Perspective-based Architectural Approach for Dependable Systems

Sheldon X. Liang, J. Puett, Luqi
Software Engineering Automation Center
US Naval Postgraduate School
{xliang, jfpuett, luqi}@nps.navy.mil

Abstract

Explicitly architecting dependable systems inevitably involves establishing consensus among different stakeholders' concerns and then anchoring the design on architectural components that provide robustness. The goal is to architect evolvable systems upon which users can reasonably rely on receiving anticipated services. Unfortunately, there are few established approaches for rapidly prototyping architecture to identify dependable architectural components during the early stakeholder requirements resolution phases of software design. This paper presents a perspective-based architectural (PBA) approach process using rapid prototyping to build dependable architectures using compositional patterns. The approach is achieved through explicit architecting and system composition to provide a set of rules governing the system composition from coarser-grained dependable components. The approach provides a rationale for treating dependability as a set of semantic constraints localized on compositional patterns.

1. Introduction

Building dependability into the architectural design aims at attaining the benefits of reduced cost and increased quality. The central idea is that dependable architectures in large, complex, evolving systems will provide their users with a reasonable assurance that the system will deliver the services promised. Explicitly architecting such systems requires identifying and resolving different stakeholders' concerns. For instance, the architect may have to resolve the inherit conflicts between a user stakeholder that is concerned with achieving a particular computational requirement and an implementer stakeholder that may be concerned with achieving systematic long-term evolution of the system. Perspective-based architectural design [1-4] allows some resolution between these perspectives.

The difficulties in engineering software-intensive systems are further exacerbated by requirements uncertainty, dynamic organizational structures (and concerns), and the requirement for rapid application development. Engineering dependable systems involves three crucial aspects: 1) accurately identifying all customer requirements, 2) resolving customer requirement conflicts within the context of different customer perspectives, and 3) verifying that the resulting system satisfies customer intent (and if not, correcting the requirements and the system).

A number of techniques, frameworks, and approaches have emerged to address the problems in engineering software-intensive systems. Widely embraced efforts include rapid system prototypes [5-6], software architectures [7-11], and component techniques [12-14]; all of which focus on composing software systems from coarser-grained components. Rapid system prototyping is useful in effectively capturing and resolving uncertainty about requirements and providing computational visibility [6]. Component techniques assume a homogeneous architectural environment in which all components adhere to certain implementation constraints (e.g., design, packaging, and runtime constraints). They are unalterably associated with derivational implementation with little concern of the perspectives of the customer or architect [8-10]. Software architecture approaches typically separate computation (components) from interaction (connectors) in a system. However, the current level of understanding and support for connectors has been insufficient, so that connectors are often considered to be explicit at the level of architecture, but intangible in the system implementation [9-10]. Several sources have recommended the use of architecture views [1, 2]. Yet, while they provide guidance on how architecture should be represented, they generally do not provide a prototyping process for the early development of the artifacts that are used in that representation [3-4].

The rapid prototyping of architectural components shows promise in acquiring accurate and timely requirements and in establishing appropriate compartmentalization of functionality [2-4]. To reduce the amount of re-certification effort required after each requirement change, the approach presented in this paper helps to maintain the assurance of dependability as the system evolves by combining rapid prototyping with explicit architecting so that the system's architecture is based on properties that are invariant with respect to system requirements changes. This research integrates requirements validation techniques and stakeholder perspective resolution into a single model of explicit architectural composition.

2. Overview of the Approach

Fig. 1 depicts the PBA approach embodied in three perspectives: computational activity, compositional architecture and derivational implementation. Starting by rapid prototyping the user's informal needs, an initial prototyping model is created that represents the computational activity needed to implement the operational concept. Continued analysis and refinement of the prototype then derives the explicit architecture from which it is possible to extract valuable architectural properties. Compositional architecture is then explicitly built under the support of compositional patterns, and the generation of application framework is driven by both prototyping and architecting documentations. Next, PBA composers are applied to derive PBA components.

