

# Refinement and Evolution Issues in Bridging Requirements and Architectures

Alexander Egyed, Paul Gruenbacher, and Nenad Medvidovic

University of Southern California, Computer Science Department,  
Los Angeles, CA 90089-0781, USA  
{aegyed, gruenbac, neno}@sunset.usc.edu

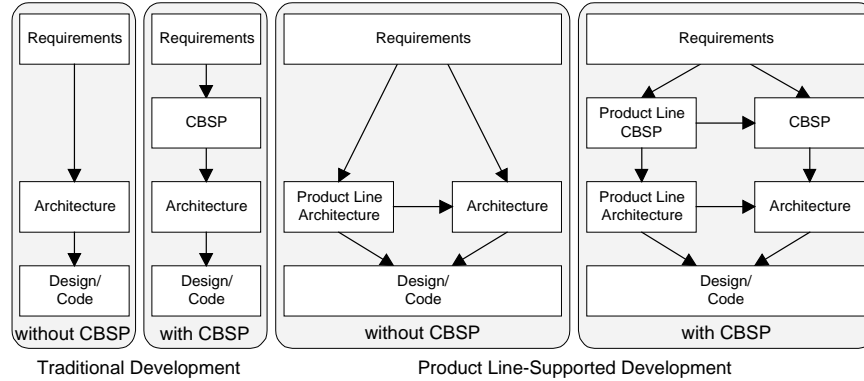
**Abstract.** Though acknowledged as very closely related, to a large extent requirements engineering and architecture modeling have been pursued independently of one another, particularly in the large body of software architecture research that has emerged over the past decade. The dependencies and constraints imposed by elements of one on those of the other are not well understood. This paper identifies a number of relevant relationships we have identified in the process of trying to relate the WinWin requirements engineering approach with architecture and design-centered approaches (e.g., C2 and UML). This paper further discusses their relevance towards product line issues which currently are obscured by the fusion of product-specific and product-line information.

## 1 Introduction

Requirements negotiation and elaboration success addresses issues such as identifying critical stakeholders, capturing stakeholder goals and concerns, discovering conflicts between stakeholder concerns, and addressing those conflicts by identifying suitable options for resolving them. Architectural modeling, on the other hand, deals with issues such as defining components, connectors, and systems, as well as their properties and roles.

The software engineering community has not failed to notice a natural gap between the two tasks of requirements negotiation and architecting. Despite extensive attention, this gap remains. Effectively, transitioning from requirements to an architecture is still an unsolved problem. People have found that this task becomes more manageable in a waterfall-like situation where requirements are clearly specified and completed before the actual product is built. The drawback of a waterfall approach is that most projects exhibit strong iterative features. The complex task of refining requirements to an architecture is therefore made even more difficult by having to consider continuous evolution.

In order to deal with this issue, we have created a systematic approach for refining a system's requirements to its architecture, called CBSP [7]. CBSP stands for *Component, Bus (Connector), System, and Property*. A CBSP-enabled development approach aims at identifying and qualifying requirements that are architecturally relevant with



**Figure 1. Traditional and Product Line Development with or without CBSP**

respect to those four properties. Initially, we created CBSP to support traditional architecture-based software development without an explicit focus on product lines. Figure 1 (left) depicts at a high level development with and without CBSP. CBSP refinement, which will be discussed in more detail shortly, simplifies the refinement from requirements to architecture by defining a process for eliciting architecture-relevant information from requirements. CBSP was built to support evolutionary development where both architecture and requirements are evolved continuously.

This paper shows how the CBSP approach can support architecture-based development of product lines. The right of Figure 1 depicts a model of product-line-development with and without CBSP. It has been our observation that without CBSP the roles of requirements and architecture modeling in product lines become somewhat obscured because of the conceptual differences between product-line artifacts and regular product artifacts.<sup>1</sup> The basic problem is that requirements, initially, are in a product-specific form (e.g., *Optimize applications concurrent routing to increase speed of high-priority cargo delivery*). Because of a product-line-supported development approach, the task of refining requirements to architectures (a very complex task in its own right) gets obscured by having to additionally handle product-line-architectural elements that are conceptually different from the product-specific requirements.

Naturally, one could conceive a product-line specific form of requirements capture analogous to our concept of family design [6]. However, requirements capture frequently involves plain English and therefore lacks precision. “Product Line Requirements” cannot easily be captured systematically to support their instantiation (into product requirements) as well as refinement into architectures. Using our CBSP approach as a mediator between requirements and architecture gives the advantage that CBSP artifacts can be systematically defined and captured under the umbrella of architectural relevance.

