

Bridging Models across the Software Lifecycle

Nenad Medvidovic¹ Paul Grünbacher² Alexander Egyed³ Barry W. Boehm¹

¹ Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 USA
{nen,boehm}@sunset.usc.edu

² Systems Engineering and Automation
Johannes Kepler University Linz
4040 Linz, Austria
pg@sea.uni-linz.ac.at

³ Teknowledge Corporation
4640 Admiralty Way, Suite 231
Los Angeles, CA 90292 USA
aegyed@ieee.org

ABSTRACT

Numerous notations, methodologies, and tools exist to support software system modeling. While individual models help to clarify certain system aspects, the large number and heterogeneity of models may ultimately hamper the ability of stakeholders to communicate about a system. A major reason for this is the discontinuity of information across different models. In this paper, we present an approach for dealing with that discontinuity. We introduce a set of “connectors” to bridge models, both within and across the “upstream” activities in the software development lifecycle (specifically, requirements, architecture, and design). While the details of these connectors are dependent upon the source and destination models, they share a number of underlying characteristics. These characteristics can be used as a starting point in providing a general understanding of software model connectors. We illustrate our approach by applying it to a system we have designed and implemented in collaboration with a third-party organization.

KEYWORDS

Software model, software requirements, software architecture, software design, refinement, traceability, model connector

1 Introduction

Software engineering researchers and practitioners have developed a plethora of models that focus on different aspects of a software system. These models fall into five general categories: domain, success, process, product, and property models (Boehm and Port, 1999). Numerous notations, methodologies, and tools exist to support models in each category. For example, within the last decade, the heightened interest in software architectures has resulted in several product and property models based on architecture description languages (ADLs), architectural styles, and their supporting toolsets (Medvidovic and Taylor, 2000; Perry and Wolf, 1992; Shaw and Garlan, 1996).

Models are an indispensable tool in software development. They help developers curb system complexity; they also help the many stakeholders in a project convey their concerns to other stake-

holders in a manner that is understandable and that will ensure the proper treatment of those concerns. However, the preponderance of models actually renders the ultimate goal of development—*implementing* dependable software—more difficult in many ways. The reason for this is the *discontinuity* of information across different models. For example, a system’s requirements might be described using use-case scenarios and entity-relationship diagrams, while its design may be captured in class, object, collaboration, and activity diagrams. The problem, then, is twofold:

1. ensuring the consistency of information across models describing the same artifact (e.g., a class instance in object and collaboration diagrams in a design), and
2. ensuring the consistency of information across models describing different artifacts (e.g., use-cases in a system’s requirements and classes in its design).

In both cases, each model provides (different) information in different ways, making it very difficult to establish any properties of the modeled phenomena as a whole.

In principle, this discontinuity among models can be dealt with by employing *synthesis* and *analysis*. Synthesis enables one to generate a new model (e.g., collaboration diagram) from an existing model (e.g., class diagram), while analysis provides mechanisms for ensuring the preservation of certain properties across (independently created) models. Software engineers extensively employ both kinds of techniques. For example, program compilation involves both the analysis of the syntactic and semantic correctness of one model (source code) and the synthesis of another model from it (executable image).

Synthesis and analysis techniques span a spectrum from manual to fully automated. Manual techniques tend to be error prone, while fully automated techniques are often infeasible (Partsch and Steinbruggen, 1983). Furthermore, in some cases one technique (e.g., analysis) is easier to perform than another (synthesis). For this reason, one typically must resort to using some combination of synthesis and analysis techniques of varying degrees of automation when ensuring inter-model consistency.

The focus of our previous work was on identifying and classifying different categories of models (Boehm and Port, 1999) and providing support for specific models *within* each category (e.g., requirements models (Boehm et al., 1998), architecture models (Medvidovic et al., 1999), and design models (Egyed and Medvidovic, 2000)). This paper discusses a set of techniques we have developed to *bridge* the information gap created by such heterogeneous models.

In many ways, we view the problem of bridging heterogeneous models as similar to the one that has recently generated much interest in the software architecture community: a software architecture can be conceptualized as a diagram consisting of “boxes,” representing components, and “lines,” representing component relationships (i.e., connectors); while we may have a more complete understanding of the components, many of the critical properties of a software system are hidden within its connectors (Mehta et al., 2000; Shaw, 1993). Similarly, the individual models produced during a software system’s lifecycle comprise the “lifecycle architecture” boxes; the properties of these individual

models are typically well understood. Much more challenging is the problem of understanding and providing the necessary support for the lines between the boxes, i.e., the model “connectors.”

The work described in this paper focuses on model connectors traditionally associated with the “upstream” activities in the software lifecycle: requirements, architecture, and design. In particular, we have devised a set of techniques for bridging

1. requirements and architecture models,
2. architecture and design models, and
3. different design models, both at the same level and across levels of abstraction.

As this paper will demonstrate, each of the three cases introduces its own issues and challenges. Moreover, for practical reasons, our investigation to date has focused on a limited number of models. Nevertheless, we have been able to successfully develop and combine a set of model connectors that allow us to start with a high-level requirements negotiation and arrive at a low-level application design in a principled manner. In the process, we have developed a novel, light-weight technique for transferring requirements into architectural decisions. We have also introduced a model transformation framework that supports multiple views of a system’s design.

The results outlined above are specific to our approaches to requirements, architecture, and design modeling. However, we have leveraged this experience, along with existing literature on software model transformations, to devise a set of shared principles we believe to be model-independent. In particular, we classify the *properties* of model connectors and *relationships* among individual elements of different models. We illustrate these properties and relationships both via examples drawn from our work and from well-understood software transformation techniques (e.g. compilation).