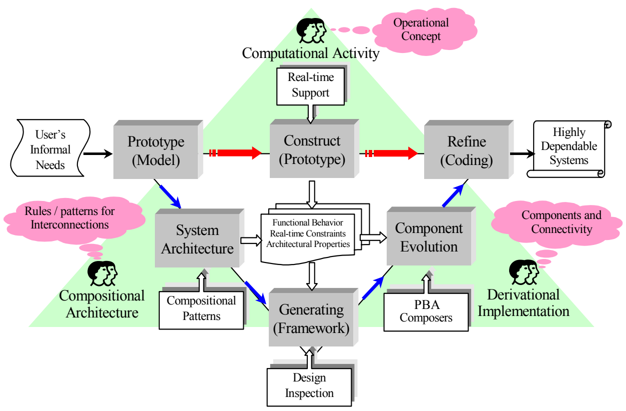

Fig 1 Synthesizing Approach

For each perspective design artifact, a computer-aided foundation is provided with significant formulated attributes enabling automated analysis, reasoning and code/framework generation. For instance, the computational activity captures the activities and information flows that will accomplish the operational concept (e.g., real-time support is the foundation for hard real-time systems [5]); the compositional architecture details what kinds of rules (patterns) are used to govern the interactions among components (e.g., compositional patterns [8,11] and design inspection [15-17] support semi-automated architecture generation); and the derivational implementation identifies physical components and connectivity that will be instantiated to carry out the computational activities (e.g., based on PBA composers [18]). Thus, compositional architecture bridges gaps between the computational and derivational artifacts (user and implementer perspectives).

3. Perspective-based Architecting

Central to the PBA approach, compositional patterns provide principles for guiding the design and evolution of system architecture and can be treated as architectural elements governing system composition from coarser-grained components. The transitional process is embodied in three perspective designs.

3.1 Computational activity

Computational activity accounts for the customer perspective concerns of computation and interconnection. This perspective addresses system requirements by capturing three kinds of formal arguments: components from which the system is built, interconnections enforcing interactions among them, and constraints on both components and interconnections:

$$\boldsymbol{P}_{\text{computation}} = [\boldsymbol{C}_c, \boldsymbol{I}, \boldsymbol{Ct}\,(\boldsymbol{C}_c, \boldsymbol{I})]$$

Where $\boldsymbol{C}_c$ is the set of conceptual components hierarchically decomposed, $\boldsymbol{I}$ is the set of interconnections among components, $\boldsymbol{Ct}\,(\boldsymbol{C}_c, \boldsymbol{I})$ is the set of constraints localized on components and their interconnections, respectively.

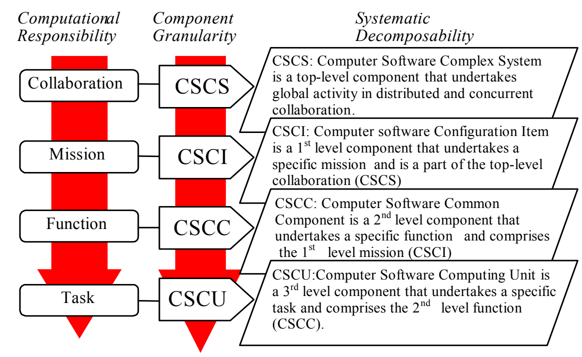

Fig. 2 Computational Responsibility and Properties

The constraints on components have properties of decomposability (representing the hierarchical level at which the constraint is implemented) and granularity (representing the logical packaged complexity of the component). Granularity is an important factor for constructing complex systems because well-grained components are helpful not only to increase productivity but also to improve understandability, reliability and maintainability. As illustrated in Fig. 2, a schema is introduced for PBA components, which identifies the granularity and decomposability of each level of computational responsibility.

3.2 Compositional architecture

Compositional architecture accounts for the architect's perspectives of explicit treatment of system composition and architecture with constraints localized on compositional patterns. Detailing what kinds of rules (patterns) are used to govern interactions among components and how quantitative constraints are associated with the patterns, this perspective addresses what kinds of interactions are applied among components and how to associate constraints with compositional patterns. This perspective is represented as follows:

$$\boldsymbol{P}_{\text{composition}} = [\boldsymbol{C}_c \Rightarrow \boldsymbol{R},\ \boldsymbol{R}_o \xrightarrow{S/P} \boldsymbol{R}_i,\ \boldsymbol{Ct}\,(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P})]$$

Where $C_c \Rightarrow R$ is the set of roles extracted from conceptual components. $R_o \text{—}^{S}/_{P}\text{→} R_i$ is the set of compositional patterns: R_o (output/producer) interacts with R_i (input/consumer) via architectural styles S while complying with communicatory protocols P. $Ct(R, S, P)$ is the set of constraints localized on roles, styles, and protocols, respectively.

