Softw Syst Model (2003) / Digital Object Identifier (DOI) 10.1007/s10270-003-0038-6

Reconciling software requirements and architectures with intermediate models

Paul Grünbacher[1], **Alexander Egyed**[2], **Nenad Medvidovic**[3]

[1] Systems Engineering and Automation, Johannes Kepler University, 4040 Linz, Austria; E-mail: gruenbacher@acm.org
[2] Teknowledge Corporation, 4640 Admiralty Way, Marina Del Rey, CA, USA; E-mail: aegyed@acm.org
[3] Computer Science Dept., Univ. of Southern California, Los Angeles, CA, USA; E-mail: neno@usc.edu

Published online: 10 December 2003 – © Springer-Verlag 2003

Abstract. Little guidance and few methods are available for the refinement of software requirements into an architecture satisfying those requirements. Part of the challenge stems from the fact that requirements and architectures use different terms and concepts to capture the model elements relevant to each. In this paper we will present CBSP, a lightweight approach intended to provide a systematic way of reconciling requirements and architectures using intermediate models. CBSP leverages a simple set of architectural concepts (components, connectors, overall systems, and their properties) to recast and refine the requirements into an intermediate model facilitating their mapping to architectures. Furthermore, the intermediate CBSP model eases capturing and maintaining arbitrarily complex relationships between requirements and architectural model elements, as well as among CBSP model elements. We have applied CBSP within the context of different requirements and architecture definition techniques. We leverage that experience in this paper to demonstrate the CBSP method and tool support using a large-scale example.

Keywords: Requirements elicitation and negotiation – Architecture modeling – Intermediate models – Traceability

1 Introduction

Software systems of today are characterized by increasing size, complexity, distribution, heterogeneity, and lifespan. They demand careful capture and modeling of requirements [37, 44] and architectural designs [40, 46] early on, before low-level system details begin to dominate the engineers' attention and significant resources are expended for system construction. Understanding and supporting the interaction between software requirements and architectures remains one of the challenging problems in software engineering research [36, 37]. Evolving and elaborating system requirements into a viable software architecture satisfying those requirements is still a difficult task, mainly based on intuition and experience. Similarly, little guidance is available for modeling and understanding the impact of architectural choices on the requirements. Software engineers face some critical challenges when trying to reconcile requirements and architectures:

- Requirements are frequently captured informally in a natural language. On the other hand, entities in a software architecture specification are usually specified in a more formal manner causing a semantic gap [31].
- System properties described in non-functional requirements [9] are commonly hard to specify in an architectural model [11, 31].
- The iterative, concurrent evolution of requirements and architectures demands that the development of an architecture be based on incomplete requirements. Also, certain requirements can only be understood after modeling or even partially implementing the system architecture [13, 36].
- Mapping requirements into architectures and maintaining the consistency and traceability between the two is complicated since a single requirement may address multiple architectural concerns and a single architectural element (for example a COTS component) typically has numerous non-trivial relations to various requirements. The increasing importance of component-based software development (CBD) emphasizes the need for agile techniques that allow cap-

This paper represents a major revision and extension of the work that has been published in the Proceedings of the Requirements Engineering 2001 conference [21].

ture and understanding the complex relationship between requirements and architectural elements.

- Real-world, large-scale systems have to satisfy hundreds, possibly thousands of requirements [6]. It is difficult to identify and refine the architecturally relevant information contained in the requirements due to this scale.
- Requirements and the software architecture emerge in a process involving heterogeneous stakeholders with conflicting goals, expectations, and terminology [3]. Supporting the different stakeholders' interests demands finding the right balance across these often divergent interests.

In aiming to address these challenges we have developed a lightweight method for identifying the key architectural elements and the dependencies among those elements, based on the stated system requirements. Our CBSP (Component-Bus-System-Property) approach helps to refine a set of requirements by applying a taxonomy of architectural dimensions. The intent of our work is to provide a generic approach that primarily works with arbitrary informal or semi-formal requirements representations as well as different architecture modeling approaches. Although requirements may also be captured in a formal language such as KAOS [27], informal or semi-formal approaches are still used very frequently. In order to validate our research to date, we have applied CBSP extensively in the context of EasyWinWin [4, 20], a groupware-supported requirements negotiation approach. EasyWinWin has been selected because it supports multi-stakeholder elicitation of requirements and captures requirements informally but in a structured fashion.

CBSP provides an intermediate model between the requirements and the architecture that helps to iteratively evolve the two models [36]. For example, a set of incomplete and quite general requirements captured as statements in a natural language might be available. The intermediate CBSP model then captures architectural decisions as an incomplete "proto-architecture" that *prescribes* [5] further architectural development. The intermediate model still "looks" like requirements but "sounds" like an architecture. The CBSP approach also guides the selection of a suitable architectural style to be used as a basis for converting the proto-architectures into an actual implementation of a software system architecture.

Figure 1 shows the CBSP model in the context of the Twin Peaks model [36]. The Twin Peaks model suggests that requirements and architectures are evolved iteratively and concurrently. In such a context, the intermediate CBSP model can be used at different levels of detail in the modeling process. For example, it can help to refine high-level, informal requirements early in a project and more elaborated requirements in later iterations; or it can also help to understand how issues arising in architecture modeling and simulation relate to the requirements.

CBSP provides:

- a lightweight way of refining requirements using a small, extensible set of key architectural concepts;
- mechanisms for "pruning" the number of relevant requirements, rendering the technique scalable by focusing on the architecturally most relevant set of artifacts;
- involvement of key system stakeholders, allowing non-technical personnel (e.g., customers, managers, even users) to see the impact of requirements on architectural decisions if desired;
- adjustable voting mechanisms to resolve conflicts and different perceptions among architects; and
- tools supporting selected steps in the approach (however, we would like to stress that a full automation

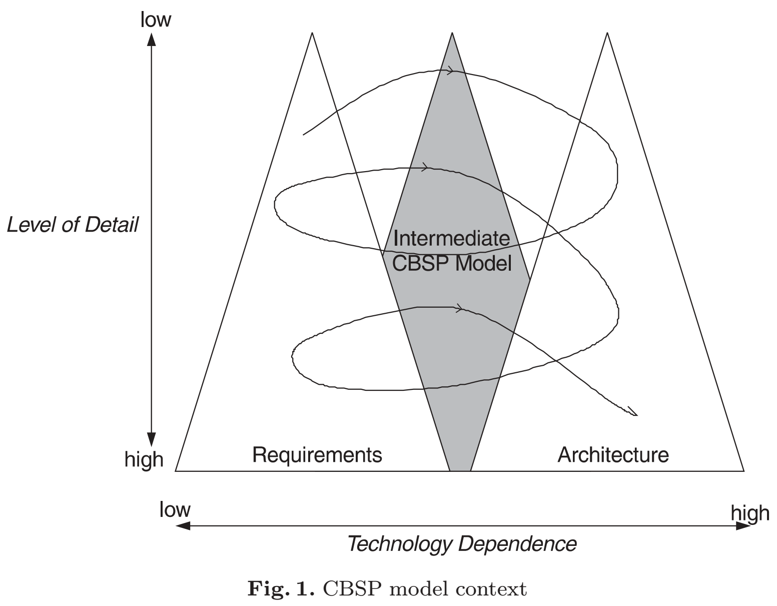


Fig. 1. CBSP model context

of the approach is neither possible nor desirable and human decision-making is an important aspect of the approach).

Together, these benefits afford a high degree of control over refining large-scale system requirements into architectures.

The paper is organized as follows: Sect. 2 discusses details of the CBSP approach. Section 3 describes the application of CBSP in a large-scale example. Section 4 describes our current tool support. Related work will be discussed in Sect. 5. Conclusions round out the paper in Sect. 6.

2 The CBSP approach

According to (IEEE-610-1990) [23] a requirement is "a condition or capability needed by a user to solve a problem or achieve an objective." *Requirements* largely describe aspects of the problem to be solved and constraints on the solution, i.e., desired system features and properties (both functional and non-functional [9]). Requirements are derived from the concepts and relationships in the problem domain (e.g., medical informatics, E-commerce, avionics, mobile robotics).

Requirements may be simple or complex, precise or ambiguous, stated concisely or elaborated carefully. Of particular interest to our work is a large class of requirements that is predominantly stated in a natural language, as opposed to precise formalisms. On the surface, such requirements are easier to understand by humans, but they frequently lead to ambiguity, incompleteness, and inconsistencies in the architecture and, eventually, the software system. It is advisable not only to capture the outcome of requirements elicitation, i.e., the requirements, but also the evolution of goals of system stakeholders such as customers, users, managers, developers [3] to preserve the history and rationale of the requirements production process [18].

The relationship between a set of requirements and an effective architecture for a desired system is not readily obvious. *Architectures* model a solution to a problem described in the requirements and provide high-level abstractions for representing the structure, behavior, and key properties of a software system. The terminology and concepts used to describe architectures differ from those used for the requirements. An architecture deals with components, which are the computational and data elements in a software system [40]. The interactions among components are captured within explicit software connectors (or buses) [46]. Components and connectors are composed into specific software system topologies. Finally, architectures both capture and reflect the key desired properties of the system under construction (e.g., reliability, performance, cost) [46]. These elements of software architectures can be specified formally using architecture description languages (ADLs) [31].

The above-described differences between requirements and architectures make it difficult to build a bridge that spans the two. For example, it is unclear in general whether and how a statement of stakeholder goals should affect the desired system's architecture; similarly, deciding how to most effectively address a functional requirement often boils down to relying on the architects' intuition, rather than applying a well-understood methodology. For these reasons, we have formulated CBSP, a technique easing the development of an architecture addressing a given set of requirements in a more straightforward and consistent manner than attempting to transfer directly requirements into architectures. This section will introduce the CBSP taxonomy of architectural dimensions and describe a process that guides the development of the intermediate CBSP model.

2.1 CBSP taxonomy

The fundamental idea behind CBSP is that any software requirement may *explicitly* or *implicitly* contain information relevant to the software system's architecture. It is frequently very hard to surface this information, as different stakeholders will perceive the same requirement in very different ways [20]. At the same time this architectural information is often essential in order to properly understand and satisfy requirements. CBSP supports the task of identifying architectural information contained in the requirements and explicating it in an intermediate model.

The CBSP dimensions include a set of general architectural concerns we have derived from existing software architecture research [31, 33, 40, 45, 46, 49]. These dimensions can be applied to systematically classify and refine requirements and to capture architectural trade-off issues and options (e.g., impact of a connector's throughput on the scalability of the topology).

Each requirement is assessed for its relevance to the system architecture's components, connectors (buses), topology of the system or a particular subsystem, and their properties. Thus, each derived CBSP artifact explicates an architectural concern and represents an early architectural decision for the system. For example, a requirement such as

R: The system should provide an interface to a Web browser.

can be recast into a CBSP processing component element (C_p) and a CBSP bus element (B)

C_p: A Web browser should be used as a component in the system.
B: A connector should be provided to ensure interoperability with third-party components.

