionality Specs in Figure 2] and Non-Functional Specs [similar to the Architectural Non-Functional Specs elaborated in Figures 3, 4 and 5]. There is one additional element in the specification: a *certification spec*. When specifying asset components it is important to capture some notion of dependability of a software component. The Certification Spec captures information such as the maturity of the development process, product and process related metrics and verification and validation data from the component development. With the above model for specification of asset components, we start our exploration for identifying attributes and algorithms necessary for doing architectural style based reasoning.

3. Feature Category Analysis

With the specification model in place, we analyze the various feature categories proposed by Shaw and Clements to ensure the information needed for architectural style discrimination is captured in our model and elaborate our approach for determining the feature categories for a given configuration of components. We start with the constituent elements of a configuration. Then we explore the Control Issues followed by Data Issues. Finally, we investigate the Control/Data interactions.

3.1. Constituent Elements

Components: From a review of the identified classifications, components have been classified into Stand-Alone Programs, Transducers, Procedures, Managers, Processes and Filters. Thus the components in the Pipes and Filter architectural style are Transducers whereas in the Batch Sequential style, the components are Stand Alone Programs. Hence the need for classifying the components during the specification process, as one of the component types mentioned above, becomes obvious. The *Component Type* attribute associated with the Asset Component Specs [Figure 6] captures whether a component is a Stand Alone Program, a Transducer, a Procedure, a Manager, a Process or a Filter. This piece of information will be captured when a component provider specifies a component using our model. The tool that is being built to facilitate the specification process will provide guidance for the component provider to classify the component accurately.

Connectors: Connectors are usually distributed over many system components and often do not correspond to discrete elements of a software system. The different types of connectors identified in the classification include static calls, dynamic calls, shared representations, remote procedure calls, message-passing protocols, data streams, ASCII stream, batch data, signals, transaction streams and direct data access. This information is captured by attributes in the Interaction Constraints of the Deployment Constraints explained in Figure 5. Thus during the specification process, the *Connector Type* attribute of the Interaction Constraints will capture the connector used by the component as one of the different types of connector identified in the Shaw Clements classification.

Though the components and connectors are the primary discriminators among styles, identifying the components and connectors often do not uniquely identify the style. Data and control issues and their interactions affect style distinctions. Hence we next consider the Control Issues

3.2. Control Factors

The Control Factors help understand the temporal flow of control between the various components in a configuration. The feature based classification focuses on Topology, Synchronicity and Binding Time.

Topology: Topology is the geometric form of the control flow of a system. The identified control topologies are Linear, Acyclic, Arbitrary, Hierarchical and Star. For example a Batch Sequential data flow architecture has a Linear control topology while a data centered Blackboard style has a Star topology. The information for determining the topology of a system configuration is captured in the Asset Functionality Specs [analogous to the Architectural Functionality Specs for architectural elements elaborated in Figure 2]. Below we develop the algorithm for determining the control topology for a set of co-operating components in a configuration.

The initial selection of the set of components for the configuration is done based on the usage scenario or use case specified by the System Architect that needs to be satisfied by the target configuration. For specifying a scenario, the Architect selects services from the Architectural Functionality Specs [Figure 2] of the application domain. Note that during the specification process for asset components, we capture the services in the architectural component that the component supports in the Provided Service Spec of the asset component [analogous to the Provided Service Spec for architectural elements elaborated in Figure 2]. Thus, we can identify the 'best-fit' components "registered" (i.e. supports the service specified in the architectural element) to the services in the scenario by searching the component repository for the component with the highest value of the Service Compliance Metrics [4] (the details of the metrics defined as part of this research is excluded from here due to constraints of space). Similarly, we can identify the set of components that are needed to satisfy all the services for the System Architect's use case. For services for which no asset components can be found in the repository, notional components will be recommended.

Next we explain the algorithm for determining the control topology. From step 1 to step 7, we build the Control Flow List (CF List) while in steps 8 to 12, we identify the topology from the CF List. The CF List is an ordered list of components for execution of the scenario.

Step 1: Select service from the service list of the Scenario.

Step 2: If the selected service is the last service in the scenario, go to Step 8.

Step 3: Pick a component from the repository that is registered to the selected service and has the highest value of Service Compliance Metric [4].

Step 4: Add the component to the Control Flow (CF) List.