The remainder of the paper is organized as follows. Section 2 introduces the notion and properties of model connectors. Section 3 outlines the example application we will use for illustration throughout the paper. Sections 4, 5, and 6 briefly introduce the requirements, architecture, and design modeling approaches we developed in the past and used as the basis of this work, and then provide in-depth discussions of the model connectors we have developed for bridging them. Due to the scope of our work and number of model connectors we have developed, at times we are forced to omit some of the techniques’ details and convey their general flavor to the reader instead. Section 7 revisits the general properties of software model connectors we have identified and ties them to the examples discussed throughout the paper. A discussion of related work and conclusions round out the paper. It is important to note that our approach does not assume any particular lifecycle model (e.g., waterfall or spiral) or software development process. The sequential ordering of lifecycle activities implied by the paper’s organization (Sections 4, 5, and 6 in particular) was adopted for presentation purposes only.

2 Connecting the Software Life Cycle

When we speak of models, diagrams, or views, we mean any form of graphical or textual depiction that describes the software system itself and/or decisions about the system made along the way.

Models may be described separately, but they are not independent of one another. Models may be created individually and validated for syntactic and even semantic correctness within a given context. However, models are interdependent because they must somehow reflect the general objectives of the software system under development. Successful modelling thus requires more than generating and validating individual models — it is also about ensuring the consistency of all models with the general objectives of the software system.

This paper discusses ways of bridging information across models. *Connectors* between models satisfy two primary goals:

1. they are able to transform model information (a form of model synthesis) or
2. they are able to compare model information (a form of model analysis).

In both cases, model connectors maintain consistency by helping to transform or compare the information two or more models have in common. When we talk about model transformation and comparison in the context of this work, we really mean “inter-model” transformation and comparison, that is, transformation and comparison between separate models, diagrams, or views with the primary goal of ensuring a common objective. Although this paper discusses various instances of bridging model information across the software life cycle, we must emphasize that the key contribution of this work is not those instances, but rather their combined, collective properties. The most generic property of a model connector is that it re-interprets information. Re-interpretation is a fundamental requirement for model connectors in order to baseline the relationships between models to overcome syntactic and semantic differences between them.

This paper will show that model connectors can have very unique implementations. However, we will also show that there are some common ways of categorizing their differences by using a set of properties. In particular, model connectors may be directional in that one type of model can be transformed into another type of model, but perhaps not vice versa; model connectors may also only be partially automatable or reliable (i.e., “trustworthy”). We will discuss in this paper that some of those properties apply to model connectors directly whereas other properties apply to the modeling elements they bridge. For instance, modeling elements belonging to different models may complement or outright contradict one another. Sometimes, one modeling element may relate to exactly one element in another model (1-to-1 mapping); or the mappings may be more complex (i.e., many-to-many mappings). In creating and validating model connectors, one has to define and analyze these properties. As an illustration of these properties, the next section will introduce an example. The following sections will then outline some connectors between different models developed in the context of this example. We will then revisit the general properties of model connectors.

3 Example Application

We use an example application to illustrate the concepts introduced in this paper. The application is motivated by the scenario we developed in the context of a U.S. Defense Advanced Research Project Agency (DARPA) project demonstration and recently refined in collaboration with a major U.S. soft-

ware development organization. The scenario postulates a natural disaster that results in extensive material destruction and casualties. 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.

We have performed a thorough requirements, architecture, and design modeling exercise to address these concerns. We have also provided a partial implementation for the resulting system (referred to as “cargo router”). This implementation is an extension of the logistics applications discussed in (Medvidovic et al., 1999).

4 Software Requirements Model Connectors

4.1 Modeling Software Requirements

During requirements engineering, the needs, expectations, constraints, and goals of a project’s stakeholders have to be gathered, communicated, and negotiated to achieve a mutually satisfactory solution. We have developed the WinWin approach for collaborative requirements negotiation and successfully applied it in over 100 real-client projects (Boehm et al., 1998; Boehm et al., 2001). WinWin defines a model guiding the negotiation process: stakeholder objectives and goals are expressed as *win conditions*; known constraints, problems, and conflicts among win conditions are captured as *issues*; *options* describe possible alternative solutions to overcome the issues; if a consensus is achieved among stakeholders, *agreements* are created. We have recently enhanced the WinWin approach and have used a COTS groupware environment as its implementation substrate (GroupSystems, 2001). The result, “EasyWinWin,” supports brainstorming, categorization, and prioritization of win conditions, identification and resolution of conflicts, as well as collaborative characterization of application domain properties (Boehm et al., 2001; Grünbacher and Briggs, 2001).

A team of stakeholders used EasyWinWin to gather, negotiate, and elaborate requirements for the cargo router system. In the first round of requirements negotiation the team came up with 64 win conditions, which provided a starting point for further negotiation and architectural refinements. Figure 1 shows a snapshot of the EasyWinWin negotiation tool: WinWin artifacts are organized in a tree and marked with artifact type and stakeholder tags (top pane); a voting tool is used to aid the transformation of software requirements into an architecture, as discussed below (bottom pane).