<sup>1</sup> Product artifacts are product-specific instantiations of product line artifacts.

The next section describes how CBSP can be used to capture and refine architectural relevant artifacts. Thereafter, we will describe the product-line extensions to CBSP. This paper also briefly discusses needed technologies and summarizes our findings in the end.

## 2 CBSP Refinement

This section discusses CBSP in context of a real-world negotiation example. Consider the following evolutionary example taken from an actual negotiation: A group of stakeholders meet to negotiate the requirements for a *Cargo Routing* system. The actual negotiation included roughly 60 individual stakeholder “win conditions” (concerns, goals, and wishes). Two such conditions are: “Optimize concurrent routing to increase speed of high-priority cargo delivery” (W1) and “Support real-time communication from system to vehicle” (W2). As the negotiation unfolds, an issue (conflict) between those two concerns is identified: “In order to optimize concurrent routing, we need to support bi-directional real-time communication” (I1). This issue leads to an option (O1) that suggests realizing a two-way communication. O1 then *replaces* W2 and can now be considered a newer version of W2.

The left side of Figure 2 shows a graphical breakdown of above negotiation process using the WinWin’s negotiation rationale view. The WinWin requirements negotiation process has been defined and discussed previously [3]. To briefly summarize, the model contains four major artifact types – *Win Condition* (W), *Issue* (I), *Option* (O), and *Agreement* (A). *Win Conditions* capture stakeholder goals and concerns with respect to the new system. If a win condition is non-controversial, it is covered by an Agreement. Otherwise, an Issue artifact is created to record the resulting conflict among win conditions (and stakeholders). Options allow stakeholders to suggest alternative solutions, thereby addressing issues. Options can be explored and refined via tradeoff analysis, expectations management, and negotiation, eventually leading to an agreement to adopt an option, thus resolving the issue. The example in Figure 2 indicates a part of this process.

The WinWin negotiation model is very powerful in capturing and resolving stakeholder concerns. To create an architecture, however, it provides rather incomplete information. This may cause two fundamental problems: (1) how to systematically derive an architecture out of stakeholder goals and expectations; and (2) how to provide feedback from the architecture to the negotiation process (e.g., in form of issues and options).

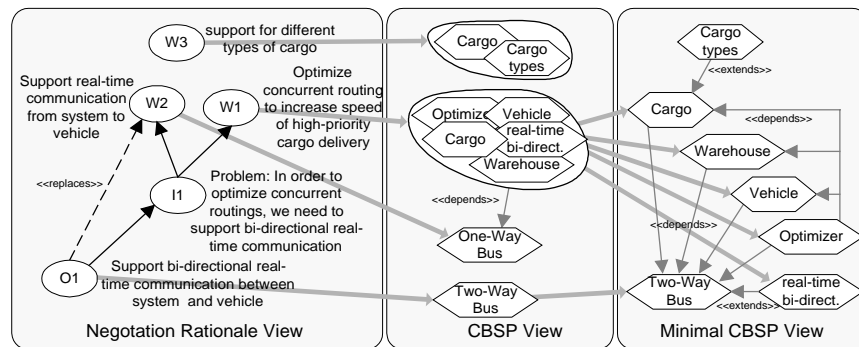
### 2.2 Refinement from Requirements to Architecture

To handle round-trip engineering between requirements and architecture, we introduce the CBSP process (see also [7]) that is briefly discussed below.

- [1] Identify architecturally relevant artifacts: In the course of WinWin negotiations, artifacts such as win conditions, issues, options, and agreements are uncovered

by stakeholders. These artifacts cover a broad spectrum to expected capabilities, interface issues, levels of services, and development issues. Our starting hypothesis was that artifacts could be partitioned into two major groups: the ones that are architecturally relevant, and the ones that are not. The stakeholders (e.g., architects) are thus asked to classify each artifact with respect to their relevance in becoming architectural components (C), connectors (B)<sup>2</sup>, systems (S), and properties (P) that can be associated with C, B, and S (categories reflect a general classification of architecture description languages [9]). In our example, W1 was voted as being *fully* component relevant, *largely* connector (bus) relevant, and *largely* property relevant (CP and BP). W2 and O1 were voted as being *largely* connector (bus) relevant and *fully* connector property relevant.<sup>3</sup> W3 was voted as being largely component relevant and I1 (I itself) was voted as being not architecturally relevant.