Step 5: For all the events in the Input Event Specs of the service delivered by asset component identified in Step 3, identify the asset components that *generate* the corresponding events (captured in the Output Event Specs). If the identified list of components is not already in the CF List, add the components to the CF

List *before* the component under consideration. The ordering of the event generators is done based on the pre-condition and post-condition dependencies among themselves.

Step 6: For all the events in the Output Event Specs of the service delivered by asset component identified in Step 3, identify the asset components that *consume* the corresponding events (captured in the Input Event Specs). Add the components to the CF List *after* the component under consideration. Again, the ordering of the event consumers is done based on the pre-condition and post-condition dependencies among themselves.

Step 7: Select next service from Scenario and go to Step 2.

Step 8: If all components occur only once in the CF List, Control Topology is *Linear*. Exit program.

Step 9: If the components in the CF List follow a tree-pattern, the Control Topology is *Hierarchical*. Exit program.

Step 10: If the components in the CF List follow a 'hub-and-spoke' pattern, the Control Topology is *Star*. Exit program.

Step 11: If the first component in the CF List is different from the last, the Control Topology is *Acyclic*. Exit program.

Step 12: The Control Topology is *Arbitrary.*

Synchronicity: Synchronicity is the nature of the dependence of the component's action upon each other's control state. Shaw and Clements have classified synchronicity into Batch Sequential, Synchronous, Asynchronous and Opportunistic. We leverage the Control Flow List developed for determining the control topology for determining the synchronicity of the set of components.

The determination of synchronicity is explained by a 4 step process.

Step 1: In the Control Flow List, if the output events of one component are the same as that of the input-events of the next component, the synchronicity is Sequential

Step 2: If at any point while traversing the Control Flow List, the list of output events of all preceding components exactly match the input events of the next component, the synchronicity is Synchronous

Step 3: In the Control Flow List, if the input events for all components corresponds to only output events of services supported by the same component and does not match the output events generated by services of any other component, the synchronicity is Opportunistic. Examples of this are autonomous agents that work completely independently from each other in parallel.

Step 4: If the synchronicity of a configuration couldn't be determined by any of the three previous steps, the synchronicity is Asynchronous.

Though the usage of events to determine the control flow may not seem intuitive, the generation and consumption of events are used here, as an indicator of passage of control from one component to another.

Binding Time: Binding time is the time of establishing of the identity of a collaborating component for transfer of control. Typical control transfers are determined at program-write time, compile time, or invocation time. Given our level of treatment of components, at this time we do not think that the Binding Time can be identified from the component interaction.

3.3. Data Factors

Data factors investigate the movement of data in the system. It focuses on the topology of the data movement, the continuity of data flow, the mode and the binding time. In this section we elaborate our approach for determining the data topology and the data continuity

Topology: Data topology explores a system's data flow graph, the different classifications being the same as those for the control topology, namely Linear, Acyclic, Arbitrary, Hierarchical and Star. Examples of the Star topology are the Blackboard and the Repository architectural style while the Batch Sequential and Pipe and Filter architectural styles represent a Linear data topology. A Hierarchical data topology is demonstrated by the Layered architectural style.

We can derive the data topology for the collaborating set of components using the Input and Output Data Specs associated with the Service Data Spec for the selected asset components. The derivation of the data topology is explained below.

Just as in the Control Topology determination, we use the System Architect's scenario to determine the Data Topology. Steps 1 to 7 builds the Data Flow List (DF List) which is analogous to the Control Flow List used for determining the Control topology. The subsequent steps help with the classification.

Step 1: Select service from service list of Scenario.

Step 2: If the selected service is the last service in the scenario go to Step 8.

Step 3: Pick an asset component from the repository that is registered to the selected service and has the highest value of the Service Compliance Metric [4].

Step 4: Add the asset component to the Data Flow (DF) List.

Step 5: Build a list of data entities referred to by the *Input Data Spec* for the service in the selected asset component.