It is important to emphasize that, while CBSP supports recasting requirements into more architecturally "friendly" model elements along well-defined dimensions, it does not prescribe a particular transformation of a requirement. Instead, our intent is to give a software architect sufficient leeway in selecting the most appropriate *refinement* or, at times, *generalization* of one or more requirements. Examples of both refinement and generalization are given below. Architecting software systems is inherently a human-intensive activity. The goal of CBSP is, therefore, not to eliminate the role of human architects in the process. Instead, its goals are to aid the architects in making decisions that are effective and timely and in preserving these decisions.

There are six possible CBSP dimensions discussed below and illustrated with a simple example from a spreadsheet manipulation application. The six dimensions involve the basic architectural constructs [31] and, at the same time, reflect the simplicity of the CBSP approach.

1. *C* are model elements that describe or involve an individual Component in an architecture. For example, the requirement:

   R: Allow user to directly manipulate spreadsheet data.

   may be refined into CBSP model elements describing both processing components (C_p) and data components (C_d)

   C_p*: Spreadsheet manipulation UI component.*
   C_d*: Data for spreadsheet.*
2. *B* are model elements that describe or imply a Bus (connector). For example:

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

   may be refined into

   B: Connector enabling interaction between UI and persistency components.
3. *S* are model elements that describe System-wide features or features pertinent to a large subset of the system's components and connectors. For example:

   R: The user should be able to select appropriate data filters and visualizations.

   may be refined into

   S: The system should employ a strict separation of data storage, processing, and visualization components.
4. *CP* are model elements that describe or imply data or processing Component Properties. As discussed above, the properties in CBSP are the "ilities" in a software system, such as reliability, portability, incrementality, scalability, adaptability, and evolvability. For example:

   R: The user should be able to visualize the data remotely with minimal perceived latency.

   may be refined into

   CP: The data visualization component should be efficient, supporting incremental updates.
5. *BP* are model elements that describe or imply Bus Properties. For example:

   R: Updates to system functionality should be enabled with minimal downtime.

   may be refined into

   BP: Robust connectors should be provided to facilitate runtime component addition and removal.
6. *SP* are model elements that describe or imply System (or subsystem) Properties. For example:

   R: The spreadsheet data must be encrypted when dispatched across the network.

   may be transformed into

   SP: The system should be secure.

Note that, e.g., the BP example (5) involved refining a general requirement into a more specific CBSP element. On the other hand, the SP example (6) involved the generalization of a specific requirement into a CBSP artifact. Refinements and generalizations such as those shown above are a function of the needs of the specific system under construction, the characteristics of the application domain, and the software architect's background and experience. Additionally, refining or generalizing a requirement may also require consulting the system's customers for additional context and information. As such, while it would undoubtedly be very useful, it is unrealistic to expect that formal rules could be provided for transforming a requirement into more specific or general CBSP elements.

A meta-model showing the different model elements relevant to CBSP is given in Fig. 2. Requirements are related to architectural elements such as components or connectors via an intermediate CBSP model that acts as a bridge. Different subtypes of CBSP elements are used to represent different architectural dimensions listed in the CBSP taxonomy.

2.2 The CBSP process

We also provide a step-by-step process and techniques supporting the synthesis of the intermediate CBSP model and the architecture in a collaborative manner. Figure 3 depicts process activities and deliverables using the IDEF0 notation [24]. In an envisioned iterative life-cycle the depicted CBSP process represents one cycle of evolving and refining an architecture out of a given set of requirements.

Each CBSP step is discussed in more detail below. We use ETVX (**E**ntry, **T**ask, **V**erification, and e**X**it) [42] to document the steps. ETVX cells consist of four components: (1) *Entry* lists all items required for the execution of the task (e.g., people, tools, artifacts, etc.); (2) *Task* describes what should be done, by whom, how, and when (this includes appropriate standards, procedures and responsibilities); (3) *Verification/ Validation* describes all checks and controls that help to indicate if the task is being executed properly; and (4) *eXit* lists criteria which

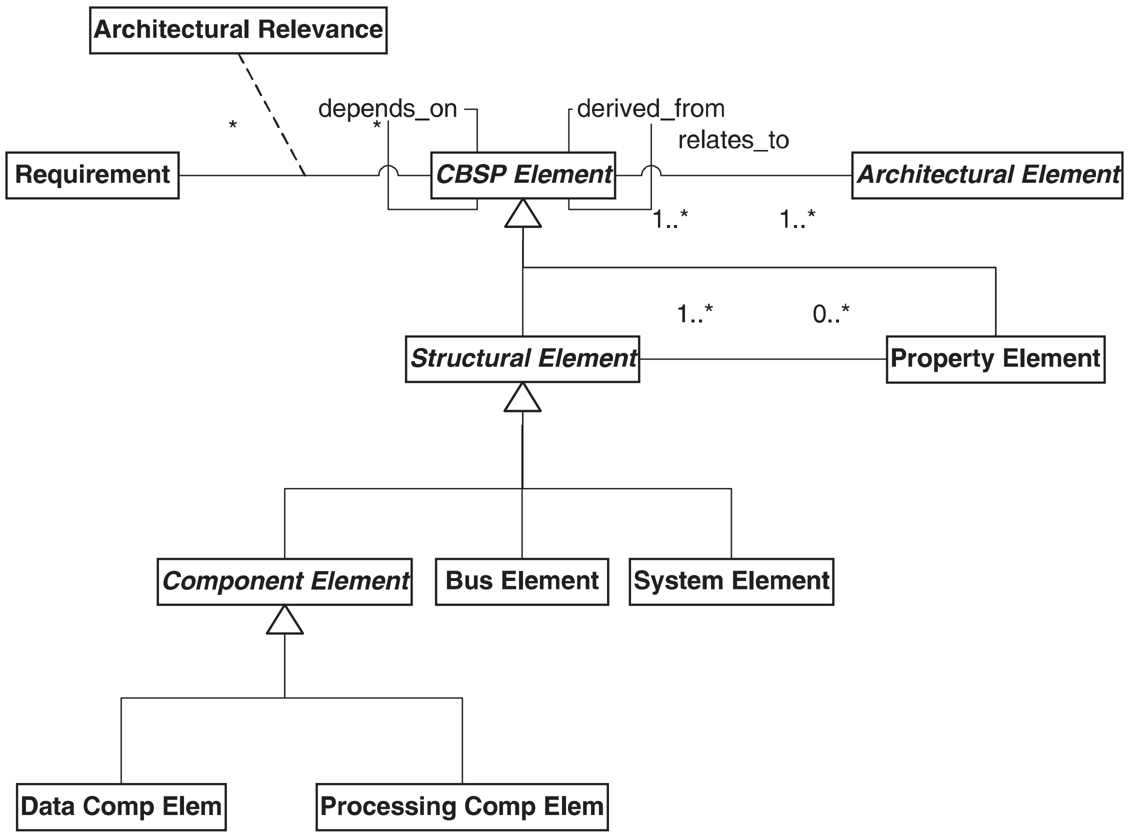


Fig. 2. CBSP Meta Model

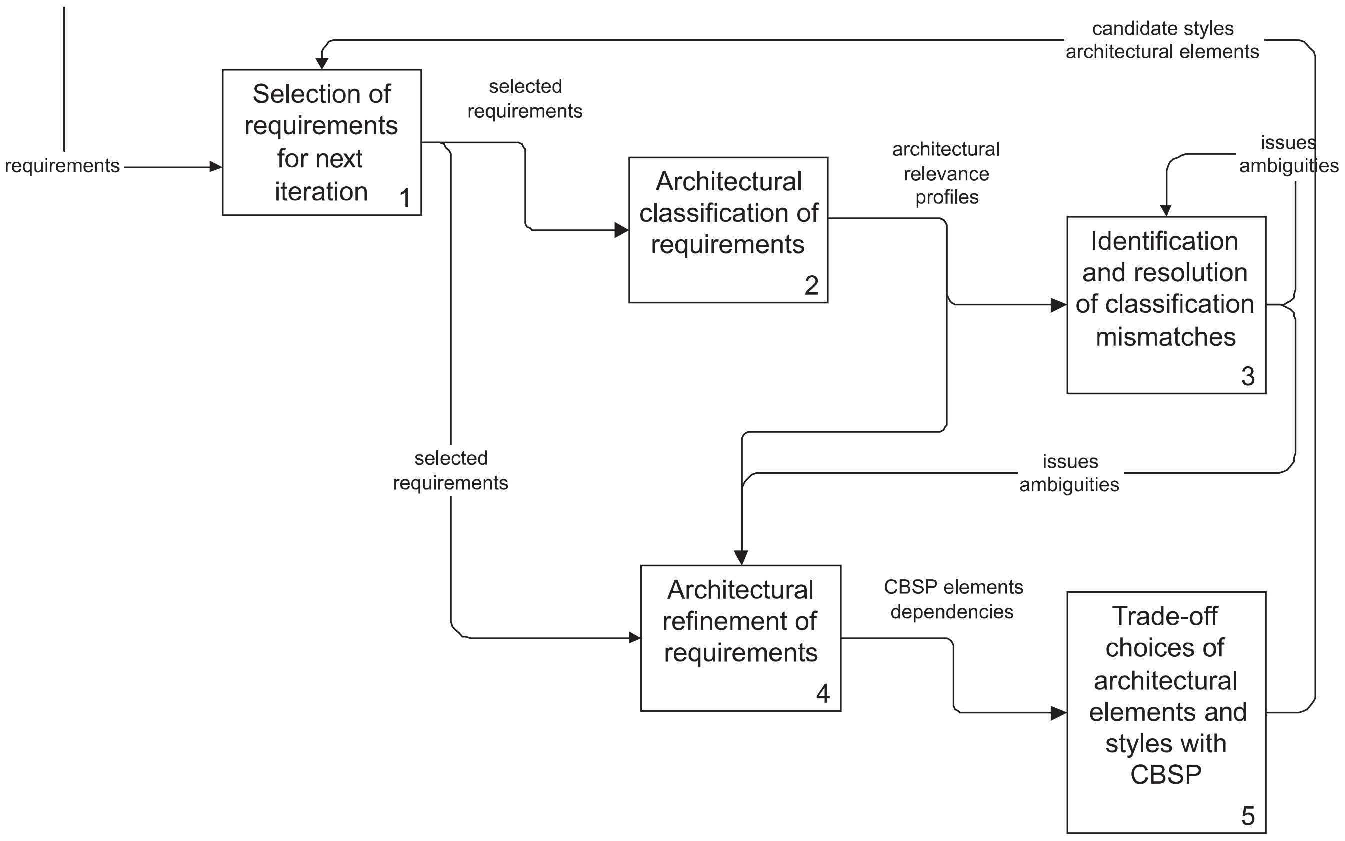

requirements
Selection of requirements for next iteration 1
selected requirements
Architectural classification of requirements 2
architectural relevance profiles
Identification and resolution of classification mismatches 3
issues ambiguities
candidate styles architectural elements
selected requirements
issues ambiguities
Architectural refinement of requirements 4
CBSP elements dependencies
Trade-off choices of architectural elements and styles with CBSP 5

Fig. 3. CBSP process

need to be satisfied before the task can be considered complete and the output(s) of the task itself.

