Predicting Architectural Styles from Component Specification

Sutirtha Bhattacharya
PTD Automation
Intel Corporation
Hillsboro, OR – 97124
sutirtha.bhattacharya@intel.com

Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
Austin, TX 78712
perry@ece.utexas.edu

Abstract*

Software Product Lines (SPL), Component Based Software Engineering (CBSE) and Customer 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, a System Integrator could make necessary changes to ensure that the quality attributes dictated by the system requirements were satisfied before the actual system was deployed and tested. In this paper we propose a model for predicting architectural styles based on use cases that need to be met by a system configuration. Moreover, our technique can be used to determine stylistic conformance and hence indicate the presence or absence of architectural drift

Keywords

Component Based Software Engineering, Architectural Style, System Composition, Reuse

1. Introduction and Scope

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 Issues i.e. the flow of control among components
- Data Issues i.e. details on how data is processed
- Control/Data Interaction i.e. the relation between control and data
- Type of Reasoning: Analysis techniques applicable to the style

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 (CBSE) 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 components thus reducing software costs and delivery time. The focus of this area has mostly been directed towards understanding and resolving integration issues between the various components and establishing a common vocabulary for facilitating the integration. The focus of Software Architecture, on the other hand, is concerned with 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 what the component characteristics that need to be understood are, to enforce an architectural style by choice. Since different architectural styles support distinct sets of quality attributes, the benefit of

* This research is supported in part by NSF CISE grant IIS-0438967. Please note that any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

evaluating components for suitability to an architectural style is obvious, as the quality attributes for a system are often dictated by the system requirements. 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 a-priori. We analyze the different feature categories proposed by Shaw and Clements and identify the component attributes that would help determine the architectural style, in 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 architectural style determination. Section 5 outlines the proposal for the validation of our approach 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 domain have been specified using our asset specification model (briefly explained here) against our architectural specification. A System Integrator identifies a deployment use-case (made up 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 System Integrator 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 integrator'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 our specification approach which will be leveraged for the style related reasoning.

Our specification model captures the architecture of a certain domain in terms of architectural elements. These elements are essentially the components and connectors that are relevant for the application domain and enable functional partitioning as well as introduce the notion of object orientation. Figure 1 shows the structure of architectural element specification.

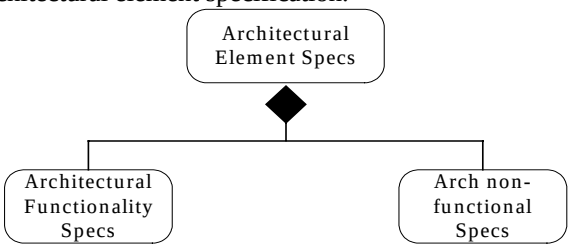

Figure 1: Architectural Element Specification

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.

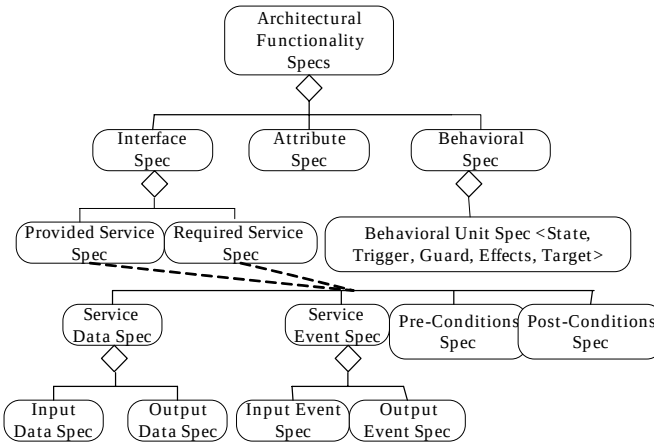

Figure 2: The Architectural Functionality Specs

In the above diagram, the

- Interface Spec captures the interface information for the services provided by the architectural element
- Attribute Map captures the domain data supported by the architectural element
- Behavioral Map captures the state transitions supported

The Architectural non-functional specs are specified as in below

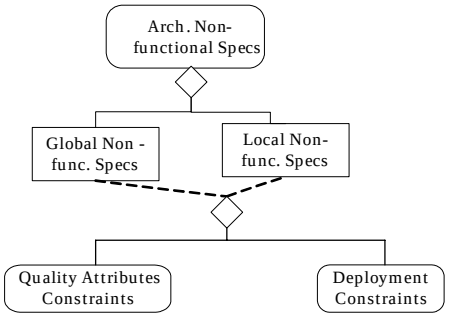

Figure 3: Architectural Non-Functional Specifications

For the architectural non-functional Specs, the Quality Attributes Constraints are shown in Figure 4 while the Deployment Constraints are shown in Figure 5.