Constraints on interactions further localized on architectural styles are embodied in such properties as composability and heterogeneity. Composability represents the hierarchical composition of architecture (i.e., an entire architecture becomes a single component in another larger architecture). Heterogeneity represents the diverse ways components interact with each other. Heterogeneity is inevitable in complex systems because diverse components or systems will have to work and interact together. In Fig. 3, a compositional coupling schema is introduced for PBA approach.

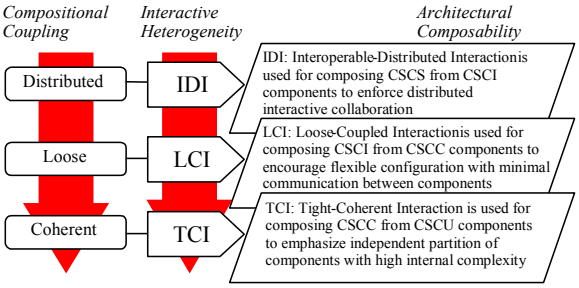

Fig. 3 Compositional Coupling and Properties

3.3 Derivational implementation

Derivational implementation accounts for the implementer's perspectives of component derivation and connectivity. This perspective addresses what kinds of components are needed to carry out computational activity, what connectivity is needed between the components and how to glue the components to specific roles. This perspective is represented as follows:

$$P_{\text{derivation}} = [R \supset C_p,\ (C_p \rightarrowtail R_o) \text{—}^{S}/_{P}\text{→} (R_i \mapsto C_p),\ Ct\,(C_p\, S,\ P)]$$

Where $R \supset C_p$ is the set of physical components derived from the associated role. $C_p \rightarrowtail R$ (its peer $R_i \mapsto C_p$) is the set of instantiated components that are glued to associated roles. $Ct(C_p, S, P)$ is the set of constraints localized on physical components, styles, and protocols, respectively.

Constraints on components are embodied in such properties as connectivity (representing the way components are derived from the related role) and evolvability (representing the evolution from roles to components). Interactive roles are represented as generalized role wrappers (GRWs) (an abstract class) to support component evolution through sub-typing and refinement. As illustrated in Fig. 4, the GRWs defined in PBA composers introduce derivational gluing to refer to connectivity and evolvability.

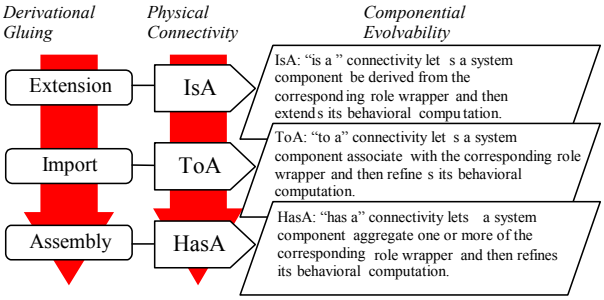

Fig. 4 Derivational Gluing and Properties

3.4 Automated transitional process

Starting with a prototyping model in the computational activity perspective, a transitional process is formed from computational activity, through compositional architecture, to derivational implementation. Two kinds of architectural elements evolve: PBA composers and PBA components. Under the support of automated software tools, two key mappings are used to bridge the gaps between perspectives: explicit architecting via compositional patterns and physical evolution via PBA composers. PBA approach is associated with support tools such as Prototyping Analyzer, Pattern Selector, Framework Generator, and Component Evolver [5,8]. Fig. 5 illustrates this transitional process.