2.2.1 STEP 1: Selection of requirements for next iteration

To reduce the complexity of addressing large numbers of requirements, a team of architects applies the CBSP taxonomy to the most essential set of requirements in each iteration. The architects eliminate requirements considered unimportant or infeasible through stakeholder-based prioritization, thus arriving at a set of core requirements to be considered for the next level of refinement (see Table 1). Stakeholders rate each requirement for each of two criteria [6]: (1) importance shows the relevance and value to project success; while (2) feasibility addresses perceived technical, economic, or political constraints on implementing a requirement.

2.2.2 STEP 2: Architectural classification of requirements

A team of architects classifies the selected requirements using the CBSP taxonomy (see Table 2). Each requirements is assessed by the experts based on the requirement's relevance to the CBSP dimensions, using an ordinal scale (not=0; partially=1; largely=2; fully=3). For instance, a requirement that is rated as partially relevant along the component (C) dimension implies that it has some (partial) impact on one or more architectural components.

A profile showing the aggregated architectural relevance (e.g., C/B/S/CP/BP/SP) is created for each requirement. Figure 4 shows some examples of relevance profiles.

2.2.3 STEP 3: Identification and resolution of classification mismatches

If multiple architects independently perform an architectural classification of requirements using CBSP, their findings may diverge since they may perceive the same statement differently. Revealing the reasons for diverging opinions is an important means of identifying misunderstandings, ambiguous requirements, tacit knowledge, and conflicting perceptions [20]. The voting process is as a mechanism to reveal dissent among the architects and to reduce risks in requirements refinement (see Table 3). The measured consensus among the architects serves as a proxy for their mutual understanding of a requirement's meaning and their agreement on the architectural relevance of a requirement. We determine the level of consensus through Kendall's coefficient of concordance, a measure of the association among stakeholders' ratings [47].

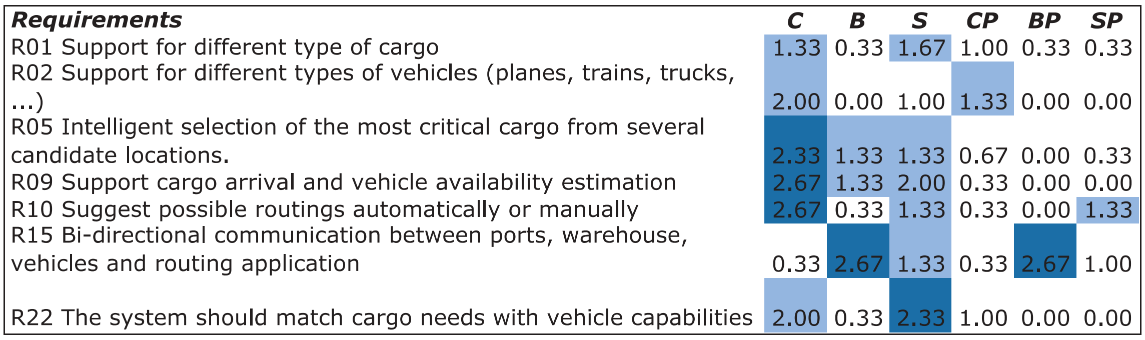
The rules in Table 4 indicate how to proceed in different situations: in case of consensus among architects, the requirements are either accepted or rejected

Table 1. ETVX cell for step 1

	Selection of requirements for next iteration
E	Initial set of requirements (in informal or semi-formal notation) Prioritization method Voting tool
T	All success-critical stakeholders eliminate unimportant and infeasible requirements
V	Check selection of stakeholders Check level of consensus among stakeholders to initiate discussions in case of diverging opinions
X	Set of requirements for next-level CBSP refinement

Table 2. ETVX cell for step 2

	Architectural classification of requirements
E	Set of requirements for next-level CBSP refinement CBSP taxonomy Voting tool
T	Architects classify selected requirements using the CBSP taxonomy
V	Check selection of architects Check completeness of classification
X	Voting ballots Architectural relevance profiles for all requirements



Requirements	***C***	***B***	***S***	***CP***	***BP***	***SP***
R01 Support for different type of cargo	1.33	0.33	1.67	1.00	0.33	0.33
R02 Support for different types of vehicles (planes, trains, trucks, ...)	2.00	0.00	1.00	1.33	0.00	0.00
R05 Intelligent selection of the most critical cargo from several candidate locations.	2.33	1.33	1.33	0.67	0.00	0.33
R09 Support cargo arrival and vehicle availability estimation	2.67	1.33	2.00	0.33	0.00	0.00
R10 Suggest possible routings automatically or manually	2.67	0.33	1.33	0.33	0.00	1.33
R15 Bi-directional communication between ports, warehouse, vehicles and routing application	0.33	2.67	1.33	0.33	2.67	1.00
R22 The system should match cargo needs with vehicle capabilities	2.00	0.33	2.33	1.00	0.00	0.00

Fig. 4. Relevance profiles

Table 3. ETVX cell for step 3

	Identification and resolution of classification mismatches
E	Voting ballots Architectural relevance profiles for all requirements
T	Architects discuss reasons for diverging opinions for low-consensus items Architects update requirements to address issues and ambiguities Architects exclude architecturally irrelevant requirements
V	Check dependencies among requirements to make sure critical requirements are not dropped
X	Issues and ambiguities Architecturally relevant requirements

Table 4. Concordance/relevance matrix

	Relevance	
Concordance	≥ Largely	< Largely
Consensus	**Accept**	**Reject**
Conflict	**Discuss**	

based on the voted degree of architectural relevance. We accept requirement as architecturally relevant if the mean of all stakeholder is at least "largely", otherwise the requirement is rejected. If the stakeholders cannot agree on the relevance of a requirement to the architecture, they further discuss it to reveal the reasons for the different opinions. This discussion process may also involve customers and other stakeholders to clarify a requirement and eases the subsequent step of refining it into one or more architectural dimensions.

2.2.4 STEP 4: Architectural refinement of requirements

In this activity the team of architects rephrases and splits requirements that exhibit overlapping CBSP properties and concerns (see Table 5). Each requirement passing the consensus threshold (concordance and at least largely relevant) may need to be refined or rephrased since it may be relevant to several architectural concerns. For instance, if a requirement is largely component relevant, fully bus relevant, and largely bus property relevant, then splitting

Table 5. ETVX cell for step 4

	Architectural refinement of requirements
E	Issues and ambiguities Architecturally relevant requirements
T	Architects rephrases and splits requirements that exhibit overlapping CBSP properties Architects eliminate redundancies
V	Check to make sure that redundancies are minimized
X	CBSP elements Dependencies among CBSP elements

it up into several architectural decisions using CBSP will increase clarity and precision.

During this process, a given CBSP artifact may appear multiple times as a by-product of different requirements. For example, the following two requirements result in the identification of a *Cargo* data component.

R01: Support for different types of cargo.
R09: Support cargo arrival and vehicle estimation.

Such redundancies are identified and eliminated in the intermediate CBSP model. It is also possible to merge multiple related CBSP model elements and converge on a single artifact (e.g., *R09_$C_{d:}$ Vehicle* in Fig. 5).

This step of the process may be undertaken using two slight variations. The first variation involves identifying only the structural aspects of the system, i.e., its C, B, and S elements. Then, for each of these elements, their properties are identified in a separate (sub)step. This has the advantage of separating the two concerns (structure and functionality vs. non-functional system aspects) and allowing an architect to focus more clearly on the "big picture". The second variation is to identify, in a single step, both the system's structural elements and their non-functional properties. This has the advantage of streamlining the process and allowing the architect to focus on a single system concern (or small set of concerns) at a time. Both variations can be combined in one project. For example, the team of architects may perform a single iteration using the first variation followed by iterations using the second variation.

2.2.5 STEP 5: Trade-off choices of architectural elements and styles with CBSP

At this point, requirements should have been refined and rephrased into CBSP model elements in such a manner that no stakeholder conflicts exist and all model elements are at least largely relevant to one of the six CBSP dimensions. Based on simple CBSP model elements, a "proto-architecture" can be derived.

Architectural styles [1, 30, 40, 46] provide rules that exploit recurring structural and interaction patterns (referred to as "architectural patterns") across a class of applications and/or domains. A style guides the architectural design of a system, with the promise of desirable system qualities. At the same time, the rules guiding the selection and application of a style (or of specific architectural patterns suitable in that style) are typically semi-formal at best, requiring significant human involvement. Furthermore, multiple architectural styles may appear to be (reasonably) well suited to the problem at hand, requiring additional work to select the most appropriate style. This issue is further discussed below.

Based on the dependencies among the elements in CBSP, the rules of the selected style allow us to compose them into an architecture. In other words, we select the