4.2 Requirements-to-Architecture Model Connector

The relationship between a set of requirements, such as those produced by an EasyWinWin negotiation, and an effective architecture for the desired system is not readily obvious. Requirements largely describe the *problem* to be solved (and *constraints* on its solution), whereas architectures model a

solution to the problem. The terminology and concepts used to describe the two also differ. For example, WinWin deals with *win conditions*, *issues*, *options*, and *agreements*, while architectures deal with *components*, their interactions (i.e., software *connectors* or *buses*), system *topologies*, and *properties* (Shaw and Garlan, 1996). For these reasons, we have investigated principled ways of relating requirements and architecture models and defining a viable architecture that addresses a given set of requirements. Unfortunately, the large semantic gap between high-level, sometimes ambiguous requirements artifacts and the more specific architectural artifacts (e.g., modeled in a formal ADL) often does not allow one to establish meaningful links between them. This section proposes a model connector that remedies the problem and facilitates the bridging of the two models.

We have developed the CBSP (Component, Bus, System, Property) model connector that bridges requirements and architectures. CBSP artifacts refine WinWin's artifacts into architectural decisions. CBSP is a tool-aided, but highly human-intensive technique. Software architects assess the win conditions for their relevance to a system's architecture: its components (i.e., processing and data elements (Perry and Wolf, 1992)), buses (i.e., connectors (Shaw and Garlan, 1996)), overall configuration (i.e., the system itself or a particular subsystem), and their properties (e.g., reliability, performance, and cost). If it is deemed architecturally relevant, a win condition is refined into one or more artifacts in the CBSP model connector. Each CBSP artifact thus explicates an architectural concern and represents an early architectural decision for the system. For example, a win condition such as

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

can be recast into a processing component CBSP artifact

C_p: A Web browser should be used as a component in the system.

and a bus CBSP artifact

B: A connector should be provided to ensure interoperability with a third-party Web browser.

The CBSP dimensions include a set of general architectural concerns that can be applied to systematically classify and refine requirements negotiation artifacts and to capture architectural tradeoff issues and options. There are six possible CBSP dimensions. They are discussed below and illustrated with examples drawn from the cargo router system negotiation.

(1) C – artifacts that describe or involve a Component in an architecture. For example, the win condition

W12: Allow customizable reports, generated on the fly.

is refined into CBSP artifacts describing both processing (C_p) and data (C_d) components

C_p: Report generator component.

C_d: Data for report generation.

(2) B – artifacts that describe or imply a Bus. For example

W30: The system should have interfaces to related applications (vehicle management system, staff availability).

can be refined into

B: Connector to staff and vehicle management systems.

(3) S – artifacts that describe System-wide features or features pertinent to a large subset of the system’s components and connectors. For example

W6: Capability to react to urgent cargo needs.

is refined into

S: The system should deploy automatic agents to monitor and react to urgent cargo needs.

(4) CP – artifacts that describe or imply Component Properties. For example

W44: Client UI should be accessible via a palm-top or lap-top device.

is refined into

CP: The client UI component should be portable and efficient to run on palm-top as well as lap-top devices.

(5) BP – artifacts that describe or imply Bus Properties. For example

W42: Integration of third party components should be enabled without shutting down the system.

is refined into

BP: Dynamic, robust connectors should be provided to enable “on the fly” component addition and removal.

(6) SP – artifacts that describe or imply System (or subsystem) Properties. For example

W6: Operators must be promptly notified of subsystem failures.

is refined into

SP: The system should support real-time communication and awareness.

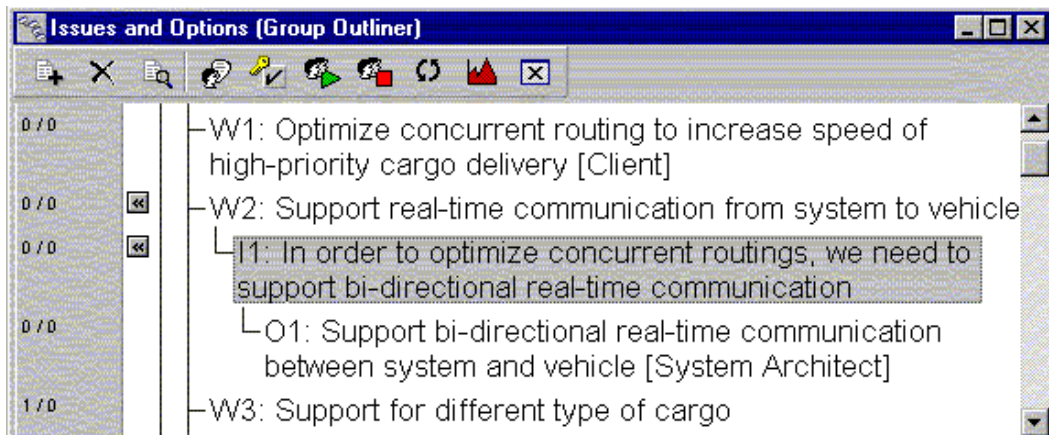
During this process of refining requirements, a given CBSP artifact may appear multiple times as a by-product of different requirements. For example, in the cargo router system requirements negotiation, two win conditions

W1: Optimize concurrent routing to increase speed of high-priority cargo delivery.

and

W3: Support for different types of cargo.

result in the identification of a *cargo* data component (see Figure 2). Such redundancies are identified and eliminated by the CBSP model connector, resulting in a *minimal* (intermediate) CBSP model. During minimization, it is also possible to merge multiple related CBSP artifacts and converge on a single artifact. The minimal CBSP model thus allows architects to maintain arbitrarily complex dependencies between a system’s requirements and its architecture.