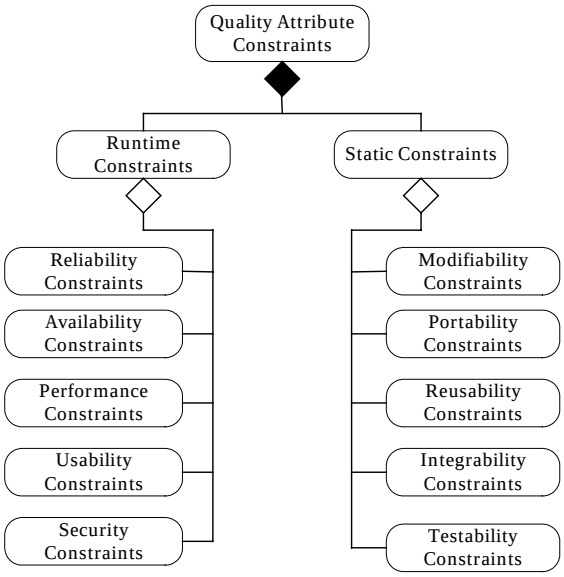

Figure 4: Quality Attribute Constraint

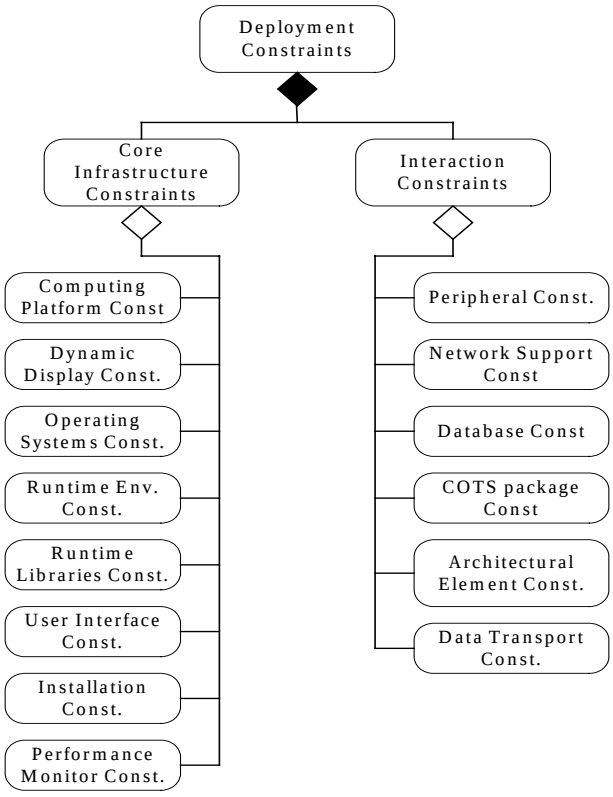

Figure 4: The Deployment Constraints

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 that we mentioned at the beginning of this paper. The specification of the asset components are shown in Figure 6

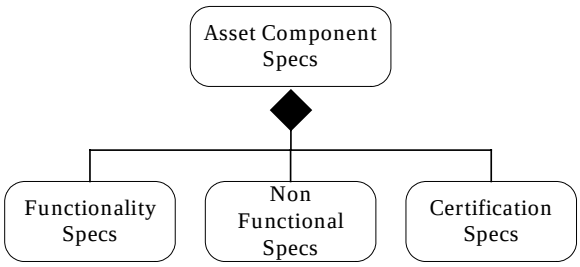

Figure 6: Asset Component Specification

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 about 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 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 reasoning 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.