- [2] Specify interdependencies between negotiation artifacts: To further elaborate about the nature of the relationships between architecturally relevant artifacts, dependencies among them need to be created. In our example, we found W1 to be dependent to W2 (and indirectly to O1, since O1 replaced W2). Dependency implies that W2 offers services that W1 needs.
- [3] Split complex negotiation artifacts into atomic CBSP artifacts: This is done in (at least) those cases where architecture-relevant artifacts are classified into multiple CBSP categories. For instance, W1 was voted to be component-, connector-, and property-relevant. Upon investigation, it is revealed that it describes multiple architectural elements. Figure 2 (middle) shows the result of the splitting and dependency activities in our example: There, W1 is divided into several components, a connector, and a connector property.<sup>4</sup> The figure also depicts the dependency between the architecture-relevant artifacts described by W1 and W2.
- [4] Minimize CBSP by *eliminating replaced* and *merging related* artifacts. In our example, “One-Way Bus” was replaced by a “Two-Way Bus” and, if the former is not used anywhere else, it can be removed (hidden). Further, both W1 and W3 describe a *Cargo* artifact, allowing it to be merged. Figure 2 (right) shows the result of minimizing our example. Note that the artifacts *Cargo Types* and *Real-*

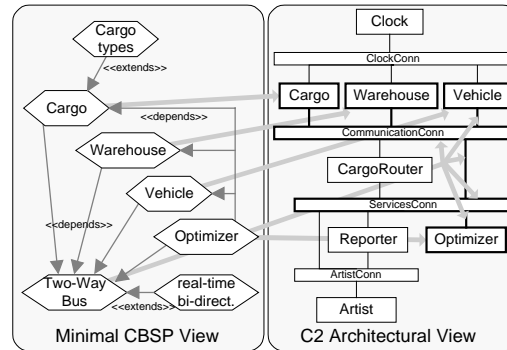


**Figure 2. From negotiation rationale view to architecture relevant CBSP view**

<sup>4</sup> We did not display all of them to improve visualization

*time bi-direct.* could be collapsed into *Cargo* and *Two-Way Bus*, respectively, since they are only extensions (features).

The minimized CBSP view may not necessarily cover all relevant architectural elements, nor all their interdependencies. Further, it may not separate subsystems and items at different levels of abstraction. Nevertheless, the CBSP view captures and relates significant architectural elements and can thus be considered a coarse approximation of an architectural view in its own right. For an architect, the task



**Figure 3. Architecture Relevant Artifacts**

of refining a CBSP view into a more standardized architectural representation is an easier one than it was the case with the negotiation rationale view of on the left side of Figure 2. To this end, Figure 3 shows one possible realization of our example using the C2 architectural style [8]. The figure also depicts how the C2 view and its elements relate to the minimal CBSP view. The architecture in Figure 3 depicts the major components *Vehicle*, *Warehouse*, *Cargo*, and *Optimizer* as they were identified in Figure 2. The “Two-Way-Bus” from Figure 2 was however refined into a set of two Connectors and a link between them. This is done to accommodate core cargo routing services on top of which the *Optimizer* sits. The C2 solution further incorporates some of the other stakeholder goals we did discuss here for brevity. Their omission should not matter since even at a later refined stage, the relationships between this piece of the negotiation and the architectural realization(s) should remain intact.

### 2.3 Mismatch Detection

Making easier the refinement of requirements to architecture is one benefit of the CBSP approach. However, as the example in Figure 3 indicates, the relationships between architecture, CBSP, and the negotiation rationale become more complex when both architecture and requirements evolve independently (this property is also particularly important for product-line development since there product-lines and products must evolve concurrently). To this end, we found that CBSP provides powerful support for simplifying mismatch detection between requirements and architectural concerns. For instance, we observed the following cases:

- [1] Mismatch between CBSP artifact categorization/dependencies and their actual realization. For instance, if a CBSP artifact categorized as a component is implemented as a connector in C2, then this case could indicate a potential lack of

understanding of either the requirement or architecture. Similar conclusions can be drawn when CBSP dependencies do not match architectural ones.