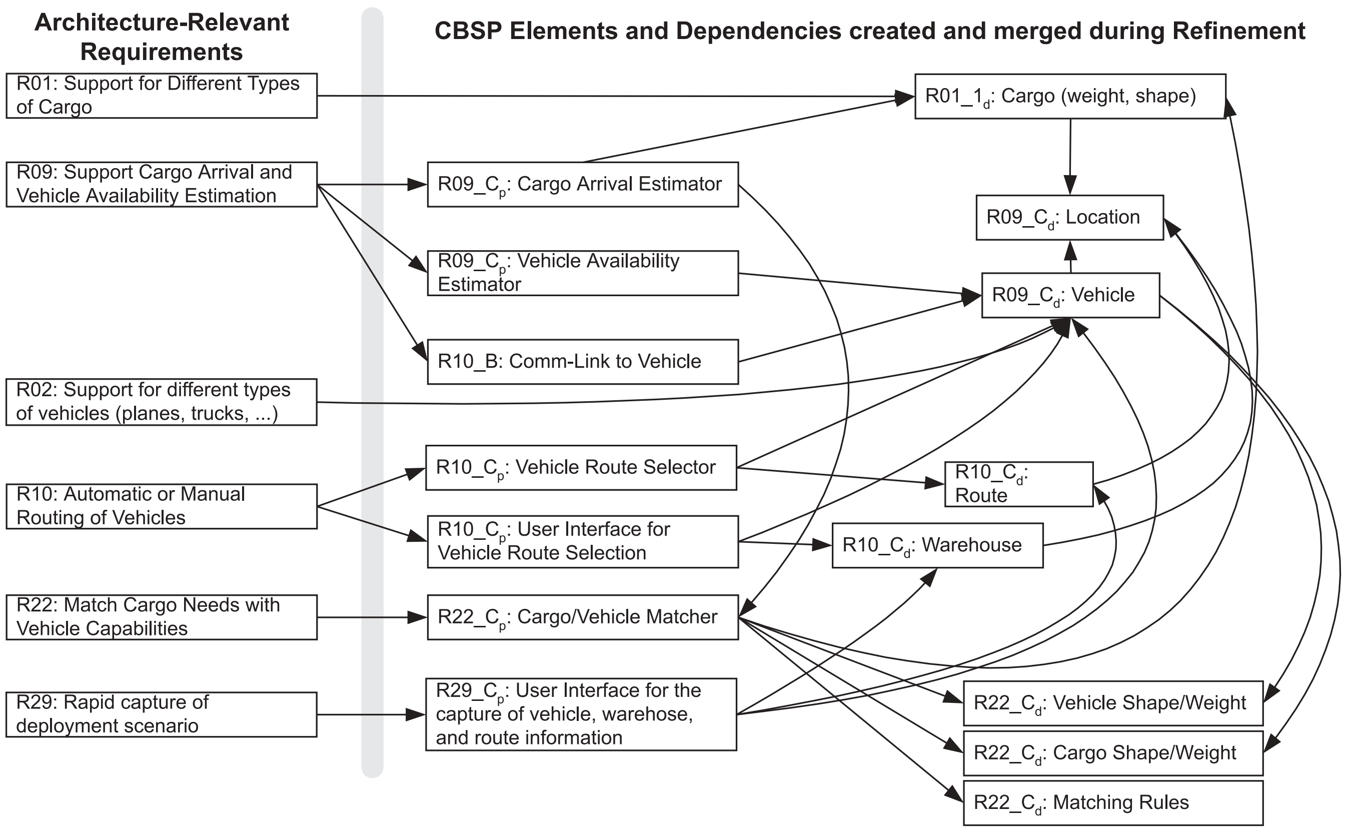

Architecture-Relevant Requirements
CBSP Elements and Dependencies created and merged during Refinement
R01: Support for Different Types of Cargo
R09: Support Cargo Arrival and Vehicle Availability Estimation
R02: Support for different types of vehicles (planes, trucks, ...)
R10: Automatic or Manual Routing of Vehicles
R22: Match Cargo Needs with Vehicle Capabilities
R29: Rapid capture of deployment scenario
R01_1d: Cargo (weight, shape)
R09_Cp: Cargo Arrival Estimator
R09_Cd: Location
R09_Cp: Vehicle Availability Estimator
R09_Cd: Vehicle
R10_B: Comm-Link to Vehicle
R10_Cp: Vehicle Route Selector
R10_Cd: Route
R10_Cp: User Interface for Vehicle Route Selection
R10_Cd: Warehouse
R22_Cp: Cargo/Vehicle Matcher
R29_Cp: User Interface for the capture of vehicle, warehose, and route information
R22_Cd: Vehicle Shape/Weight
R22_Cd: Cargo Shape/Weight
R22_Cd: Matching Rules

Fig. 5. Relationships between requirements and CBSP model in cargo router example

Table 6. ETVX cell for step 5

	Trade-off choices of architectural elements and styles with CBSP
E	CBSP elements Dependencies among CBSP elements Property to Style Mapping Table
T	Architects identify candidate styles Architects identify candidate architectural elements Analyze properties of CBS elements
V	Reconcile CBSP elements with architectural elements to avoid incompatible solutions
X	Candidate styles Candidate components Candidate connectors

style based on (1) the characteristics of the application domain and (2) the desired properties of the system, identified in the requirements negotiation and elaborated in the CBSP model (see Table 6). This can, of course, result in multiple candidate styles (or no obvious candidates). The selection of the style is choice of the architects. By considering the rules and heuristics of the selected style(s) the architects start converting the CBSP model elements into components, connectors, configurations, and data, with the desired properties.

The specific procedure we are using for selecting the suitable architectural style(s) is derived influenced by our previous work [12, 31, 32, 34], from the Architect's Automated Assistant (AAA) approach [12, 16], as well as the work of other researchers [14, 16, 45] on characterizing ad classifying architectural styles. AAA supports rapid evaluation of architectural options with respect to recurring stylistic concerns (i.e., architectural *properties*) such as concurrency, distribution, loci of control, layering, reentrance, latency, data transfer types, and so forth. In addition to these recurring properties (which map to CBSP's P dimension), we augment the support provided by AAA by decomposing architectural styles along five concerns (which account for CBSP's C, B, and S dimensions): data, behavior, interaction, structure, and composition rules. Our expanding classification of styles, which grew out of our classification of software connectors [33], captures known styles in terms of these five concerns. Together, AAA and the style classification provide a significant aid to the architect in further refining CBSP model elements into architectures that adhere to specific styles.

The observation that guides our selection of styles is that, while individual architectural elements and their composition in an architectural topology are important, it is the *properties* of those elements and the topology that guide the selection of a given style.

An example set of properties for data components, processing components, buses, and (sub-) systems is specified as the rows of Table 7. For example, we may be interested whether a given data component is *cached*; we may also like to know whether a given connector is *asynchronous*. Note that Table 7 is clearly not complete; many other properties for each CBSP element are possible. However, the goal is not to produce a complete such table, for two reasons: First, a complete table may be infeasible since it would constitute a complete characterization of *any* software system. Second, the goal of the table is to help us identify the appropriate architectural style. As such, it needs to be only as comprehensive as necessary to narrow down the choice of the appropriate style based on the properties in question; if the table proves to be insufficient in helping an architect determine a suitable style in a given project, it is simply expanded to include additional properties.

Table 7 shows a characterization of five architectural styles (client-server, C2, event-based, layered, and pipe-and-filter) based on a set of example properties. The styles are characterized in detail by Fielding [14]. In this paper, we only briefly summarize them:

- Client-server architectures involve synchronous call-based interactions between service requesting clients and service providing servers. The clients are aware of the server's identity and location, but not the other way around. The clients may communicate only with the server, but not with other clients.
- C2 architectures involve asynchronous event-based interactions among components that may be both service providers (i.e., servers) and requesters (i.e., clients) simultaneously. Connectivity among components is controlled by a set of layering rules and explicit, event-broadcasting connectors.
- Event-based architectures also involve event-based interactions among components. These interactions may be synchronous or asynchronous. Each component may provide or request services. Events are dispatched by special-purpose (meta-level) facilities in the system.
- Layered architectures involve synchronous call-based interactions between components that are divided into layers. A component in a given layer may only invoke components in the layer below it, and is invoked by the components in the layer above it.

Table 7. CBSP to style mapping

CBSP Dimensions	*Properties*	*Client-Server*	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
Data Component	aggregated	++	++	++	+	−
	persistent	++	o	o	o	o
	streamed	−	−	−	−	++
	cached	++	+	−	−	−
Processing Component	service provide/consume only	++	o	o	o	o
	has N interfaces	++	+	++	−	−
	stateful	+	++	++	+	−
	Loose coupling	+	+	++	−	++
	can be migrated	+	++	++	−	−
Connector/bus	synchronous	++	−	+	++	−
	asynchronous	−	++	++	−	++
	local	−	++	o	++	+
	distributed	++	++	++	−	+
	secure	+	o	o	+	o
(sub)System	efficient	o	+	+	o	−
	scalable	+	o	−	−	+
	evolvable	++	++	++	−	++
	portable	o	+	o	++	o
	reliable	o	o	−	o	o
	dynamically reconfigurable	+	++	++	−	++

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

- Pipe-and-filter architectures involve asynchronous interactions involving the exchange of untyped streams of data between components (filters) via connectors (pipes).

For each of these styles, we have identified the degree to which they satisfy the properties identified in Table 7. These (partial) characterizations of the five styles will be used in the example in the next section to drive the selection of the appropriate style(s) and, subsequently, architecture(s) in an example application. Both additional styles and additional properties should be added to the table as the need for them arises.

The resulting architectural elements, their dependencies, properties, and styles give the building blocks for developing an architecture. At this point, the nature of software systems demands the involvement of software architects in order to reconcile CBSP artifacts and package them into an effective architecture (see Sect. 3.5).

3 The Cargo Router case study

This section illustrates the development of an intermediate CBSP model in the context of a case study in which we applied the CBSP approach. Our example application is developed in collaboration with a major U.S. software development organization. It addresses a scenario in which a natural disaster results in extensive casualties and material destruction. In response to the situation, an international humanitarian relief effort is initiated, causing several challenges from a software engineering perspective. These challenges include efficient routing and delivery of large amounts of material aid; wide distribution of participating personnel, equipment, and infrastructure; rapid response to changing circumstances in the field; using existing software for tasks for which it was not intended; and enabling the interoperation of numerous, heterogeneous systems employed by the participating countries. In particular, our system (called *Cargo Router*) must handle the delivery of cargo from delivery ports (e.g., shipping docks or airports) to warehouses close to the distribution centers. Cargo is moved via vehicles (e.g., trucks and trains) selected based on terrain, weather, accessibility and other factors. Our system must also report and estimate cargo arrival times and vehicle status (e.g., idle, in use, under repair). The primarily responsibility of the system's user is to initiate and monitor the routing of cargo through a simple user interface.

We have performed a thorough requirements, architecture, and design modeling exercise to evaluate CBSP in the context of this application. We used EasyWinWin to identify and negotiate requirements for the Cargo Router system. EasyWinWin is a groupware-supported methodology [4] based on the WinWin approach [3] that enhances the directness, extent, and frequency of stakeholder interactions. EasyWinWin adopts a set of COTS groupware components (e.g., electronic brainstorming, categorizing, voting, etc.) developed at the University of Arizona and commercialized by GroupSystems.com.

The stakeholders jointly elaborated 81 requirements. In a first step, the team converged on 64 requirements (79% of all requirements) by reviewing and reconciling similar or redundant ones. For instance, one stakeholder asked to "track location of vehicles" whereas another asked for "the system [to] enable real-time status reports and updates on ports, warehouses, vehicles, and cargo." Obviously, both stakeholders pursued similar requirements which were merged (after verifying stakeholder consensus) into the more general requirement "Support for real-time communication and awareness."

3.1 Selection of requirements for next iteration

The 64 requirements became the initial baseline for the Cargo Router project and covered functional and non-functional aspects, system interfaces, software process aspects, as well as time and budget constraints. To identify the set of requirements needed for a first CBSP model in the first iteration, we performed a joint prioritization of the 64 requirements and selected the 25 (39%) requirements with the highest importance and feasibility.

3.2 Architectural classification of requirements

To surface and extract the architecturally relevant information from the pool of 25 core requirements, a voting process was initiated. Four *architects* classified each requirement with respect to its architectural relevance along the six CBSP dimensions (C/B/S/CP/BP/SP). With four stakeholders involved, a total of 600 votes were cast. This classification process was carried out in less than one hour.