Step 6: For each data entity in the list, find the asset components which generate the data element (captured in the Output Data Specs). If the component is different

from the one being considered, add it to the DF List *before* the component. The ordering of the data generators is done based on the pre-condition and post-condition dependencies among themselves.
Step 7: Select next service from Scenario and goto Step 2
Step 8: If all components occur only once in the DF List, then Data Topology is *Linear*. Exit program.
Step 9: If the components in the DF List follow a tree-pattern, the Data Topology is *Hierarchical*. Exit program.
Step 10: If the components in the DF List follow a 'hub-and-spoke' pattern, the Data Topology is *Star*. Exit program.
Step 11: If the first component in the DF List is different from the last, the Data Topology is *Acyclic*. Exit program.
Step 12: The Data Topology is *Arbitrary*

With the algorithm mentioned above, the data topology of most configurations can be determined. The main distinction between the approaches for determining the control topology and the data topology lies in the fact that for the control topology we need to identify all the asset components that generate the input events for a service as well as all the asset components that consume the output events of a service, and include them in the configuration. This is because if any event is not satisfied or consumed, the overall system may not perform to specifications. This is not true for the determination of the Data topology. For the Data Topology, we need to ensure that we include only the asset components that generate or produce the data that is needed by the service in the Architect's scenario. Without all the data elements, the desired service may not function satisfactorily. However it is not necessary to ensure that the output data generated by the service in the usage scenario gets consumed, unlike the output events for the control topology.

Continuity: Continuity is a measure of the flow of data through the system. 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. The further categorization of data continuity into high volume and low volume will not be used for our discrimination, as the *high* and *low* categorization seems too subjective and does not lend themselves to any objective measurement.

We propose the following algorithm for determining whether data continuity is *continuous* or *sporadic*. For all the services in the scenario (except the first and last in the DF List), if the asset component identified for supporting the scenario, requires a set of input data for executing the service, and generates output data as a result of executing the service, we call the system of components continuous, else we call the system sporadic. If there is generation and consumption of data at every service it is likely that the data continuity is continuous. Note that the first and last services in the DF List are not considered, because the first service not requiring any input data and the last service not generating any output data is a plausible deviation from the necessity of requiring input data and generating output data for the services in the scenario.

Mode: Data Mode refers to how the data is made available throughout the system. The identified modes include *passed* (for an object system), *shared* (for all data shared systems), *copy-out-copy-in*, *broadcast*, and *multicast*. Given our level of reasoning for the components, we do not use mode for our style distinction.

Binding Time: Analogous to the binding time for *Control Factors*, binding time for data issues is the discrimination on the time when the identity of a partner in a transfer of control is identified. Just as the binding time for control issues, binding time for control issues is not used for our classification.

3.4. Control/Data Interactions

Control/Data Interaction describes the relationship between the data and control factors.

Shape: The Shape for Control & Data interaction is an indicator of whether the control and data topologies are similar. If they are, the topologies are said to be Isomorphic. A number of architectural styles have their data and control topologies isomorphic, examples include Batch Sequential, Data Flow Network and Call Based Client Server. Some styles are not isomorphic e.g. Blackboard and the Main Program-Subroutine.

If the control and data topologies identified using the algorithms developed earlier are the same, we determine the shape of the control and data interactions to be isomorphic.

Directionality: Directionality is an indicator of whether the direction of flow is the same for the control and data for isomorphic configurations, or not. Directionality is irrelevant for non-isomorphic data and control topologies. We do not consider Directionality for our classification.

This concludes our feature category analysis. With the approach defined for determining each of the feature category attributes for a configuration of components, we would be able to perform analysis for a configuration's compliance to an architectural style.

4. Architectural Style Determination

Based on the feature category attributes determined in the previous section, we can predict the emergent architectural styles. We represent the value of the different feature category attributes and the corresponding architectural styles in a table format. This table was developed by Shaw and Clements as part of their approach for classifying architectural styles.

With the knowledge captured in Table 1 we determine the architectural style that the set of components identified from the usage scenario conforms to. The prediction is based on the values of the feature category attributes determined using the approaches developed in Section 3.

The process for predicting the emergent architectural style is outlined below.

Step 1: The System Architect specifies a use case/scenario for which a software configuration needs to be built from the services specified in the Architectural Functionality Specs.

Step 2: For each service in the use case, identify the best fit candidate from the component repository i.e. the component with the highest value of the Service Compliance Metric [4] and build the Base Component List.

Step 3: For each component in the *Base Component List*, make note of its *Component Type* Attribute. If all the components are not of the same type, we consider the component type of the set of components to be the one that is predominant.