Constituent Elements

Components: From a study of the identified classifications, components are classified into stand-alone programs, Transducers, Procedures, Managers, Processes and filters. Thus the components in the Pipes and Filter architectural Styles are transducers whereas in the batch sequential architectural 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 procedures, a managers, a processes 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 4. 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

Control Factors

The Control Factors helps 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 integrator that needs to be satisfied by the target configuration. For specifying a scenario, the system integrator 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 satisfies 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 of the scenario by searching the component repository for the component with the highest value of the Service Compliance Metrics [16] (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 Integrator'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. 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
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 are 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. The ordering of the event consumers are 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, then 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*

The determination whether the ordered list of components in the CF list follow a tree-shaped pattern or a hub and spoke pattern are ignored for now as implementation details.

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.

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.

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 architecture.

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

Just as in the Control Topology determination, we use the system integrator's scenario/use-case 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
Step 4: Add the asset component to the Data Flow (DF) List
Step 5: Build a list of data elements referred to by the *Input Data Spec* for the Service in the selected asset component.
Step 6: For each data element 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 are 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*
Step 13: Exit Program

As in the Control Topology determination, whether the ordered list of components in the DF List follow a 'star-shaped' pattern or a 'hub and spoke' pattern is ignored as implementation details for now. 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 system integrators 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 them to any objective measurement.
We propose the following algorithm for determining whether data continuity is *continuous* or *sporadic*.

For all service in the scenario (except the first and last), 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. Note that the first and last services 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.

Control/Data Interaction

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 architectural style. Some styles are not isomorphic. This includes the Blackboard architectural style and the main program-subroutine call & return architectural style.

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 component configuration's compliance to an architectural style.

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	Possibly
Data Centered							
Repository	Memory, Computations	Queries	Star	Asynchronous	Star	Sporadic	Possibly
Blackboard	Memory, Components	Direct Access	Star	Asynchronous	Star	Sporadic	No

Table 1: Architectural Style Classification

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 style

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

The step-by-step process for predicting the emergent architectural style is outlined below:

Step 1: The System Integrator 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, we identify the best fit candidate from the component repository i.e. the component with the highest value of the Service Compliance Metric [25] and build the Base Component List.

Step 3: For each component in the *Base Component List*, we make a 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 most common.

Step 4: For each component in the *Base Component List*, we make a 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 of components to be the one that is most common.

Step 5: We determine the Control Topology of the set of components by developing the Control Flow List (details outlined in Section 3)

Step 6: We determine the Control Synchronicity of the configuration of the components (details outlined in Section 3)
Step 7: The Data Topology of the configuration of components is determined by developing the Data Flow List (details outlined in Section 3)
Step 8: The Data Continuity of the configuration is determined (details outlined in Section 3)
Step 9: We determine whether the Control and Data Topologies are isomorphic (details outlined in Section 3)
Step 10: From the feature category attributes derived in Steps 3 to Step 10, we reference the 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 the table. 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

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

Where

- *FCA(s):* The set of feature category attributes relevant for a given style s. In our case FCA(s) = [Components, Connectors, Control Topology, Synchronicity, Data Topology, Data Continuity, Isomorphic Shapes] for all styles by the Shaw Clements classification.
- 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
- 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

5. Validation of Approach

Having developed the approach for predicting the architectural style for a configuration of components, in this section we explore methods for validating our approach.

Popular Textbooks on metrics [5] recommends separation of concerns about the two typical types of systems for which metrics are often used

- Measures or measurement systems, which are used to assess an existing entity by characterizing one or more of its attributes, numerically
- Prediction Systems, which are used to predict some attribute of a future entity involving mathematical models and associated prediction procedures.

Our proposal in this paper regarding the prediction of architectural styles fall in the second category as what we are doing is essentially trying to predict what the architectural style of a configuration of components is likely to be. Therefore we go down the path of Prediction Systems and try to find the validation approach that will be most suitable for our purpose.

Empirical investigation is the typical tool that is used for validating Prediction systems and the investigative techniques used commonly involve surveys, case studies or formal experiments.

Surveys are typically retrospective studies of a situation for documenting relationships and outcomes. They are conducted after the occurrence of an event. Used most frequently for the social sciences, surveys have also been used extensively for Software Engineering for determining trends and relationships. Given the retrospective nature of surveys, we feel that it would not be the correct tool for validating the correctness of the architectural style predicted by our approach.