For instance, the requirement "Support cargo arrival and vehicle availability estimation" was voted to be strongly component-relevant by all architects, whereas the requirement "The system must be operational within 18 months" was not voted to be architecturally relevant. This does not mean that process and business aspects are, in general, unrelated to a system's architecture. For example, a schedule constraint may cause architecturally-relevant requirements to be dropped or relaxed. Some requirements also received contradictory votes: the requirement "Automatic routing of vehicles" was voted component relevant (C_p and C_d) by all architects, but system property (SP) relevant by only one architect. At the same time, there was a high degree of consensus on other requirements. For example, since architects voted the requirement "Support cargo arrival and vehicle availability estimation" to be only component relevant (*largely* and *fully* ratings), this requirement was accepted as architecturally relevant without further discussions.

3.3 Identification and resolution of classification mismatches

Ambiguities in the requirements' meanings led to several conflicts when the architects contradicted one another in rating the architectural relevance of requirements. For instance, the architects disagreed on the system property (SP) relevance of the requirement "Automatic routing of vehicles," and cast ballots for *not*, *partial*, and *full* architectural relevance. The concordance matrix in Table 4 suggests that in such a situation stakeholders need to discuss the mismatch. In this particular case, the discussion revealed different perceptions of what this requirement implied. One stakeholder thought this requirement implied that the system needs to suggest paths that vehicles travel (e.g., via navigation points), but not their sources and targets. Another stakeholder perceived the requirement such that the system would also need to suggest the sources and destinations for vehicles. The discussion clarified this conflict and an instant re-vote identified this requirement as indeed system property relevant.

From the total of 150 decisions (= 25 requirements × 6 dimensions), 43 decisions (29% of all decisions) affecting 19 requirements (30% of all requirements) turned out to be controversial and needed further attention. After all conflicts were resolved, 19 out of the original 25 requirements remained in the pool of architecturally relevant requirements.

3.4 Architectural refinement of requirements

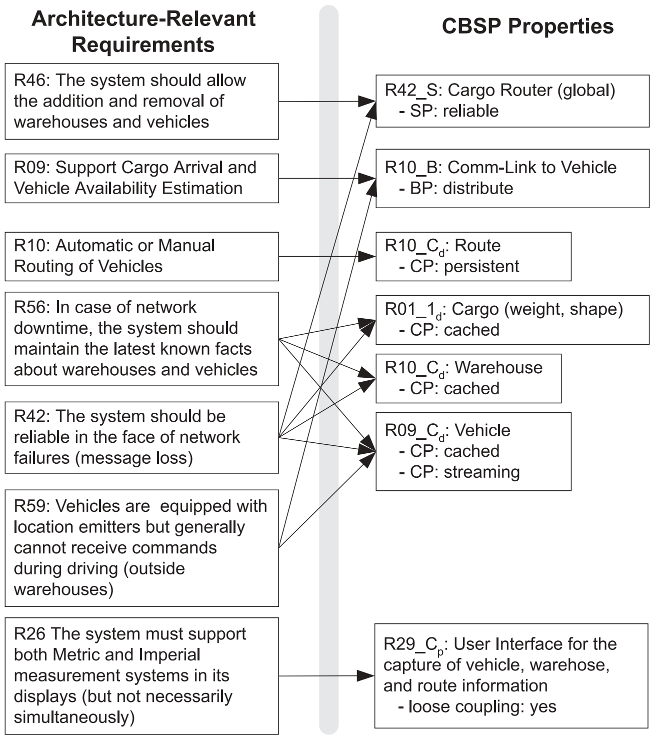
Architecturally relevant requirements explicate at least one CBSP dimension. Often requirements address multiple dimensions. For instance, the requirement "Match cargo needs with vehicle capabilities" was voted to be only processing component (C_p) relevant, whereas the requirement "Support cargo arrival and vehicle availability estimation" was voted to be fully component relevant, fully system relevant, and largely bus relevant. As such, the latter requirement was much more comprehensive than the former. In fact, the latter requirement even depended on the former. For instance, the need for special types of vehicles (e.g., to ship liquid substances) also has an impact on delivery time. In order to better relate such requirements, it is necessary to refine them into more atomic entities. For instance, the fact that cargo arrival estimation depends on vehicle capabilities does not imply that the former requirement fully depends on the latter.

CBSP dimensions also play an important role in the requirements refinement process. For example, the requirement "Match cargo needs with vehicle capabilities" was determined to be only component relevant. As such, the requirement was analyzed and refined into the processing component *Cargo/Vehicle Matcher*. This processing component requires as input cargo and vehicle information, resulting in its dependency on relevant data components (e.g., *Cargo weight*). Since some of those data components did not exist beforehand, they were also created. As a result, the refinement of this requirement produced two types of information: (1) CBSP model elements (processing and components) describing architectural decisions and (2) links describing dependencies among those model elements. The CBSP model elements provide potential building blocks for the architecture, whereas the CBSP dependencies help to clarify potential control and data dependencies within the architecture.

The refinement process of a more complex requirement is similar but more elaborate. For instance, as discussed above, the requirement "Support cargo arrival and vehicle availability estimation" was determined to be C, B, and S relevant. At a high level, this requirement supports two processing components: *Cargo Arrival Estimator* and *Vehicle Availability Estimator*. *Cargo Arrival Estimator* depends on data components representing *Cargo*, the *Vehicle* carrying the cargo, and the *Location* of the vehicle. *Vehicle Availability Estimator* only depends on the knowledge about the vehicle and its location (but not cargo). The above requirement was also rated bus-relevant. This was the case because the location of a vehicle (and its cargo) is variable as it moves. A connector (bus) is therefore needed to allow the system to track vehicles (recall requirement "Real-time communication and awareness").

Figure 5 shows an excerpt of the CBSP model for the Cargo Router example. It depicts the refinement of six requirements into CBSP model elements. As an example, the figure depicts the requirement "Support for different types of cargo" (left) and shows that it was refined into a simple data component called *Cargo* (right). The requirement "Support cargo arrival and vehicle availability estimation" was more complex and was broken up into several model elements including two processing components and a bus component. The figure shows that those model elements were additionally refined via sub-elements (e.g., *Vehicle*) and dependencies. This additional refinement is not necessary, but can result in useful insights into overall system interdependencies. For instance, we learn that the cargo component is also needed by other requirements, making *Cargo* a centerpiece of the system. Should we later want to remove cargo descriptions from the system, the existing dependencies would allow us to reason about the impact of this removal on other parts of the architecture and/or the requirements. In the context of CBSP, refined model elements can be merged in case of similarity resulting in less duplication but more CBSP artifact dependencies (e.g., we do not create multiple *Cargo* data components). As stakeholders in our case study refined all requirements into CBSP model elements, they identified 27 CBSP model elements across the architecturally relevant requirements.

While refining requirements into structural CBS elements and dependencies, it becomes obvious that not all architecturally relevant information can be expressed this



Architecture-Relevant Requirements
CBSP Properties
R46: The system should allow the addition and removal of warehouses and vehicles
R09: Support Cargo Arrival and Vehicle Availability Estimation
R10: Automatic or Manual Routing of Vehicles
R56: In case of network downtime, the system should maintain the latest known facts about warehouses and vehicles
R42: The system should be reliable in the face of network failures (message loss)
R59: Vehicles are equipped with location emitters but generally cannot receive commands during driving (outside warehouses)
R26 The system must support both Metric and Imperial measurement systems in its displays (but not necessarily simultaneously)
R42_S: Cargo Router (global) - SP: reliable
R10_B: Comm-Link to Vehicle - BP: distribute
R10_Cd: Route - CP: persistent
R01_1d: Cargo (weight, shape) - CP: cached
R10_Cd: Warehouse - CP: cached
R09_Cd: Vehicle - CP: cached - CP: streaming
R29_Cp: User Interface for the capture of vehicle, warehose, and route information - loose coupling: yes

Fig. 6. Identifying properties in architecture-relevant requirements

way. For example, the requirement "Support cargo arrival and vehicle availability estimation" among other elements resulted in a connector "Comm-Link to vehicle" but this requirement also implied a central need to access (potentially) distributed resources. The connector must thus accommodate distributed vehicles; ergo a property of the connector is "distributed." Other requirements infer additional properties. For example, automatic routing of vehicles requires access to some predefined set of routes (or navigation points). These need to be persistent; hence, the persistent property for the data component *Route*. Multiple requirements may infer properties on single structural CBS elements. For example, the requirement "In case of network downtime, the system should maintain the latest known facts about [. . .] vehicles" implies vehicles to "cache" data. Furthermore, the requirement "Vehicles are equipped with location emitters and generally cannot receive commands during driving" implies vehicles to accept "streamed" data (see Fig. 6).

3.5 Trade-off choices of architectural elements and styles with CBSP

CBSP model elements, their dependencies, and their properties are valuable for architecture and requirements trade-off analyses. They are also useful in creating and modifying architectural representations. Components and buses typically relate to architectural elements directly, or they may encapsulate sets of architectural elements. For example, the architecture of the Cargo Router will likely have a *Vehicle* component and a *Cargo Arrival Estimator* component. Even dependencies derived during the CBSP refinement are likely to infer architectural data and/or control dependencies. For example, the *Cargo Arrival Estimator* component was found to depend on vehicle information. This dependency indicates a data dependency in the architecture.

CBSP properties likely will not translate into architectural elements directly. Instead, they will imply constraints on architectural elements. The most obvious constraints on architectural elements are imposed through architectural styles. Properties can thus be used to infer or restrict the types of architectural styles applicable to the Cargo Router case study. Table 8 lists properties identified above and their ability to satisfy a given architectural style.

A handful of properties does not allow an architect to derive an architectural style for a complex software system. However, architectural styles differ in their ability to satisfy certain properties. It follows that one can favor some architectural styles over others if the styles favor the desired properties. As foreshadowed above, the current treatment of architectural styles is still largely informal. For this reason, we do not believe that styles can be selected and composed automatically from requirements. Instead, this is the responsibility of the architect(s). However, CBSP can recommend styles based on how well they satisfy the properties of a given system. If an architectural style is consistently unsuitable for a given set of CBS properties then clearly this style is a bad fit. Table 8 lists the properties from the Cargo Router system in the context of the five example styles introduced in Sect. 2 and depicted in Table 7. An architect may infer from this table that the layered style is clearly less suitable than the client-server style for architecting this system. Not surprisingly, none of the given styles is a perfect fit. For example, the client-server style is suitable for all but the streaming property in the Vehicle data component. This does not necessarily imply that this style is unsuitable for the system, only that the style needs to be somehow adapted to accommodate this property. The client-server style is clearly better suited than the layered style; however, the C2 and the pipe-and-filter styles are reasonable alternatives to client-server, depending on what properties are deemed most important.