The screenshot shows a window titled "Determine Architectural Relevance (Alternative Analysis)". It displays a table with the following data:

	WinConditions	C	B	S	C
4.	W09 Support cargo arrival and vehicl	Fu(2.67)	Pa(1.33)	La(2.00)	No(0.00)
5.	W10 Suggest possible routings autom	Fu(2.67)	No(0.33)	Pa(1.33)	No(0.00)
6.	W15 Bi-directional communication be	No(0.33)	Fu(2.67)	Pa(1.33)	No(0.00)
7.	W22 The system should match cargo	La(2.00)	No(0.33)	La(2.33)	Pa(1.00)
8.	W26 The system must support both M	La(2.00)	Pa(0.67)	Pa(1.00)	No(0.00)
9.	W28 Support proritized processing qu	Pa(1.33)	La(1.67)	Pa(1.00)	Pa(0.67)
10.	W29 Support rapid deployment and c	No(0.33)	Pa(0.67)	La(2.33)	No(0.00)
11.	W37 The system should support vario	No(0.33)	Fu(3.00)	No(0.33)	No(0.00)
12.	W40 The system should have an intui	Pa(1.00)	No(0.00)	No(0.00)	Pa(0.67)
13.	W41 The system should be robust	No(0.00)	Pa(0.67)	No(0.00)	No(0.00)

Figure 1. EasyWinWin negotiation tree and CBSP vote views.

We have developed tool support for identifying and classifying the architectural relevance of win conditions as part of the EasyWinWin environment (recall Figure 1). The CBSP dimensions are applied in a voting process involving multiple experts (e.g., software architects, developers). The experts use the six criteria described above to classify the architectural relevance of each win condition as *unknown*, *not relevant*, and *partially, largely, or fully relevant*. The voting results assist architects in focusing on the relevant subset of the system requirements. The bottom pane of Figure 1 shows a screenshot of the voting tool. Shaded cells in the figure indicate large discrepancies in votes among the experts and reflect potentially confusing win conditions. These win conditions must be discussed, and often reframed, in order to avoid costly errors and misunderstandings.

4.3 Application to the Cargo Router Example

CBSP bridges the requirements and architecture models by providing comprehensible views accessible to both the requirements engineer and the software architect. Figure 2 shows an example of the use of CBSP; it depicts the relationships between partial models taken from the cargo router case study. The *Negotiation Rationale View* shows a set of WinWin artifacts. The *Architectural View* is a possible architecture for the cargo router example further discussed in Section 5. The CBSP model connector comprises two views: the *CBSP View*, created by classifying and refining win conditions, and the *Minimal CBSP View*, created by eliminating replaced and merging related CBSP artifacts.

In the example shown in Figure 2, win condition W1 was voted as being *fully* component relevant, *largely* bus relevant, and *largely* bus property relevant. Win conditions W2 and W4 were voted as being *fully* bus property relevant (omitted from the diagram for simplicity) and *largely* bus relevant. Finally, W3 was voted as being *largely* component relevant. Upon further analysis, it is revealed that W1 describes multiple architectural elements. The two middle diagrams in Figure 2 show the result of this process: W1 is eventually divided into several components, a connector, and a connector property in the minimal CBSP view.

5 Software Architecture Model Connectors

5.1 Modeling Software Architectures

A minimal CBSP view suggests the key architectural elements and their properties for an application. However, it does not provide guidance for achieving an effective topology of those architectural elements: the S and SP categories of architectural decisions provide only hints about the characteristics of the topology. Similarly, in the course of architectural decomposition, the architect may discover that additional components and connectors are needed that have not been identified through requirements elicitation and refinement. For these reasons, the architectural details suggested by CBSP must be complemented with architectural design principles.

There exists a large body of work on arriving at an effective architecture for a given problem. Architectural *styles* (Shaw and Garlan, 1996) provide rules that exploit recurring structural and inter-

action patterns across a class of applications and/or domains. *Domain-specific* software architectures (DSSA) and *product-line* architectures (Perry, 1998) provide generic, reusable architectural solutions (*reference architectures*) for a class of applications in a single domain and instantiate those solutions to arrive at a specific application architecture. Finally, a large body of ADLs and their supporting toolsets (Medvidovic and Taylor, 2000) allow developers to model, analyze, and implement software systems.

In our work to date, we have chosen to use architectural styles as guides in transforming the initial architectural decisions produced by the CBSP model connector into an actual architecture. We have explored the feasibility of composing CBSP artifacts into an architecture according to the Pipe-and-Filter (Shaw and Garlan, 1996), GenVoca (Batory and O'Malley, 1992), Weaves (Gorlick and Razouk, 1991), and C2 (Medvidovic et al., 1999) styles. An analysis of the key requirements for the cargo router system (e.g., scale, distribution, evolvability, heterogeneity) suggested Weaves and C2 as suitable styles. Since our software architecture research is centered around C2 and we had previously applied C2 in the design and implementation of a logistics application, it became our primary choice, as already foreshadowed in Figure 2.

C2 provides a number of useful rules for high-level system composition. A C2-style architecture consists of processing components, buses, and their configurations; data components are treated implicitly, as attributes of the processing components' interactions. For example, in Figure 2 *Cargo* is not explicitly represented in the C2 architecture. C2 imposes a particular topological order on the components and buses in an architecture: components may interact only via buses and may have at most one bus on their top and one on their bottom sides; as a side-effect, topologically adjacent components may not directly interact. Furthermore, each component is substrate-independent and may only have knowledge of the components above it in the architecture.

Based on the dependencies among the elements in the minimal CBSP view, the rules of the C2 style allow us to compose them into an architecture. For example, as shown in Figure 2, *Optimizer* depends on *Vehicle* and *Warehouse*; C2's substrate independence principle mandates that *Optimizer* be placed below them in the architecture. Since there are no direct dependencies between *Vehicle* and *Warehouse*, they may be adjacent. Note that the same dependency relationship would have different topological implications in a different style. For example, GenVoca would require *Optimizer* to be *above* the *Vehicle* and *Warehouse* components (while still allowing *Vehicle* and *Warehouse* to be at the same level). Furthermore, unlike C2, GenVoca would allow direct interactions among its components, without the intervening connectors.