- [2] Mismatch between architectural and/or CBSP core elements and negotiation agreements. For instance, the *Optimizer* component in Figure 3 depends on the *Two-Way-Bus*. If during the WinWin negotiation, it is decided to implement the *Optimizer* but, at the same time it is decided to implement the *One-Way-Bus* instead of the *Two-Way-Bus*, then this indicates a potential mismatch (*Optimizer* needs the *Two-Way-Bus*).
- [3] Completeness mismatch between architecture and requirements. For instance, are all agreed-upon architecturally relevant artifacts realized? Are there any architectural elements for which there are no corresponding negotiation artifacts? Completeness issues such as the ones above could suggest a lack of awareness by stakeholders of some architectural aspects that could have influenced the negotiation process.
- [4] Property mismatches indicating that architecture cannot satisfy requirements. A strength of many architecture description languages is their ability to validate the architecture early and simulate certain system properties such as performance, reliability, schedulability, etc. Feedback from such analyses to the negotiation could significantly influence the negotiation process. The CBSP approach provides an enabling foundation for doing this since it is aware of how architecturally relevant artifacts relate to negotiation artifacts. We refer to this relationship as traces.

#### 4 Trace Requirements

To enable tool-supported refinement and evolution, we need to provide the following models and traces:

- [1] Negotiation model supporting the definition of goals, concerns, conflicts, alternatives, and agreements, as well as traceability links among those elements to describe their relationships and change history. We currently make use of WinWin, which supports *win conditions*, *issues*, *options*, and *agreements* and allows *involve*, *address*, *adopt*, *cover*, and *replace* trace links between them.
- [2] CBSP model supporting the definition (and voting) of architecturally relevant artifacts as well as the activities described in the *Refinement* section. To link the CBSP model with the negotiation model we employ simple *describe* traceability links. The CBSP model itself needs to distinguish between *depends* and *replaces* within CBSP artifacts. We currently use GroupSystems (from GroupSystems.com) for this task.
- [3] Architecture and design models supporting the realization of architecturally relevant artifacts. To link the CBSP model to an architecture, only simple *describe* traceability links are needed. We currently use the C2 architectural model to realize (higher-level) architectures and the Unified Modeling Language

(UML) [4] to realize (lower-level) designs. The details of our approach to refine C2 to UML can be found in [1].

### 3 Product-Line CBSP

Our CBSP approach has the benefit that requirements artifacts are mapped onto architectural artifacts by identifying more atomic elements the two have in common. A product-line-supported CBSP would do the same for requirements and product line architectures. The product line CBSP is the accumulation of project-specific CBSPs originally derived from previous requirements negotiations. Having a product line CBSP has the advantage that product specific CBSPs are more easily identifiable by using the product line CBSP as a reference taxonomy (analogous to how a product line architecture functions). The product line CBSP has the additional advantage that it also serves as a mediator to product-line architecture(s) by easing the identification of product-line-architectural artifacts that relate to CBSP artifacts and subsequently to the requirements we are interested in.

The *Cargorouter* application we briefly discussed above has been built multiple times before using different requirements. Currently, five versions of the *Cargorouter* application exist, all of which have been architected using the C2 style. This application is therefore well suited in investigating product line issues. Figure 4 depicts, at a high level, product line versions of CBSPs and architectures. The left side depicts one possible view of the combined CBSPs of our version of the *cargorouter* and a multilingual version of it where a translator component is used to intercept messages between the *Cargorouter* component and the artist component (the artist is responsible for the user interface). The dashed items reflect CBSP artifacts that have only been used in some versions. The other items are artifacts used in all versions. The right side