CBSP gives the architect the necessary input for deciding how best to design and implement a given system in a manner that is easier to understand than the original requirements. It is then still the architect's responsibility to build the system. Both C2 and client-server were considered for the Cargo Router system in our case study, even though the system was eventually implemented according to the C2 style. Figure 7 depicts the resulting architectures in both styles and shows potential solutions to alleviate the style shortcomings. The solution on the left of Fig. 7 implements the Cargo Router system using the C2 architectural style primarily. We find the *Estimator* component placed underneath the *Vehicle* component in the C2 architecture following C2's rules. C2 requires service-providing components to be put above service-requiring components. The CBSP model elements discussed previously are either directly represented in C2 or are grouped together into C2 elements (e.g., *Vehicle* data components, *Estimator* processing components). The only CBSP artifact not represented in C2 is *Cargo*, which is defined using C2's ADL [30] (not depicted here). The C2 architectural style dictates a strict request/response behavior of components. It is thus not very suitable to model the streaming behavior required for vehicles. This problem was solved by implementing the C2 vehicle component using the pipe-and-filter style. Internally, *Vehicle* is thus able to receive streaming data

Table 8. Architectural styles for cargo router

CBSP Dimensions	*Properties*	*Client-Server*	*C2*	*Event-Based*	*Layered*	*Pipe-and-Filter*
Route (data component)	persistent	++	o	o	o	o
Warehouse (data component)	cached	++	+	−	−	−
Vehicle (data component)	streamed	−	−	−	−	++
	cached	++	+	−	−	−
User Interface for Vehicle, Warehouse, and Route (processing component)	loose coupling	+	+	++	−	++
Comm-Link to Vehicle (connector)	distributed	++	++	++	−	+
System	dynamic reconfigure	+	++	++	−	++
	reliable	o	o	−	o	o

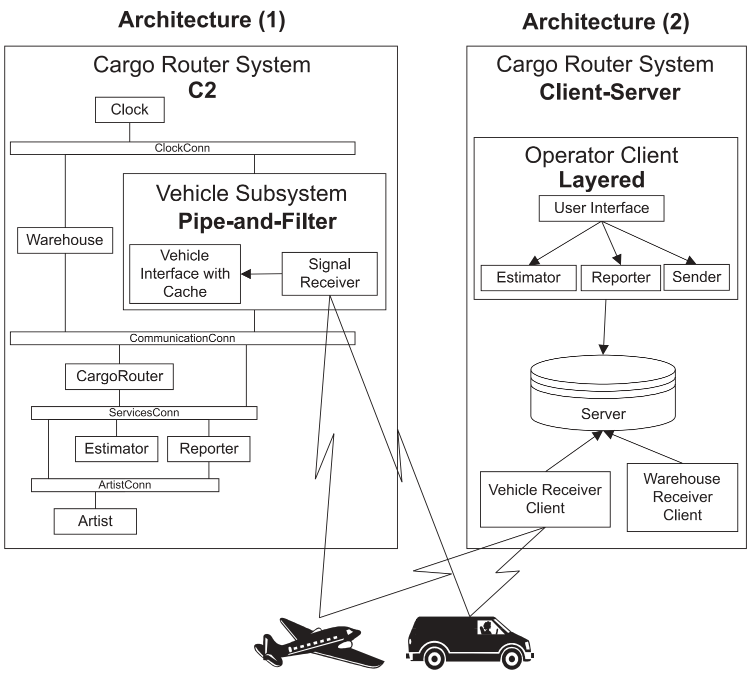


Fig. 7. Two cargo router architectures consistent with CBSP results

from vehicles while externally it appears to be a normal C2 component. The combined system thus implements all given CBS elements, dependencies, and properties despite the imperfect fit of individual styles.

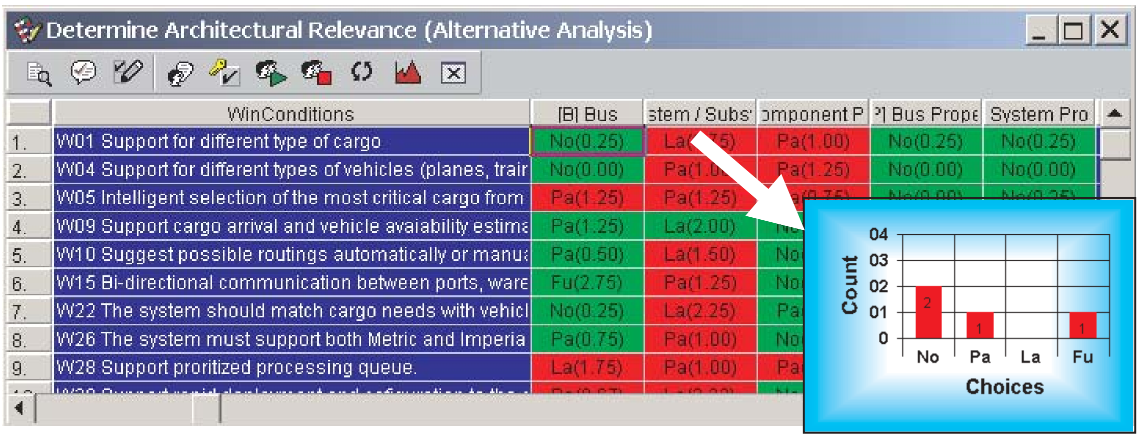
The solution on the right of Fig. 7 implements the Cargo Router system primarily using the client-server style. Instead of "hiding" the signal receiving element as in the C2 solution, this solution models the receivers as clients. Clients are standalone applications that interact with servers. In this solution, the server is primarily a database to store persistent and cached data. The actual application used by operators of the Cargo Router system is implemented through a separate client that communicates with the same data server. Vehicle and warehouse data is thus readily availably without requiring the operator client to handle streaming data. It is interesting to note that the overall client-server architecture addresses most of the properties required by the Cargo Router: clients can be distributed, they are loosely coupled, they can be reconfigured dynamically, and they support persistent data. The internals of the *Operator Client* thus can be implemented in the simple and efficient layered architectural style that previously seemed so unsuitable. This solution, like the previous one, demonstrates that by carefully architecting a system, one can accommodate a variety of properties that may even appear contradictory at times.

4 Tool support

Our ultimate goal is to provide tool support for the CBSP approach. This section discusses tools we have devised to support the CBSP approach. We have adopted off-the-shelf components from GroupSystems.com's groupware suite and we have developed a bridge to the COTS modeling tool Rational Rose to ease the transition of requirements into architecture and allow integration of this work with our existing requirements [20], architecture [30], and design tools.

Selection of requirements for next iteration. This activity is supported as part of the EasyWinWin approach. Stakeholders use a distributed voting tool to assess requirements for their importance and feasibility [6]. Voting is used to determine priorities of requirements and to reveal conflicts, mismatching perceptions, or hidden assumptions. Stakeholders rate each requirement for each of two criteria: Importance shows the relevance of a requirement to project success; feasibility indicates perceived technical or economic constraints of implementing a requirement. In the voting process developers typically focus on technical issues, while clients and users concentrate on the relevance. Requirements are automatically grouped in one of four categories: "low hanging fruit" (high importance, low expected difficulties), "important with hurdles" (high priority, difficult to realize), "maybe later" (low-priority, maybe considered later as easy to realize), and "forget them" (unimportant, difficult to achieve).

Architectural classification of requirements. This step is fully tool-supported by a COTS voting tool. The CBSP dimensions are assessed in a voting process involving multiple experts. Figure 8 depicts voting results (architectural relevance profiles) showing requirements as *not relevant (No)*, and *partially (Pa)*, *largely (La)*, or *fully (Fu) relevant*.

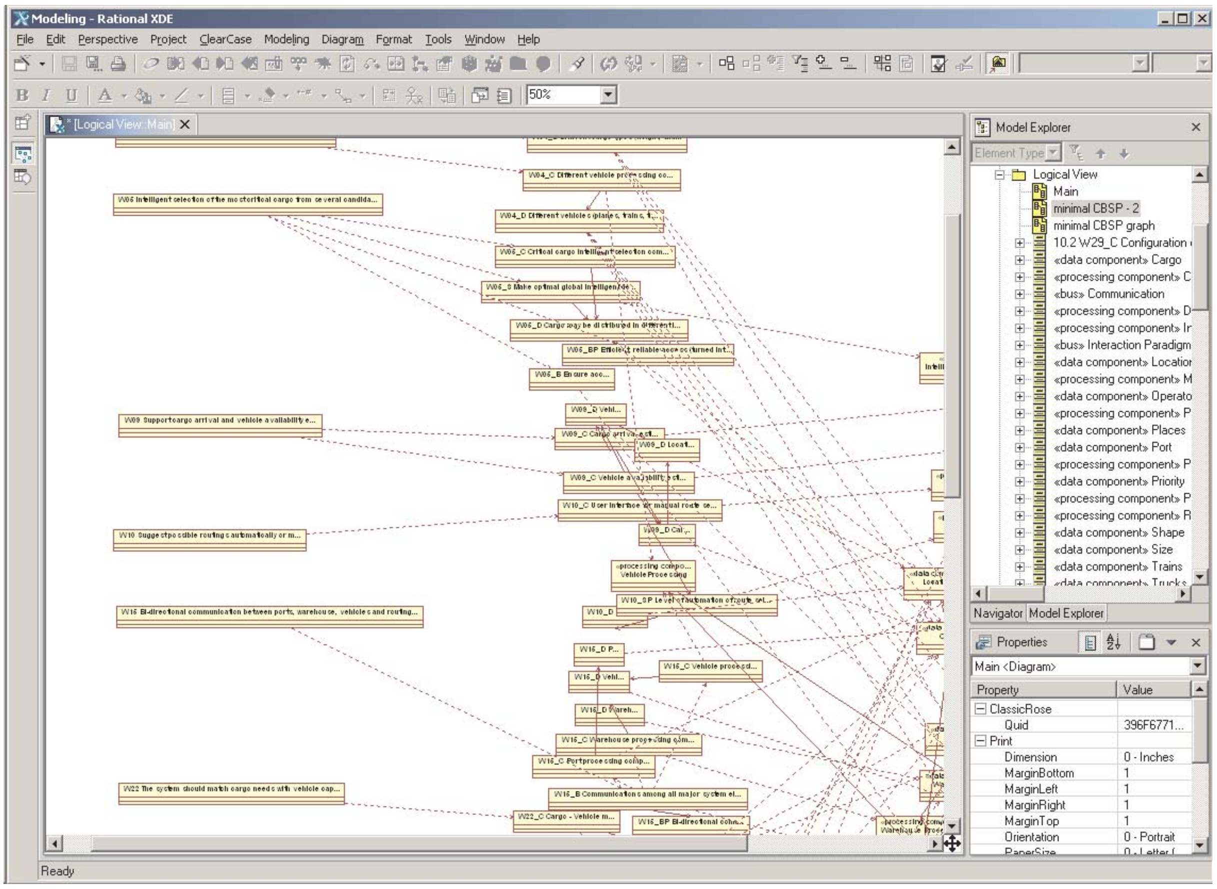


	WinConditions	[B] Bus	stem / Subs	omponent P	P] Bus Prope	System Pro
1.	W01 Support for different type of cargo	No(0.25)	La([illegible]75)	Pa(1.00)	No(0.25)	No(0.25)
2.	W04 Support for different types of vehicles (planes, trair	No(0.00)	Pa(1.0[illegible]	Pa(1.25)	No(0.00)	No(0.00)
3.	W05 Intelligent selection of the most critical cargo from	Pa(1.25)	Pa(1.25)	[illegible]a(0.75)	No(0.00)	No(0.25)
4.	W09 Support cargo arrival and vehicle avaiability estima	Pa(1.25)	La(2.00)	No[illegible]	[illegible]	[illegible]
5.	W10 Suggest possible routings automatically or manua	Pa(0.50)	La(1.50)	No[illegible]	[illegible]	[illegible]
6.	W15 Bi-directional communication between ports, ware	Fu(2.75)	Pa(1.25)	No[illegible]	[illegible]	[illegible]
7.	W22 The system should match cargo needs with vehicl	No(0.25)	La(2.25)	Pa[illegible]	[illegible]	[illegible]
8.	W26 The system must support both Metric and Imperia	Pa(0.75)	Pa(1.00)	No[illegible]	[illegible]	[illegible]
9.	W28 Support proritized processing queue.	La(1.75)	Pa(1.00)	Pa[illegible]	[illegible]	[illegible]