The C2 architecture of a subset of the cargo routing application is shown in Figure 3a. The *Port*, *Vehicle*, and *Warehouse* components maintain the state of the application. *Optimizer* ensures the most efficient distribution of vehicles at the delivery ports, assignment of cargo to the vehicles, and routing of vehicles to the warehouses. *CargoRouter* tracks the cargo during its delivery to a warehouse, while *Reporter* allows progress tracking of the system by a human operator. *SystemClock* provides consistent

time measurement to interested components. Finally, the *Artist* component renders the application's user interface.

C2-style architectures are modeled in an ADL, C2SADEL (Medvidovic et al., 1999). C2SADEL allows modeling of component and connector types, which are then instantiated and composed into a configuration. For illustration, an excerpt of a C2SADEL model of the cargo router architecture is shown in Figure 3b, while a partial specification of the *Port* component type is given in Figure 3c. Such a specification is analyzed for consistency by C2's DRADEL environment (Medvidovic et al., 1999).

5.2 Architecture-to-Design Model Connector

Based on the information provided in a C2SADEL model of an architecture, DRADEL is capable of generating a partial implementation of that architecture (Medvidovic et al., 1999). However, many lower-level issues needed to complete the implementation (e.g., specific data structures and algorithms) are not provided at the architectural level. For that reason, the "outer skeleton" of the application generated from the architectural model must be complemented with the details typically provided through lower-level design activities.

To ensure the traceability of design-level details to the architecture and vice versa, we have developed a model connector that synthesizes a design model from an ADL model. We selected the Unified Modeling Language (UML) (Booch et al., 1998) as the target design language and conducted an in-depth study of the feasibility of mapping several ADLs to UML (Medvidovic et al., 2001).¹ Based on this earlier study, we have implemented a model connector between C2SADEL to UML. The transformation results in an intermediate model that is represented in UML, but reflects the structure, details, and properties of the original architectural model.² The model connector is defined by a set of rules that ensure that every C2SADEL feature is transferred into UML. A preliminary attempt at such a rule set was discussed in (Abi-Antoun and Medvidovic, 1999). We have since refined and completed these rules. Additionally, we have integrated DRADEL with the Rational Rose UML modeling environment (Abi-Antoun and Medvidovic, 1999), allowing fully automated synthesis of UML models from C2SADEL architectures.

A small excerpt of the rule set comprising the model connector between C2SADEL and UML models is shown in Figure 4. It indicates that, for example, a C2 component is modeled in UML as a collection of "stereotyped" UML elements (classes, operations, and attributes). Stereotypes are an extension mechanism provided by UML to enable modeling of constructs (e.g., «*C2-Component*») not originally envisioned by UML's designers. As demonstrated in (Medvidovic et al., 2001), the seman-

1. A more detailed overview of UML is given in Section 6.

2. We should note that the ADL and UML models are not entirely isomorphic. Space limitations prevent us from further elaborating on this issue here. Additional details can be found in (Medvidovic et al., 2001).

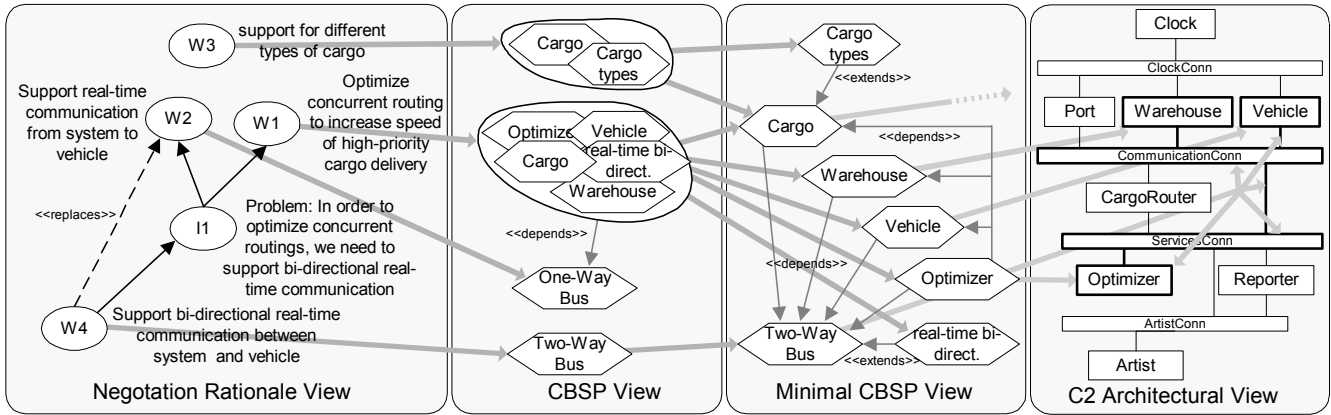


Figure 2. Transforming a requirements model into an architectural model using the CBSP model connector (shown in the two middle diagrams). Grey arrows indicate traceability links between model elements. As discussed further in Section 5.1, an architectural model cannot be directly derived from a (minimal) CBSP model. However, the intermediate CBSP model maps to an architecture in a more obvious way than does a requirements model.

tics of such constructs are specified formally using UML's Object Constraint Language (OCL), which is based on first-order predicate logic (Booch et al., 1998).