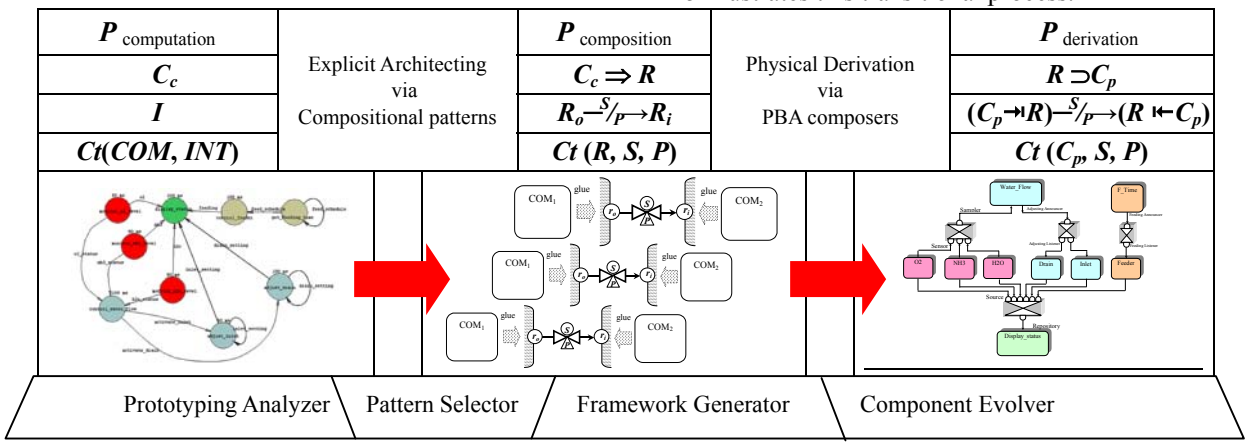

Fig. 5 Transitional Process between Architectural Perspectives

Explicit architecting of the computational activity starts with assigning components with specific roles. According to the architectural styles, related interactive roles and communicatory protocols can be determined so that suitable compositional patterns can be selected and applied to govern the interconnections among the roles. According to the assignment of which components play which specific roles, the components will be derived from the associated role facility. After being derived, the components will be instantiated and then glued to the associated roles by the PBA configuration.

A PBA composer is designed as a generic package-like architectural entity that includes two kinds of GRWs: one is for the "interactive producer" and the other is for the "interactive consumer." GRWs provide adherence to restricted, plug-compatible interfaces for interaction and provide the template of behavior that components are expected to refine. The physical connectivity between a component and a role is implemented by refining or overriding the restricted, plug-compatible interfaces defined by the GRW [11, 18].

4. Dependable Compositional patterns

Compositional patterns provide a set of rules that govern the interactions among components with localized constraints. They are characterized by three kinds of formulated arguments: interactive roles, architectural styles, and communicatory protocols.

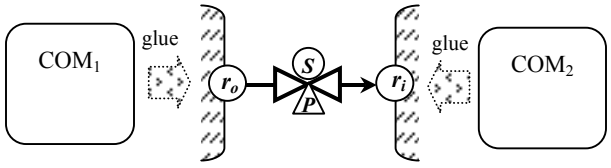

Fig. 6 Compositional pattern for interconnections

Fig. 6 depicts a compositional pattern. For a given interaction between two components (COM_1, COM_2), both are assigned to play specific roles $\mathbf{r}_o$ and $\mathbf{r}_i$ in the specific compositional pattern. An architectural style ***s*** specifies how $\mathbf{r}_o$ (output / producer) interacts with $\mathbf{r}_i$ (input / consumer), while communicatory protocol ***p*** builds a specific channel for message transportation during the interaction. More specifically, in order to construct the components as autonomous entities, roles in the compositional pattern are deputized for the components in dealing with interaction while the associated components are mainly concerned with their functionality (computation separated from interaction). The pattern also provides a means for gluing a specific component to a role.

By mathematically defining the compositional patterns, it is possible to translate, localize, and analyze them using automated CASE tools. Compositional patterns involve three sets: ***R*** representing interactive roles, ***S*** representing architectural styles, and ***P*** representing communicatory protocols. Examples include:

R = {	***S*** = {	***P*** = {
Caller, Definer,	Explicit-invocation,	Message-passing,
Announcer, Listener,	Implicit-invocation,	Event-broadcast
Outflow, Inflow,	Pipe-filter (Pipeline),	Data-stream,
Source, Repository,	Repository-centric,	Sampled-stream
Read, Writer, …	Blackboard, …	Shared-data, …
}	}	}