Step 4: For each component in the *Base Component List*, make note of the *Connector Type* attribute in the Data Transport Spec. If all the connectors are not of the same type, we consider the connector type of the configuration to be the one that is predominant.

Step 5: Determine the Control Topology of the set of components by developing the Control Flow List (refer Section 3.2).

Step 6: Determine the Control Synchronicity of the configuration of the components (refer Section 3.2).

Step 7: Determine the Data Topology of the configuration by developing the Data Flow List (refer Section 3.3).

Step 8: Determine the Data Continuity of the configuration (refer Section 3.3).

Step 9: Determine whether the Control and Data topologies are isomorphic (refer Section 3.4).

Step 10: From the feature category attributes derived from Step 3 to Step 9, we reference Table 1 to determine the architectural style of the set of components. If no clear conclusion can be drawn, we try to determine the most probable architectural style by considering the maximum number of feature category attributes that can be used in making a prediction that is consistent with the classification shown in Table 1.

The Conformance Confidence Index (CCI) described below provides an objective measure of how close a configuration of components corresponds to a given style. Higher the value of CCI, the more compliant is the configuration to the corresponding architectural style. CCI for a given style, s, is calculated as in below

$$CCI = \sum_{fc \in FCA(s)} \frac{w_{fc} * V_{fc}}{| FCA(s) |}$$

Where

- *FCA(s):* The set of feature category attributes relevant for a given style s. In our case FCA(s) = [Component, Connector, Control Topology, Synchronicity, Data Topology, Data Continuity, Isomorphic Shapes] for all styles per the Shaw Clements classification. The values for the different elements in the set FCA(s) are determined by performing the feature category analysis outlined in Section 3.
- w_{fc} = the weight of the feature category attribute in the determination of the style. This factor can be ignored if empirical analysis shows that all the feature category attributes have equal weighting. If they are found relevant (as likely they will be), the values have to be determined individually for each style. We have not determined the value of w_{fc} for the different styles at this time.
- V_{fc} = 1 if our approach reveals that the corresponding feature category for a configuration matches the Shaw Clements classification for the given style, 0 otherwise.

In all likelihood it is the value of CCI that will guide Architects to the emergent architectural style as a perfect match of all feature category attributes is quite improbable for real-life systems.

It is worth noting that identifying the architectural style or determining the degree of compliance to a given style is not the goal in itself. The real value of predicting the architectural style lies in the fact that using this information we will be able to predict the quality attributes of the emergent system.

If it is determined from our architectural level analysis that the emerging style conforms to a pipe and filter system, the Architect would be able to deduce certain characteristics that the system would likely demonstrate upon deployment. In such a case, it can be inferred that there would be no complex interactions to manage and that system maintenance would be easy. Further, if required, the system can be made parallel or distributed to enhance its performance. On the

downside, the system would likely not be able to support a highly interactive use case and overall performance may be poor because of the batch model of execution.

Similarly if the emerging style is determined to be the Layered architectural style, the Architect would be able to infer that the system in general would be modifiable and portable. A high compliance index, CCI, with a value less than one, will imply that some layer bridging is likely happening, which may affect the goals of modifiability and portability.

Style	**Constituent Parts**		**Control Issues**		**Data Issues**		**Control/Data Interaction**
	Components	*Connectors*	*Topology*	*Synchronicity*	*Topology*	*Continuity*	*Isomorphic Shapes*
Data Flow Architectural Styles							
Batch Sequential	Stand-alone programs	Batch data	Linear	Sequential	Linear	Sporadic	Yes
Data-Flow Network	Transducers	Data Stream	Arbitrary	Asynchronous	Arbitrary	Continuous	Yes
Pipes and Filter	Transducers	Data Stream	Linear	Asynchronous	Linear	Continuous	Yes
Call and Return							
Main Program/ Subroutines	Procedure	Procedure Calls	Hierarchical	Sequential	Arbitrary	Sporadic	No
Abstract Data Types	Managers	Static Calls	Arbitrary	Sequential	Arbitrary	Sporadic	Yes
Objects	Managers	Dynamic Calls	Arbitrary	Sequential	Arbitrary	Sporadic	Yes
Call based Client Server	Programs	Calls or RPC	Star	Synchronous	Star	Sporadic	Yes
Layered	-	-	Hierarchical	Any	Hierarchical	Sporadic	Often
Independent Components							
Event Systems	Processes	Signals	Arbitrary	Asynchronous	Arbitrary	Sporadic	Yes
Communicating Processes	Processes	Message Protocols	Arbitrary	Any but Sequential	Arbitrary	Sporadic	Yes
Data Centered							
Repository	Memory, Computations	Queries	Star	Asynchronous	Star	Sporadic	Yes
Blackboard	Memory, Components	Direct Access	Star	Asynchronous	Star	Sporadic	No