5.3 Application to the Cargo Router Example

The C2 architecture of the cargo router application is mapped into several UML diagrams as indicated by the rules in Figure 4. Each C2 component and connector is mapped to a specific set of UML class diagrams, representing its internal details as modeled in C2SADEL, while the overall configuration is mapped to UML component and object diagrams. Figure 5 shows the synthesized (partial) UML view of the cargo router architecture. All the details of the architecture represented in C2SADEL are transferred into this intermediate model.

6 Software Design Model Connectors

6.1 Modeling Software Designs

Our support for software design leverages a large body of mainstream design notations and methodologies, collected into the UML (Booch et al., 1998). UML is a graphical language that provides a useful and extensible set of predefined constructs, it is semi-formally defined, and it has substantial (and growing) tool support. UML allows designers to produce several models of a software system via the supported diagrams: class, object, collaboration, package, component, use case, statechart, activity, sequence, and deployment diagrams. As discussed above, UML allows additional semantic constraints to be placed on its modeling elements via OCL.

Once the intermediate UML model is synthesized from the architecture in the manner discussed in Section 5, that model must be further refined to address the missing lower-level design issues, such as additional processing and data elements, specific data structures, and algorithms. This section discusses model connectors we have developed to bridge related design models (e.g., class diagrams) at different levels of abstraction, as well as different design models (e.g., class and statechart diagrams) at the same level of abstraction.

6.2 Inter-Design Model Connectors

In order to help bridge design models, we have devised a set of design model connectors, accompanied by a set of activities and techniques for identifying inconsistencies among the models in an automatable fashion. We refer to the model connectors, activities, and techniques as a *view integration framework* (Egyed, 2000). The view integration framework identifies and supports two categories of design model transformations: design refinement and design view transformations. *Design refinement* involves bridging between higher-level and lower-level views, while *design view transformations* provide bridges among different system views at the same level of abstraction.

As discussed above, UML supports a wide range of diagrams to model a system. Our view integration framework currently encompasses eight transformations between models expressed in four

different UML diagrams: class, object, sequence, and statechart diagrams. Due to space constraints, we will discuss the general principles of the view integration framework, but will only focus on the details of transformations across class and object diagrams.

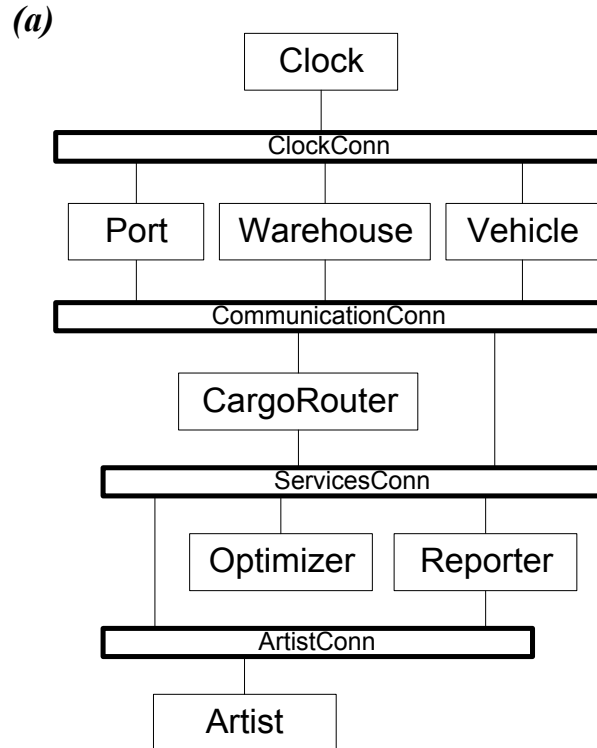
In our investigation of UML diagrams, we have identified three major transformational dimensions (see Figure 6). Views can be seen as *abstract* or *concrete*, *generic* or *specific*, and *behavioral* or *structural* (Egyed, 2000). The abstract-concrete dimension was foreshadowed in Section 5, where a C2 architecture was the abstract view and the generated UML model the concrete view. The generic-specific dimension denotes the generality of modeling information. For instance, a class diagram naturally describes a relationship between classes that must always hold, whereas an object diagram describes a specific scenario. Finally, the behavior-structure dimension takes information about a system's behavior to infer its structure. For instance, test scenarios (which are behavioral) depict interactions between objects (structural) and may thus be used to infer structure.

Manual management of design model connectors across these three dimensions is often infeasible due to the complexity of the models. Two factors contribute to the complexity: (1) the existence of model elements that are only relevant to one view, but not to others (e.g., “helper” classes such as *theWarehouseCollection* in Figure 8d), and (2) the large number of interdependencies between model elements that must be traced and understood (e.g., the grey arrows between elements in the four models shown in Figures 2, 5, 8, and 9). In order to control this complexity, we have developed a tool, UML/Analyzer (Egyed, 2000), that uses an abstraction technique to eliminate helper classes. UML/Analyzer searches for class and object patterns and replaces them with simpler, more abstract patterns of the same type based on a set of over 60 rules. An excerpt of UML/Analyzer's rule set is shown in Figure 7.

For instance, to identify a mismatch in the class diagram shown in Figure 8d, we need to eliminate the helper classes *availableGoods* and *aSurplus* that “obstruct” our view of the relationship between *aVehicle* and *aWarehouse*. In this example, UML/Analyzer sees an aggregation from *aVehicle* to *availableGoods*, followed by a generalization (inheritance) from *availableGoods* to *aSurplus*, which is, in turn, followed by an association from *aSurplus* to *aWarehouse* (Figure 8c). The tool then uses its abstraction rules to replace the class and relationship patterns. Further applying our abstraction rules on the example, we end up finding an association relationship between *aVehicle* and *aWarehouse* (Figure 8a). This example is further discussed below.