Regardless of any constraint, a composition is defined as an interaction between two roles (e.g., Caller and Definer) via an architectural style (e.g., explicit-invocation), while complying with a communicatory protocol (e.g., message-passing). So, the Cartesian product R x S x P x R enumerates all possible compositions C, represented as follows:

$$\boldsymbol{C}\,(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{\, \boldsymbol{r}_o \text{—}^{s}/_{p}\!\rightarrow \boldsymbol{r}_i \mid \mathbf{r}_o, \mathbf{r}_i \in \boldsymbol{R}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{p} \in \boldsymbol{P} \,\}$$

Where $r_o \text{—}^{s}/_{p}\!\rightarrow r_i$ represents interaction between $\boldsymbol{r}_o$ and $\boldsymbol{r}_i$ via a style $\boldsymbol{s}$ while complying with a protocol $\boldsymbol{p}$.

Applying specific constraints on compositions develops sophisticated compositional patterns. While GRWs provide adherence to restricted, plug-compatible interfaces for interaction and template of behavior for computation, the components derived from GRWs will specify, refine or override the template. In this way, interactions are separated from computations.

Compositional patterns CP are the relation on the Cartesian product of compositions with the constraints reasonably localized on roles, styles and protocol:

$$\boldsymbol{CP}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{\mathbf{GRW}(\boldsymbol{r}_o) \text{—}^{s}/_{p}\!\rightarrow \mathbf{GRW}(\boldsymbol{r}_i) \mid \boldsymbol{r}_o, \boldsymbol{r}_i \in \boldsymbol{R}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{p} \in \boldsymbol{P}, \mathbf{Ct}(\boldsymbol{r}_o, \boldsymbol{s}, \boldsymbol{p}, \boldsymbol{r}_i)\,\}$$

Where ***GRW***(***r***) abstracts the role ***r*** as a GRW that separates interaction from computation (the GRW "provides" while the component "performs"). $\text{—}^{s}/_{p}\!\rightarrow$ represents interaction between $\boldsymbol{r}_o$ and $\boldsymbol{r}_i$ via a specific style $\boldsymbol{s}$ while complying with a specific protocol ***p***. ***Ct***($\boldsymbol{r}_o$, ***s***, ***p***, $\boldsymbol{r}_i$) represents localized constraints.

4.1 Example of compositional patterns

Compositional patterns can be implemented as composers, an explicit architectural element. They can be organized in a reusable composer library that provides the evolutionary foundation for component derivation. Fig. 7 gives the typical composer *Pipeline* that exhibits dependable architectural properties (e.g., loose component coupling, asynchronous communication, and data buffering). The two sides interconnected by the composer are the *Outflow* and *Inflow* roles, respectively. *Outflow* deputizes the producer to output the data, while *Inflow* deputizes the consumer to input the data via *Pipeline*. The formal *Pipeline* composer provides two generic parameters for enhancing reusability: transported ***Data*** (a basic item for dataflow) and buffer ***Size*** (a data transportation buffer).

This example provides a template for GRWs. With respect to behavioral computation of components, the CSP-based semantic description provides not only synchronous constraints but also asynchronous control transits. Both *Output* and *Input* are designed as exclusive procedures (*execution guards* are used to coordinate concurrent synchronization). Reference timing constraints [5-6], the role of *Outflow* is subjected to a maximum execution time (*met*) while *Inflow* is subject to a maximum response time (*mrt*). Both **met**(100) and **mrt**(100) are translated as asynchronous control transits for runtime monitoring of the real time constraints. "◊" represents an asynchronous operation. When outputting produced data onto the given pipeline, *Outflow* must be synchronized within a **met**(100) otherwise, an exception is triggered.

```
composer Pipeline is generalized
      type Data is private;
      Size : Integer : = 100;
style as   <#pipe-filter#>;
protocol as   <#dataflow-stream#>;
wrapper
   role Outflow is
    port
       procedure Output(d: Data);
       procedure Produce(d: Data) is abstract;
    computation
         Produce (d);
       *[ Output (d) → Produce (d) ◊ met(100) →exception; ]
    end Outflow;
    role Inflow is
    port
       procedure Input(d: Data);
       procedure Consume(d: Data) is abstract;
    computation
       *[ Input (d)→ Consume (d) ◊ mrt(100) →exception; ]
    end Inflow;
collaboration (P : Outflow; C : Inflow)
      P•Produce(d);
   *[ P•Output(d)→ P•Produce(d) □ C•Input(d) → C•Consume (d)]
end Pipeline;
```

Fig. 7 A Formal composer for Pipeline

The collaboration portion of the composer description will generate topological configurations that are connected graphs of components and composers. In concert with models of components and composers, configurations enable assessment of the autonomous and concurrent aspects of an architecture (such as the potential for deadlocks, starvation, reduced performance, reliability, security, etc.). Configurations also enable concurrent execution immediately after the roles are glued with the instances of corresponding components.