Table 1: Architectural Style Classification

5. Validation of Approach

We validated our proposal using a real life system that was available to us. The system is involved in a complex manufacturing process and essentially transforms data originating from the process, computes a number of indicators, generates reports and provides an interface for the end user to view the computed results and perform ad-hoc analysis on the data.

The approach we took for the case study was to have the key architect of the project specify the components of the system using our specification model and then follow our algorithms for determining the architectural style of the system. No inputs were provided to the subject except for clarifications on the model and the algorithm details.

The system was partitioned into 6 architectural elements with 35 services and 6 asset components. Each of the six asset components mapped uniquely to the six architectural components. This was primarily because we were reengineering the architecture from the deployed system and the architect's thought process was influenced by the system that he was intimately familiar with. The services were specified at a significant level of detail though some clarifications had to be provided with respect to specification of events. The Behavioral Specification however was not filled out to any significant degree of detail. Also the architectural non-functional specifications captured information only to the extent that was available in the project documents.

The architect was first asked to define a usage scenario for performing the architectural style analysis.

He picked the most common use case for the system. With the software system specified using our abstract model and the key use case identified the study then got into the feature category analysis. We had to make sure that the use case was defined in terms of the services that were already in the specified architecture. The architect initially had a lot of questions on the classification of the components and connectors as it wasn't obvious what the appropriate component type and connector types should be. After some clarifications were provided the architect classified four of the six components as Transducers, one as a Manager and the sixth as a Filter. For the classification of the connectors, the architect was not sure whether to classify them as Data Stream or an ASCII stream and eventually decided on the ASCII stream for all the 5 connectors. Development of the Control Flow List was in fact quite straightforward as the event relationships were quite simple. The Control topology was determined to be Linear. Determining the Synchronicity was not as straightforward though – by the algorithm the Synchronicity came out to be Asynchronous though the architect grappled with the true implication of synchronicity and it seemed that an event based approach for determining synchronicity is counter-intuitive. The Data Topology turned out to be Linear just like the Control Topology and the Data Continuity classification was Continuous. When we matched up the findings against Table 1, the architectural style was determined to be the Pipe and Filter style. This result corroborated the architect's perception of the system.

There were several interesting lessons from this case study. Even though the final results were satisfactory, it was obvious that for a more rigorous assessment of our proposal, the same individual/team should not be involved in the definition of the architecture and the specification of the asset components and architectural style analysis. Further it became evident that more clarity needs to be provided for facilitating the accurate classification of components and connectors. It seems that in all likelihood, for complex systems, style determination would never fit exactly into the style classifications proposed by Shaw and Clements and that our Conformance Confidence Index (CCI) would play an important role in providing the System Architect with insights into the degree of compliance with a given style.

One of the drawbacks of the validation done so far is - we have not exercised our approach using a system that is yet to be built. The main reason for that has been the lack of opportunity for this research team to be involved in a component based development project during its inception phase.

7. Related Work

In 1989 Perry and Wolf ([5], published in 1992 [1]) introduced the notion of software architectural styles and demonstrated the concept using the "multi phased architectural style" of a compiler. Garlan and Shaw [6] categorized several architectural abstractions and demonstrated their applicability in real-life systems. Then in 1997, Shaw and Clements [3] proposed a feature based classification of architectural styles. These efforts firmly established the importance of architecture styles in software architecture. Along the way, different research efforts explored formal approaches for rigorously defining styles with the intent of enabling systematic analysis. Abowd et al. [2] formalized style descriptions and proposed a framework for their codification using Z. In 1998 le Metayer [7] used graph grammar for describing architectural styles and recently Bernardo et al. [8] used PI-calculus for the same purpose. Communication topologies in the context of styles have been explored by the Alfa framework of Mehta and Medvidovic [9]. Alfa enables the construction of style based software architectures from architectural primitives defined along five orthogonal characteristics of Data, Structure, Interaction, Behavior and Topology.