6.3 Application to the Cargo Router Example

As already discussed, we will focus on the application of design model connectors to class and object views of the cargo router system. We use the UML model produced by the transformation discussed in Section 5 as our starting point. Figure 9 shows an excerpt of the consistency checking process in the context of cargo router (Egyed and Medvidovic, 2000). The figure depicts a lower-level design (right side) and its intermediate abstraction produced by UML/Analyzer in the manner outlined above (middle). The intermediate model can be compared more easily with the original model (i.e., architecture,



(b) architecture CargoRouteSystem is {
 component_types {
 component Port is extern {Port.c2;}
 component Artist is virtual {}
 ... }
 connector_types {
 connector RegConn is {filter no_filter;} }
 architectural_topology {
 component_instances {
 aPort : Port;
 Display : Artist; ... }
 connector_instances {
 ClockConn, ArtistConn : RegConn; ... }
 connections {
 connector Clock Conn {
 top SimClock;
 bottom aPort; }
 connector ArtistConn {
 top Optim, Report, ServicesConn;
 bottom Display; }
 ... } }
 }

(c) component Port is
 subtype CargoRouteEntity (int \and beh) {
 state {
 cargo : \set Shipment; selected : Integer; ... }
 invariant { (cap >= 0) \and (cap <= max_cap); }
 interface {
 prov ip_selshp: Select(sel : Integer);
 req ir_clktk: ClockTick(); ... }
 operations {
 prov op_selshp: {
 let num : Integer;
 pre num <= #cargo;
 post ~selected = num; }
 req or_clktk: {
 let time : STATE_VARIABLE;
 post ~time = time + 1; }
 ... }
 map {
 ip_selshp -> op_selshp (sel -> num);
 ir_clktk -> or_clktk ();
 ... }
 }

Figure 3. (a) Architectural breakdown of the cargo routing system. (b) Partial cargo routing system architecture specified in C2SADEL. (c) Partial Port component type specified in C2SADEL. “~” denotes the value of a variable after an operation has been performed, while “#” denotes set cardinality. C2SADEL uses a backslash to distinguish a keyword from an identifier with the same name (e.g., “\set” versus “set”).

shown on the left) to ensure consistency. For example, the association relationship between *CargoRouter* and *Vehicle* in the middle diagram is in violation of the original architecture's structure since no corresponding link between the two can be found in the C2 architecture (left diagram).

Another potential mismatch between the two models depicted in Figure 9 is a result of C2's rule that two topologically adjacent components (e.g., *Vehicle* and *Warehouse*) are not allowed to directly interact. The intermediate model again helps to detect that mismatch as shown in Figure 8. The object *aVehicle* is part of *availableGoods*, which, in turn, is a child of *aSurplus*. Since *aSurplus* can only access the object *aWarehouse* (part of another component), it follows that it is possible for *Vehicle* to interact with *Warehouse*—a violation of the original architectural model.

7 Properties of Model Connectors

This paper has presented three classes of model connectors needed at the “upstream” stages of the software lifecycle. Each of the three is different from the others and, in this paper, they have been applied only on our own modeling techniques. At the same time, the model connectors share several characteristics we believe to be more generally applicable (e.g., they all employ intermediate models and a combination of synthesis and analysis). Explicating such characteristics will help software researchers and practitioners to better understand software model connectors; it will also potentially help them in developing their own or adapting existing techniques for bridging software models.

In this section, we discuss several properties of model connectors we have identified to date. We illustrate each property with examples drawn from the model connectors discussed earlier in the paper. The properties can be organized in two major categories:

- properties relevant to a model connector *as a whole*. The identified properties are purpose, directionality, automatability, and reliability.
- properties relevant to the relationships between *individual elements* of the involved models. These specify the nature of traceability links between the elements.

7.1 Purpose

Models are transformed to achieve certain objectives during development. The transformation purpose describes the underlying intent behind a model transformation. Examples of purposes are refinement, mismatch detection, or the creation of a stakeholder-specific (e.g., user) view.

A single model connector often serves several purposes. For example, the main purpose of the CBSP model connector is the refinement of WinWin requirements negotiation artifacts into architectural elements. CBSP also supports analysis indirectly, by capturing architectural trade-offs and mismatches revealed in the process of architectural modeling. Problems detected during architectural modeling and simulation can be captured as CBSP architectural decisions, such as

S: Three seconds system response time not possible due to limited network bandwidth.

7.2 Directionality

We can distinguish between unidirectional and bi-directional model connectors. Unidirectional connectors allow transformation in one direction only. For example, in Section Figure 6 we discussed a unidirectional connector that allows derivation of a model's structural view from its behavioral view. Another common model connector that is typically unidirectional is compilation: it is difficult to derive source code from a compiled image. Another example is CBSP. While it allows feedback from architecture modeling to requirements negotiation via the traceability links it maintains (recall Figure 2), CBSP currently does not make any specific provisions for actually traversing those links, leaving the task to humans and external tools.

Bi-directional model connectors establish a “two-way bridge” between two models. An example of a bi-directional connector is the bridge between C2SADEL and UML: the mapping between a C2SADEL model and the UML model initially generated by the transformation discussed in Section 5 is bijective.

7.3 Reliability

Reliability describes the degree of confidence in a model connector. Reliability depends on the rules that can be established to guide the application of a model connector. We distinguish between informal, semi-formal, and formal rules. While model connectors in the later stages of the life-cycle (e.g., compilation) are typically based on formal rules, connectors that are employed early in the process (e.g., CBSP) depend on heuristics.