4.2 Substantiated interconnection

It used to be that interconnections in the architecture of a software system were annotated as a series of "box-line-box" diagrams [8-10]. Over time, this annotation has become much richer (for instance, the use of Architecture Description Languages (ADLs)) in order to more precisely capture and communicate more complex ideas related to interconnection. PBA continues in this vein by substantiating the interconnections among components so that large, complex architectures of systems can be built, dealing with following four aspects:

- *Dependable composers* by which interaction among components are promoted,
- *Heterogeneous forms* by which communication during interaction can be established.
- *Topological connectivity* that guides the connected configuration of components, and
- *Constraint localization* that governs interconnections by associating constraints on patterns

Dependable composers are used to implement compositional patterns by analyzing interactive roles of interconnected components in the prototyping model. Heterogeneous forms are associated with architectural styles and the way information is transported and refers to as communicatory protocols in compositional patterns. Constraint localization is presented next.

Topological connectivity simplifies the interconnection among components and comes in the following forms:

- Fork (1~N): single producer to multiple consumers
- Merge (N~1): multiple producers to single consumer
- Unique (1~1): single producer to single consumer
- Hierarchy: external[1] producer to interact with the internal[1] consumer, and vice versa.

Fig. 8 illustrates how to use a composer to implement *Fork* between one producer and more than one consumer.

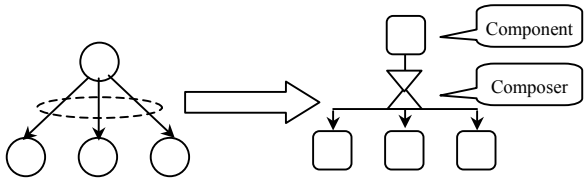

Fig. 8 Fork Connectivity with one PBA composer

4.3 Dependability as a set of Constraints

In this case "localization" represents the abstraction of dependability, its translation to quantitative constraints, and the handling of these constraints applied (localized) in the design, construction, and evolution of a software-intensive system. In order to achieve high confidence in the dependability of a system there must be a systematic method for expressing the dependability objectives via measurable constraints associated with the subsystems of the architecture. In a macro view, dependability can be abstracted as availability, reliability, safety, confidentiality, integrity and flexibility [15-17]. How these qualitative global requirements translate into quantitative constraints becomes crucial. Which dependable properties need translating and how they are localized on compositional patterns are the questions that have to be answered.

[1] External and internal refer to hierarchical decomposition. For a given hierarchical level of decomposition, a component in the current level is *external* to a component in a lower level, while the latter is the *internal* to the former.