Fig. 8. CBSP classification and conflict detection

Identify and resolve classification conflicts. We also provide tool support for identifying and resolving classification mismatches. This is achieved by highlighting conflicting opinions and perceptions. Different cell colors indicate the level of consensus among the experts. Consensus is indicated with green (light gray in Fig. 8), while disagreement is indicated with red (dark gray in Fig. 8). The vote spread can be displayed (small window in Fig. 8) to trigger discussions about differences in opinion.

Architectural refinement of requirements. A prototype interface to a UML modeling tool is provided to support repository-based integration and refinement of requirements. A bridge from the GroupSystems platform allows translating requirements negotiation results and the CBSP model into a UML representation (see Fig. 9). UML stereotypes are used to extend the modeling capabilities and enable artifact types such as bus property, component, etc. Our tool support supports traceability


Modeling - Rational XDE
File
Edit
Perspective
Project
ClearCase
Modeling
Diagram
Format
Tools
Window
Help
50%
[Logical View::Main]
W05 Intelligent selection of the most critical cargo from several candida...
W09 Support cargo arrival and vehicle availability e...
W10 Suggest possible routings automatically or m...
W15 Bi-directional communication between ports, warehouse, vehicles and routing...
W22 The system should match cargo needs with vehicle cap...
Ready
Model Explorer
Element Type
Navigator
Model Explorer
Properties
Main <Diagram>

Logical View
- Main
- minimal CBSP - 2
- minimal CBSP graph
- 10.2 W29_C Configuration
- «data component» Cargo
- «processing component» C
- «bus» Communication
- «processing component» D
- «processing component» Ir
- «bus» Interaction Paradigm
- «data component» Locatior
- «processing component» M
- «data component» Operato
- «processing component» P
- «data component» Places
- «data component» Port
- «processing component» P
- «data component» Priority
- «processing component» P
- «processing component» R
- «data component» Shape
- «data component» Size
- «data component» Trains
- «data component» Trucks

Property	Value
ClassicRose	
Quid	396F6771...
Print	
Dimension	0 - Inches
MarginBottom	1
MarginLeft	1
MarginRight	1
MarginTop	1
Orientation	0 - Portrait
PaperSize	0 - Letter (

Fig. 9. CBSP model in IBM rational's UML modeling tool

between requirements and CBSP artifacts thus easing trace-off analyses and software evolution [11].

Trade-off choices of architectures elements and styles with CBSP. We currently do not provide specific tool support for recommending architectural styles. CBSP model elements and dependencies do constrain the architectural space in a manner that potentially supports this kind of reasoning. We intend to address this area in our future work.

5 Related work

The work described in this paper is related to several areas of research covering requirements, architecture, and model transformation.

To date, we have applied CBSP in the context of the WinWin requirements negotiation approach. WinWin is related to other techniques that focus on capturing and evolving stakeholder goals into requirements [10, 13, 25, 26, 35, 41, 44] (note that goals reflect the objectives of individual stakeholders and may be contradictory). We believe that CBSP would also work with these approaches. Our work on refining requirements complements such processes with a structured transformation technique and tool support. Our approach is also consistent with the PREview approach by Sommerville and Sawyer as it emphasizes multi-perspective requirements engineering as well as conflict detection and resolution [48]. However, other approaches that enable automated refinement of requirements (e.g., [38]) are predicated on a more formal treatment of requirements.

A key issue in transforming requirements into architecture and further software model elements is traceability. Researchers have recognized the difficulties in capturing development decisions across modeling model elements [17]. Gotel and Finkelstein [19] suggest a formal approach for ensuring the traceability of requirements during development. Our approach captures extensive traces thus satisfying many of the needs identified in [17, 19, 43]. CBSP eases capturing of trace links by narrowing the gap between informal or semi-formal requirements and architecture models. Traceability can be created naturally with CBSP by monitoring the refinement process. This is easy and tool supported. We also developed techniques for dealing with traceability between architecture, requirements, and code during software evolution (e.g., [11]).

Within the area of software architectures, two concepts provide guidance for architects in converting system requirements into effective architectures. The first is architectural styles [46], which capture recurring structural, behavioral, and interaction patterns across applications that are in some way related and/or similar. As discussed above, we indeed make extensive use of architectural styles in formulating an architecture from a collection of CBSP model elements. The drawback that our approach inherits from styles is that they are typically collections of design heuristics, requiring extensive human involvement and adding a degree of unpredictability to the task of transforming CBSP model elements into architectures. The second related concept is domain-specific software architecture (DSSA) [49]. A DSSA captures a model of the (well understood) application domain, together with a set of recurring requirements (called reference requirements) and a generic architecture (called reference architecture) common to all applications within the domain. While these DSSA model elements would make the task of arriving at an architecture from CBSP model elements even simpler than by leveraging styles, CBSP does not require the existence of a DSSA, nor is it restricted only to extensively studied application domains.

Several approaches have been proposed to ease bridging requirements and architectures:

The ATAM technique [25] supports the evaluation of architectures and architectural decision alternatives in light of quality attribute requirements.

Nuseibeh [36] describes a twin peaks model that aims at overcoming the often artificial separation of requirements specification and design by intertwining these activities in the software development process. In line with the twin peaks model, the CBSP approach also argues for the iterative, concurrent development of requirements, architectures, and the intermediate CBSP model. The CBSP process describes one iteration in such a development context. The intermediate CBSP model also helps to relate architectural issues and requirements.

Hall *et al.* describe an extension to Jackson's Problem Frames Approach that adopts the idea of the Twin Peaks model supporting the iteration between problem and solution structures [22]. The authors propose to allow architectural structures, services, and artifacts to be considered as part of the problem domain rather than the solution domain. They extend the current model of the machine domain in problem frames to view it as an architectural engine.

Brandozzi and Perry [5] coin the term architecture prescription language for their extension of the KAOS goal specification language [27] towards architectural dimensions.

Bruin *et al.* [7] use an approach called FS-graphs to relate the desired system features (e.g., quality attributes) and solution fragments (e.g., development activities) that effect those features. For example, a desired feature may be "The system must be secure." while the corresponding solution fragment may be "Employ an encryption scheme." FS-graphs use weighted relations and provide a set of operators (AND, OR, XOR) to relate multiple features with multiple solution fragments. The objective of FS-graphs is to allow reuse and composition of solution fragments across systems with similar desired features. As may be seen from the simple example above, unlike CBSP, FS-graphs are not specifically geared to relating a system's quality requirements to its architecture, but instead deal with the solution space at a much higher abstraction level. Bruin *et al.* have recently begun moving their work in the direction of software architectures as part of the QUASAR (QUality-driven Software ARchitecture) project.

Chung *et al.* discuss how architectural properties such as modifiability and performance can be modeled as "softgoals" and how different architectural designs support these goals. Architectural decisions can be traced back to stakeholders [8].

Franch and Maiden apply the *i∗* actor-based modeling approach [9] for modeling software architectures, not in terms of connectors and pipes, but in terms of actor dependencies to achieve goals, satisfy soft goals, use and consume resources, and undertake tasks [15].

Finally, CBSP also relates to the field of transformational programming [2, 28, 39]. The main differences between transformational programming and CBSP are in their degrees of automation and scale. Transformational programming strives for full automation, though its applicability has been demonstrated primarily on small, well-defined problems [39]. CBSP, on the other hand, can be characterized only as semi-automated; however, we have applied it on larger problems and a more heterogeneous set of models, representative of real development situations.

6 Conclusions and further work

We have introduced the CBSP (Component, Bus, System, Property) approach that aims at reconciling software requirements and architectures using intermediate models. The CBSP intermediate model still "looks" like requirements, but already "sounds" like architecture. The use of CBSP in refining requirements allows developers to systematically explore and identify (1) Architectural Elements (Components and Connectors), (2) Architectural Properties (CP, BP, and SP), (3) Architectural Dependencies, and (4) Suitable Architectural Styles. However, the use of CBSP does not result in an architecture directly. As demonstrated on the case study, CBSP merely elicits the architectural building blocks required to architect a given system. It is the responsibility of the architect to then use these blocks to build an effective architecture that satisfies CBSP's architecturally-relevant information and, consequently, the requirements.

The quality of the results derived through CBSP varies depending on the capabilities of the participating developers. An experienced architect is likely to elicit more correct CBSP information than a novice architect. While human error cannot be eliminated in such a software development activity, CBSP is able to deal with it through conflict identification and resolution.

The number of artifacts produced by CBSP is related to the size and complexity of the system. However, the number of CBSP elements is unlikely to exceed the number of architectural elements required to build a system, as CBSP identifies architecturally relevant information only. We believe that CBSP is thus scalable to the extent that the architecture is scalable. Clearly, using CBSP is human intensive. It requires multiple architects to investigate all given requirements and refine them into CBSP artifacts. Nonetheless, this or analogous activity must be performed with or without CBSP; CBSP assists this process by providing guidance and support.

We believe that, although a deliberately simple and lightweight approach, CBSP assists in coping with the challenges discussed in the introduction.