The other two research techniques, case studies and formal experiments are usually not retrospective. In a case study, we identify the key variables that may affect the outcome of an activity and then document the activity in terms of its inputs, constraints, resources and output whereas in a formal experiment we identify the key variables and manipulate them to document their effects on the outcome. Both of these research techniques could be used to validate our approach. We next investigate whether case studies or formal experiments would be more suitable.

It is easy to identify the control variables, which would be the feature category attributes in our case. However, it is not very likely that we will have a lot of control over these variables because they would be dependant on the components that are present in the repository which have been specified using the specification technique mentioned in this paper. Given that, the feasibility of the formal experiment decreases slightly but given a large enough repository, the ability to control the variables may not be completely infeasible. The other key aspect to consider, as recommended by Fenton and Pfleeger [5], is the replicability of the various

control parameters. This in our case should be easy as the specification of the components that are to be used as a part of the experiments would be stored in the repository.

Therefore we propose a two fold validation, one using a case study which will be done in the short term and the second to be done in the long term. As a first phase of validation, we will run a case study on two well known architectural styles - the Pipes and Filter and the Blackboard architectural style and validate the steps and the conclusion with two expert architects at a minimum. Once the correctness of our approach is established by this first phase of validation, we propose to build a tool to enable automatic identification of the architectural style based on the usage scenario of the system integrator and the components registered to the repository. Using this tool we propose to run various formal experiments to ensure that changes to the control variables i.e. the feature category attributes, change the recommendation of the architectural style. As an additional measure we will run the recommendations of the tool by two (at a minimum) expert architects to ensure the correctness of our approach.

We will also empirically validate the weights to be used for our preliminary proposal of the Conformance Confidence Index.

6. Related Work

In 1989 Perry and Wolf ([20], 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 [published in 9] 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 [17] used graph grammar for describing architectural styles and recently Bernardo et al. [19] 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 [18]. Alfa enables the construction of style based software architectures from architectural primitives defined along five orthogonal characteristics of: Data, Structure, Interaction, Behavior and Topolgy

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 the approach for reasoning about architectural styles using component specification and a use case scenario which the system integrator desires to satisfy by using a configuration of components. We have also outlined the technique that we will use to validate it as part of our ongoing research.

With the proposed approach, not only will a system integrator 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 a system. 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, 20]

We envision this research to evolve, resulting in tools that would make the System Integrator’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.

7. 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, 6-13

[4] Bass, L., Clements, P., Kazman, R., “Software Architecture in Practice”, Addison Wesley, 1999

[5] N. E. Fenton, S. F. Pfleeger, “Software Metrics: A Rigorous and Practical Approach”, PWS Publishing

Company, 1997
[6] Poulin, J. S., “Measuring Software Reuse – Principles, Practices and Economic Models”, Addison-Wesley, 1996
[7] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns Elements of Reusable Object-Oriented Software”, Addison-Wesley, 2002
[8] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., “Pattern Oriented Software Architecture”, Wiley Series in Software Design patterns, 2001
[9] Mary, S., Garlan, D., “Software Architecture: Perspectives on an Emerging Discipline”. Prentice Hall, 1996
[10] Szyperski, C., “Component Software: Beyond Object Oriented Programming,” Addison-Wesley, 1999
[11] Bosch, J, “ Design & Use of Software Architectures: Adopting and evolving a product-line approach,” Addison-Wesley, 2000
[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.,. "Contextual Reusability Metrics for Event-Based Architectures". Submitted for publication, March 2005
[17] le Metayer, D., “Describing architectural styles using graph grammars”, IEEE Transactions of Software Engineering, 24, 1998, 521-533
[18] Mehta, N. R. and 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
[19] Bernardo, M., Ciancarini, P., Donatiello, L., “Architecting families of software systems with process algebra”, ACM Trannsactions of Software Engineering and Methodology, 11, 2002, 386-426.
[20] Perry, D. E., Wolf, A. L., “Software Architecture”, August 1989. http://www.ece.utexas.edu/~perry/work/papers/swa89.pdf