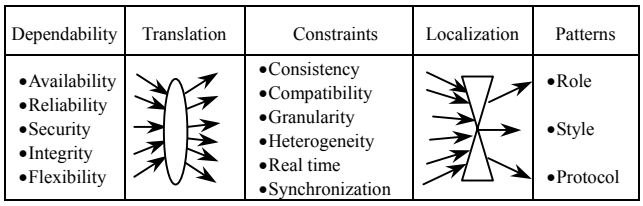

Fig. 9 Localization of Dependability

Fig. 9 shows a framework of localization applied to dependability. With respect to translating dependability and localizing the semantic constraints on the compositional patterns, the handling of real-time constraints provides a good example. Reliability of the time-critical system may be embodied as an immediate reply of a particular component, under a given request, within an *met*, or as a data stream between components performed within a specific *latency* [5]. First, this time-critical reliability should be translated into timing constraints *met* and *latency* (two quantitative constraints). Both are associated with the patterns referring to the role and protocol, respectively. *met* requires computation of the role (the component acts) and must be executed within a specific amount of time (a hard real-time constraint). The *latency* constrains the maximum delay during data transportation within the protocol. These timing constraints can be also verified by runtime monitoring and correctness assurance [15-17]. Dependability of the system would be translated into in the form of maximum execution time or latency of the data stream communication between components as shown in Fig. 10.

```
composer Pipeline is generalized
    …
  role Outflow is
  port
      procedure Output(d: Data);
      procedure Produce(d: Data) is abstract;
  computation
          Produce (d);
        *[ Output (d) ◊ latency(60) → Produce (d) ◊ met(100)
          □ latency-signaled          → LAT-EXCEPTION
          □ met-signaled              → MET-EXCEPTION
         ]
  end Outflow;
    … …
end Pipeline;
```

Fig. 10 A Formal composer for Pipeline

Procedure Output can be treated as execution guard that is tied to the communication protocol, so latency is associated with to the protocol by Output (d) ◊ **latency**(60). When executing Output is beyond the latency, the asynchronous control will set latency-signaled and abort current execution, and then raise LAT_EXECPTION. Similarly, *met* is directed to the procedure Produce by Produce (d) ◊ **met**(100). When executing Produce is beyond the *met* limitation, the asynchronous control will set met-signaled and abort current execution, and then raise MET_EXCEPTION.

5. Conclusion

Explicitly architecting software-intensive systems provides the promise of faster, better, cheaper systems. In order to consistently engineer dependable software-intensive systems, the PBA approach provides a process that uncovers perspective concerns of different stakeholders, and increases the effectiveness of requirements validation techniques. Because PBA approach can be used to localize and quantify invariant architectural constraints (such as "dependability" in the example above) it will also reduce the amount of re-certification effort required after each requirement change. The PBA approach illustrates that with automated tool support, the prototyping of software architecture can be used to identify and resolve conflicting stakeholder perspectives and develop reliable, dependable, consistent software-intensive systems.

References

[1] C. Hofmeister, R. Nord, D. Soni. Applied Software Architecture. Addison-Wesley, 2000.

[2] IEEE Standard Board, Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-std-1471 2000), September 2000.

[3] H. Alexander, *et el*, C4ISR Architectures: I. Developing a Process for C4ISR Architecture Design. Systems Engineering, John Wiley and Sons, Inc., Vol. 3 No. 4, 2000.

[4] W. Lee, et el, Synthesizing Executable Models of Object Oriented Architectures. Proc. Formal Methods in Software Engineering & Defence Systems. Adelaide, Australia, 2002.

[5] Luqi, M. Ketabchi, A computer-Aided Prototyping System, IEEE Software, March 1988.

[6] Luqi, Ying Qiao, Lin Zhang, Computational Model for High-Confidence Embedded System Development, Monterey workshop 2002, Venice, Italy, Oct 7-11, 2002.

[7] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Inc., 1996.

[8] Andrew P., Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives, System Engineering, John Wiley and Sons, Inc., 1998.

[9] N. R. Mehta, N. Medvidovic. Towards a Taxonomy of software Connectors. Proc. ICSE, Limerick Ireland, 2000.

[10] N. Medvidovic, Taylor, A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 2000, 26(1).

[11] X Liang, Event-based implicit invocation decentralized in Ada, ACM AdaLetters, March, 2002.

[12] Sessions N., COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, Inc., NY, 1997.

[13] OMG/ISO Standard, CORBA: Common Object Request Broker Architecture, http://www.corba.org/.

[14] Sun Microsystems, Inc. Java 2 Enterprise Edition Specification v1.2. http://java.sun.com/j2ee/.

[15] E. M. Clarke (CMU), R. P. Kurshan (Bell Lab). Computer-Aided Verification, Feb. 17, 1996.

[16] James C. Corbett, et el, Bandera: Extracting Finite-state Models from Java Source Code, Proc of the ICSE 2000.

[17] M. Kim, *et el*, Monitoring, Checking, and Steering of Real-Time Systems, 2nd Intl. Workshop on Run-time Verification. Copenhagen, Denmark, July 26, 2002.

[18] X Liang, Z. Wang. Omega: A Uniform Object Model Easy to Gain Ada's Ends, ACM AdaLetters, June, 2000.