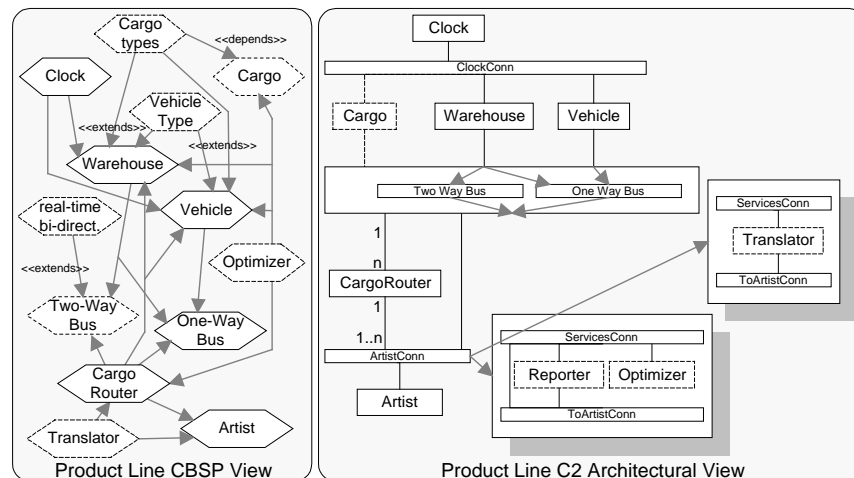


Figure 4. Product Line CBSP and Architectures

of Figure 4 depicts a product line interpretation of both versions of the *cargorouter*. In order to make the application bi-lingual, the *CargoRouter* component must be instantiated at least twice (the cardinality reflects that) and the *ArtistConn* connector must be refined into two connectors with a *Translator* component in between. Our newer version of the *Cargorouter* builds upon the same core architecture and also refines the *ArtistConn* connector, though, in a different fashion (e.g., two connectors with the components *Reporter* and *Optimizer* in between).

The major benefits of a product line CBSP are during requirements negotiation and refinement. Having a CBSP makes it easier to investigate previous product versions by using CBSP artifacts as links to their requirements and realizations. For requirements negotiation this also enable the rapid identification of previously unexplored architectural configurations – a potential source of high risks.

## 5 Tool Support and Automation

The WinWin negotiation model, the CBSP model, and the C2 architecture model are tool supported. For WinWin and CBSP modeling we use EasyWinWin [2] and GroupSystems (GroupSystems.com), for C2 architectural modeling and refinement to UML designs we use the SAAGE tool suite [8] [1], and for design modeling and consistency checking we use Rational Rose and UML/Analyzer [5]. We currently have no automation for minimizing the CBSP graph or detecting mismatches between requirements and architecture, although we envision building those in the future. GroupSystems is integrated our tools with SAAGE and UML/Analyzer via Rational Rose. We have integrated with Rational Rose in order to use it as a graphical front-end for EasyWinWin, CBSP, and UML/Analyzer (using Rose as a front-end for SAAGE is still under development) but also to enable traceability links between all artifact types under one common roof. Tool to support for product line CBSP and product line C2 is in the planning stages.

## 6 Conclusion

The paper introduced the CBSP process for refining requirements to an architecture. The process is partially tool supported and is currently integrated with our WinWin negotiation and our C2 architecture-based development process. Besides refinement, the CBSP process also improves the consistency and conformance checking between requirements and architecture. To this end, we introduced a number of mismatch types and indicated how CBSP can simplify their identification. This work also indicated how software development via product lines could benefit from a more rigorous treatment of the mapping between requirements and architectures. Product line architectures introduce an additional dimension of complexity that requires attention. Future work involves a tighter integration of our models and tools.

## Acknowledgements

This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the Affiliates of the USC Center for Software Engineering: <http://sunset.usc.edu/CSE/Affiliates.html>.

## References

- [1] Abi-Antoun, M. and Medvidovic, N.: "Enabling the Refinement of a Software Architecture into a Design," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, October 1999.
- [2] Boehm, B. and Gruenbacher, P.: "Supporting Collaborative Requirements Negotiation: The Easy WinWin Approach," *Proceedings of SCS Virtual Worlds and Simulation Conference*, January 2000.
- [3] Boehm, B. W., Bose, P., Horowitz, E., and Lee, M. J.: "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," *Proceedings of 17th International Conference on Software Engineering (ICSE 17)*, April 1995, pp.243-253.
- [4] Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
- [5] Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, May 2000.
- [6] Egyed, A., Mehta, N., and Medvidovic, N.: "Software Connectors and Refinement in Family Architectures," *Proceedings of 3rd International Workshop on Development and Evolution of Software Architectures for Product Families (IWSAPP)*, March 2000.
- [7] Gruenbacher, P., Egyed, A., and Medvidovic, N.: "Separation of Concern in Requirements Negotiation and Architecture Modeling," *Proceedings of Workshop on Multi-dimensional Separation of Concerns in Software Engineering co-located with ICSE 2000*, June 2000.
- [8] Medvidovic, N., Rosenblum, D. S., and Taylor, R. N.: "A Language and Environment for Architecture-Based Software Development and Evolution," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999, pp.44-53.
- [9] Medvidovic N. and Taylor R. N.: A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), 2000, 70-93.