Our work is essentially based on the style classification proposed by Shaw and Clements, with our contribution being the demonstration of the applicability of such classifications in predicting emergent styles during component based software construction. It is also possible to validate style conformance with our approach.

7. Conclusion

In this paper we have developed an approach for reasoning about architectural styles using component specification and a use case scenario which the System Architect's desires to satisfy by using a configuration of components. We have also shared the findings from the case study we conducted.

With our style prediction proposal, not only will a System Architect have the ability to evaluate several deployment options but will also have the ability to get a sense of the quality attributes of the final system before actually building it. This could prove to be an extremely valuable way of assessing the final system behavior a-priori.

Given that we can determine the emerging stylistic characteristics of a configuration (whether global or "regional") and determine how close it comes to satisfying a particular architectural style, we can also

use our approach to determine the conformance of that configuration to particular style. This will be particularly useful during the evolution of a system to detect either architectural drift, or even architectural erosion [1, 5, and 10].

We envision this research to evolve, resulting in tools that would make the System Architect's job easier and more efficient. We have not come across any research so far that has attempted to bridge the gap between Software Architecture and Component Based Software Engineering, and in that sense we consider our work to be novel.

8. References

[1] Perry, D. E., Wolf, A. L., "Foundations for the Study of Software Architectures", ACM Software Engineering Notes, 17, 4, October 1992, 40-52

[2] Abowd, G., Allen, R., Garlan, G., "Using Style to Understand Descriptions of Software Architecture", Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, 1993, 9-20

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

[4] Bhattacharya, S., Perry, D. E., "Contextual Reusability Metrics for Event-Based Architectures", Proceedings of the 4th International Symposium on Experimental Software Engineering (ISESE), Australia, November 2005

[5] Perry, D. E., Wolf, A. L., "Software Architecture", http://www.ece.utexas.edu/~perry/work/papers/swa89.pdf, August1989

[6] Mary, S., Garlan, D.,"Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall, 1996

[7] le Metayer, D., "Describing Architectural Styles using Graph Grammars", IEEE Transactions of Software Engineering, 24, 1998, 521-533

[8] Bernardo, M., Ciancarini, P., Donatiello, L., "Architecting Families of Software Systems with Process Algebra", ACM Transactions of Software Engineering and Methodology, 11, 2002, 386-426.

[9] Mehta, N. R., Medvidovic, N., "Composing Architectural Styles from Architectural Primitives", Proceedings of the Joint 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Helsinki, Finland, September 2003

[10] Bhattacharya, S., Perry, D. E., "Architecture Assessment Model for System Evolution", Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), Mumbai, India, January 2007

[11] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns Elements of Reusable Object-Oriented Software", Addison-Wesley, 2002

[12] Perry, D. E., "Generic Descriptions for Product Line Architectures", *ARES II Product Line Architecture Workshop*, Los Palmos, Gran Canaria, Spain, February 1998

[13] Habermann, A. N., Perry, D. E., "Well Formed System Composition", Carnegie-Mellon University, Technical Report CMU-CS-80-117, March 1980

[14] Perry, D. E., "The Inscape Environment: A Practical Approach to Specifications in Large-Scale Software Development. A Position Paper", January 1990

[15] Bhattacharya, S. "Specification and Evaluation of Technology Components to Enhance Reuse", Masters Thesis, The University of Texas at Austin, July 2000

[16] Bhattacharya, S., Perry, D. E., "Predicting Architectural Styles from Component Specifications - Extended Abstract", Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture, Pittsburgh, PA, November 2005

[17] Hirsch, D., Inverardi, P., Montanari, U., "Graph Grammars and Constraint Solving for Software Architecture Styles", Proceedings of the 3rd International Software Architecture Workshop (ISAW3), 69-72, November 1998

[18] Wermelinger, M., "Towards a Chemical Model for Software Architecture Reconfiguration", IEEE Proceedings - Software, 145(5):130-136, October 1998

[19] Medvidovic, N., Egyed, A., Grünbacher, P., "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery", Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures (STRAW), co-located with ICSE 2003, Portland, Oregon, May 2003

[20] Yakimovich, D., Bieman, J. M., Basili, V. R., "Software Architecture Classification for Estimating the Cost of COTS Integration", ICSE 1999, 296-302