- *Bridging different levels of formality:* CBSP provides an intermediate model that reduces the semantic gap between high-level requirements and architectural descriptions.
- *Modeling non-functional requirements*: CBSP allows architects to identify and isolate "ilities" in requirements at the system level (SP) and architectural-element level (CP, BP), thus improving modeling of non-functional properties.
- *Maintaining evolutionary consistency*: The intermediate model between requirements and architecture produced by CBSP allows specifying more meaningful dependency links that improve evolutionary consistency.
- *Incomplete models and iterative development*: CBSP does not mandate that the requirements be complete. CBSP also allows architects to maintain arbitrarily complex dependencies between a system's requirements and its architecture, thus easing iterations between the two [36].
- *Handling scale and complexity*: CBSP focuses only on the most essential subset of requirements in each iteration and, further, on the subset of those requirements describing architecturally relevant properties. In fact, each activity in the CBSP process results in filtering out requirements or merging multiple requirements into one. Voting is an important mechanism for reducing complexity by increasing focus and allowing to better understand different stakeholder perceptions.

In our future work we intend to extend CBSP in the following directions:

- Although successfully demonstrated in the context of EasyWinWin and C2, we believe that CBSP has potential for wider applicability as it provides a generic framework of bridging requirements into architecture and design. Further validation is needed by exploring CBSP using a wider set of requirements and architecture languages and methods (e.g., the UML).
- We are also aiming at improving the method to better support capturing feedback from architecture modeling to requirements negotiations. Specifically, we are looking at how findings from architectural modeling, simulation, etc. can be captured as CBSP model elements (e.g., bus property issue, system property issue, component option, etc.).

- We are interested in extending ADLs (e.g., C2) to better support modeling of properties captured as granular CBSP model elements. To date, ADLs have been noticeably deficient in this regard [31].
- Another thread of research is to integrate CBSP with approaches to recover architectural models from existing systems by comparing discovered and recovered architectural models [29].
- We are also applying the approach in different projects to find out if CBSP taxonomies can be tailored and optimized to different domains. Three case studies are currently carried out.
- The tools supporting CBSP are only loosely integrated. We will also work to improve the integration and level of support of our current tool set.

Acknowledgements. This research has been supported by the Austrian Science Fund (Erwin Schrödinger Grant 1999/J 1764 "Collaborative Requirements Negotiation Aids"). This material is also based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-00-2-0615 and F30602-00-C-0218. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory or the U.S. Government.

References

1. Batory D, O'Malley S (1992) The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology (TOSEM) 1(4): 355–398
2. Bauer FL, Moller B, Partsch H, Pepper P (1989) Formal Program Construction by Transformations-Computer-Aided, Intuition-Guided Programming. IEEE Transactions on Software Engineering 15(2): 165–180
3. Boehm BW, Egyed A, Kwan J, Port D, Shah A, Madachy R (1998) Using the WinWin Spiral Model: A Case Study. IEEE Computer 7: 33–44
4. Boehm BW, Grünbacher P, Briggs RO (2001) Developing Groupware for Requirements Negotiation: Lessons Learned. IEEE Software 18(3): 46–55
5. Brandozzi M, Perry DE (2001) Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. In: Workshop "From Software Requirements to Architectures" (STRAW'01) at ICSE 2001
6. Briggs RO, Grünbacher P (2002) EasyWinWin: Managing Complexity in Requirements Negotiation with GSS. In: 35th Annual Hawaii International Conference on System Sciences (HICSS'02) – Volume 1. Big Island, Hawaii
7. de Bruin H, van Vliet H, Baida Z (2002) Documenting and Analyzing a Context-Sensitive Design Space. In: Working IFIP/IEEE Conference on Software Architecture (WICSA 3). Montreal
8. Chung L, Gross D, Yu E (1999) Architectural Design to Meet Stakeholder Requirements. In: Donohue P (ed.): Software Architecture, Kluwer Academic Publishers, pp. 545–564
9. Chung L, Nixon BA, Yu E, Mylopoulos J (2000) Non-Functional Requirements in Software Engineering. Kluwer
10. Dardenne A, Fickas S, van Lamsweerde A (1993) Goal-Directed Concept Acquisition in Requirement Elicitation. In: 6th Int. Workshop on Software Specification and Design (IWSSD 6)
11. Egyed A, Grünbacher P (2002) Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In: 17th Int'l Conf. Automated Software Engineering. IEEE CS, Edinburgh
12. Egyed A, Medvidovic N, Gacek C (2000) A Component-Based Perspective of Software Mismatch Detection and Resolution. IEE Software Engineering 147(6): 225–236
13. Egyed AF, Boehm BW (1999) Comparing Software System Requirements Negotiation Patterns. Systems Engineering Journal 6(1): 1–14
14. Fielding RT (2000) Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine
15. Franch X, Maiden NAM (2003) Modeling Component Dependencies to Inform their Selection. In: 2nd International Conference on COTS-Based Software Systems, Springer
16. Gacek C (1998) Detecting Architectural Mismatches During Systems Composition. In: Center for Software Engineering, University of Southern California, Los Angeles, CA 90089
17. Gieszl LR (1992) Traceability for Integration. In: 2nd International Conference on Systems Integration (ICSI 92)
18. Gotel O, Finkelstein A (1995) Contribution structures. In: Second IEEE International Symposium on Requirements Engineering, York, England
19. Gotel OCZ, Finkelstein ACW (1994) An Analysis of the Requirements Traceability Problem. In: 1st International Conference on Requirements Engineering
20. Grünbacher P, Briggs RO (2001) Surfacing Tacit Knowledge in Requirements Negotiation: Experiences using EasyWinWin. In: 34th Hawaii International Conference on System Sciences
21. Grünbacher P, Egyed A, Medvidovic N (2001) Reconciling Software Requirements and Architectures: The CBSP Approach. In: 5th IEEE International Symposium on Requirements Engineering, IEEE CS
22. Hall JG, Jackson M, Laney RC, Nuseibeh B, Rapanotti L (2002) Relating Software Requirements and Architectures using Problem Frames. In: IEEE International Requirements Engineering Conference (RE'02), Essen, Germany
23. IEEE-610 (1990) IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers
24. IEEE-1320 (1999) IEEE Standards Software Engineering: IEEE Standard for Functional Modeling Language – Syntax and Semantics for IDEF0. The Institute of Electrical and Electronics Engineers
25. Kazman R, Barbacci M, Klein M, Carrière SJ, Woods SG (1999) Experience with Performing Architecture Tradeoff Analysis. In: International Conference on Software Engineering. Los Angeles, CA
26. Kunz W, Rittel H (1970) Issues as elements of information systems. Center for Planning and Development Research, Univ. of California, Berkeley
27. Lamsweerde Av, Darimont R, Letier E (1998) Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering 24(11): 908–926
28. Liu J, Traynor O, Krieg-Bruckner B (1992) Knowledge-Based Transformational Programming. In: 4th International Conference on Software Engineering and Knowledge Engineering
29. Medvidovic N, Egyed AF, Grünbacher P (2003) Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery. In: STRAW'03: Second International SofTware Requirements to Architectures Workshop at ICSE 2003 Portland, Oregon. `http://se.uwaterloo.ca/∼straw03/`
30. Medvidovic N, Rosenblum DS, Taylor RN (1999) A Language and Environment for Architecture-Based Software Development and Evolution. In: International Conference on Software Engineering. Los Angeles, CA
31. Medvidovic N, Taylor RN (2000) A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1): 70–93
32. Mehta N, Medvidovic N (2003) Composing Architectural Styles from Architectural Primitives: Proceedings. In: ESEC/FSE, Helsinki

33. Mehta NR, Medvidovic N, Phadke S (2000) Towards a Taxonomy of Software Connectors. In: 22nd International Conference on Software Engineering, Limerick, Ireland
34. Mikic-Rakic M, Mehta NR, Medvidovic N (2002) Architectural style requirements for self-healing systems. In: Proceedings of the Workshop on Self-Healing Systems, Charleston, South Carolina
35. Mullery G (1979) CORE: A Method for Controlled Requirements Specification. In: 4th International Conference on Software Engineering, Munich, Germany
36. Nuseibeh B (2001) Weaving Together Requirements and Architectures. IEEE Computer 34(3): 115–117
37. Nuseibeh B, Easterbrook S (2000) Requirements Engineering: A Roadmap. In: The Future of Software Engineering, Special Issue 22nd International Conference on Software Engineering, ACM-IEEE
38. Nuseibeh B, Kramer J, Finkelstein A (1994) A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. IEEE Transactions on Software Engineering 20(10): 760–773
39. Partsch H, Steinbruggen R (1983) Program Transformation Systems. ACM Computing Surveys 15(3): 199–236
40. Perry DE, Wolf AL (1992) Foundations for the Study of Software Architectures. Software Engineering Notes
41. Potts C, Burns G (1988) Recording the reasons for design decisions. In 10th International Conference on Software Engineering, IEEE Comp. Soc. Press
42. Radice R, Roth N, O'Hara A Jr, Ciarfella W (1985) A Programming Process Architecture. IBM Systems Journal 24(2): 79–90
43. Ramesh B, Jarke M (2001) Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering 27(4): 58–93
44. Robertson S, Robertson J (1999) Mastering the Requirements Process, Addison-Wesley
45. Shaw M, Clements P (1997) A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In: COMPSAC, 21st Int'l Computer Software and Applications Conference
46. Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall
47. Siegel S, Castellan NJ (1988) Nonparametric Statistics for the Behavioral Sciences, McGraw-Hill, New York
48. Sommerville I, Sawyer P, Viller S (1998) Viewpoints for requirements elicitation: A practical approach. In: Third IEEE Internation Conference on Requirements Engineering (ICRE 98), Colorado Springs, IEEE CS Press
49. Tracz W (1995) DSSA (Domain-Specific Software Architecture) Pedagogical Example. ACM SIGSOFT Software Engineering Notes

Paul Grünbacher is an Associate Professor at Johannes Kepler University Linz and a research associate at the Center for Software Engineering (University of Southern California, Los Angeles). He studied Business Informatics and holds a Ph.D. from the University of Linz. Paul's research focuses on applying collaborative methods and tools to support and automate complex software and system engineering activities such as requirements elicitation and negotiation or software inspections. He is a member of ACM, ACM SIGSOFT, and IEEE.

Nenad Medvidovic is an Assistant Professor in the Computer Science Department at the University of Southern California and is a faculty member of the USC Center for Software Engineering (CSE). He received his Ph.D. in 1999 from the Department of Information and Computer Science at the University of California, Irvine. Medvidovic is a recipient of the 2000 National Science Foundation CAREER award. Medvidovic's research interests are in the area of architecture-based software development. His work focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; product-line architectures; architectural styles; and architecture-level support for software development in highly distributed, mobile, resource constrained, and possibly embedded computing environments. He is a member of the ACM, ACM SIGSOFT, and IEEE.

Alexander Egyed received the MS and ME degrees from the University of Southern California, Los Angeles, and the Johannes Kepler University, Linz, Austria, respectively, and received the PhD degree from the University of Southern California, Los Angeles, in computer science in 2000. He is currently a research scientist at Teknowledge Corporation, Marina del Rey, California. His research interests are in software modeling, transformation, analysis, and simulation. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the ACM SIGSOFT.