For example, transforming a requirements model into an architecture model is heavily influenced by the ambiguity and imprecision of natural language and cannot be considered highly reliable. We have tried to mitigate that in CBSP via guided, expert-based refinements of negotiation results and guidelines for analyzing the vote spread of the experts (recall discussion of Figure 1). The higher degree of formalization of architectural and design models typically renders a model connector between them more reliable. At the same time, in our particular approach to bridging architectures and designs, we faced the problem that several aspects of UML semantics remain informal. DRADEL (Medvidovic et al., 1999) has tried to address this issue by placing formal constraints, specified in OCL, on UML modeling elements (recall Section 5).

7.4 Automation

This property describes the degree to which tools support the rules guiding a model connector. We distinguish between manual, semi-automated, and fully automated support.

To a large extent, the degree of automation depends upon the level of formality of the involved models. For example, the derivation of CBSP artifacts from (informal) requirements is semi-automated using EasyWinWin. On the other hand, the comparatively higher degree of design formalization allows one to build fully automated model connectors between design models. For example, UML/Analyzer (Egyed, 2000) automatically synthesizes intermediate models during a transforma-

tion. These intermediate models are then used to detect structural and behavioral inconsistencies by employing automated comparison (i.e., analysis) techniques. As it can be seen in the context of Figure 8, a series of intermediate models may be generated by a single model connector.

7.5 Element Relationship Properties

The properties discussed thus far characterized model connectors as a whole. We now turn our attention to properties of the relationships between individual elements of different models.

7.5.1 Qualifier

Elements from two models related by a model connector can be unrelated, complementary, redundant, or contradictory. Model connectors use various mechanisms to identify and/or make use of these types of relationships, as indicated by the examples below.

- *Unrelated*: If no relationship is established between two model elements by a model connector, we regard these elements as unrelated. This happens if certain elements of the source model are not refined or the target model deals with different concerns. For example, the CBSP voting process emphasizes architectural relevance, helping the architect to focus on the most relevant subset of the negotiation results and to ignore unrelated artifacts (e.g., a development schedule win condition may have no bearing on a component property CBSP artifact).
- *Complementary*: If a model element completes information provided by another model element we denote that relationship as complementary. For example, the services a C2 component requires are explicit, first-class constructs in C2SADEL and are used as the basis of architectural analysis. In the UML model, these services become a part of system documentation, intended as a guide to the designer.
- *Redundant*: A single model element is often used in multiple models. Relationships among different occurrences of such an element can be qualified as redundant. For example, *VehicleComponent* is represented in the architecture diagram, as well as the Object and Component UML diagrams in Figure 5. Such redundancy is unavoidable when a model connector's source and target models have overlapping concerns. At the same time, the redundancy presents a problem in that changes in one such view must always be propagated to all other views.
- *Contradictory*: The relationship of two or more elements is contradictory if it is impossible for (some subset of) the model properties that depend upon the elements to be valid simultaneously. For example, the architectural model in Figure 9 indicates that no interaction relationship may exist between *Vehicle* and *CargoRouter*, which is contradicted by the design model.

7.5.2 Cardinality

In addition to the qualifier, the cardinality of the relationship between model elements has to be identified. We can distinguish the following relationships. Examples of each relationship can be found in Figure 2.

Component Internal Object → Class
State Variable → Class Private Attribute
Provided Operation → Class Operation
Component → <<C2-Component>> Class
Internal Object → <<C2-Component>> Class Attribute
Component Interface → <<Interface>> Class
Connector → <<C2-Connector>> Class
Architecture Configuration → Object Diagram + Component Diagram
Component/Connector Binding → Object Link (instance of an association)

Figure 4. Partial rule set for transforming a C2SADEL model into a UML model. This rule set is implemented by the integration of DRADEL and Rational Rose.

- *Transmute*: One element of the source model is related to exactly one element in the destination model. An example is a win condition that is related to exactly one component in an architecture.
- *Diverge*: One element of the source model relates to multiple elements in the destination model. An example is a win condition that is refined into a component and a connector in an architecture.
- *Converge*: Multiple elements of the source model are related to one element in the destination model. An example are several win conditions that converge into one component in an architecture.

7.6 Summary

	<i>Model Connector Properties</i>						
	<i>Inter-model properties</i>					<i>Inter-model-element properties</i>	
	<i>Elements of intermediate model</i>	<i>Purpose</i>	<i>Directionality</i>	<i>Reliability</i>	<i>Degree of automation</i>	<i>Qualifier</i>	<i>Cardinality</i>
<i>CBSA (Requirements-to-Architecture)</i>	C _p , C _d , B, S, CP, BP, SP (see Fig. 2)	Refine requirements into architecture; Enable backward traceability between architecture and requirements	Uni-directional	Informal (based on expert judgement)	Semi-automated in EasyWinWin groupware environment	Unrelated Complementary Redundant Contradictory	Transmute Diverge Converge
<i>C2SADEL to UML (Architecture-to-Design)</i>	Class, Class Op., <<C2-component>> class, etc. (see Fig. 4)	Ensure traceability of design-level details to the architecture and vice-versa	Bi-directional	Formal	Fully via the DRADEL to Rational Rose™ Interface	Complementary Redundant	Typically Transmute
<i>Class model abstractor (Inter-Design)</i>	Classes of intermediate models (see Fig. 8)	Identify behavioural mismatch; “zoom out” to reduce complexity	Uni-directional	Formal rules and heuristics	Fully via UML/ Analyzer	Complementary Redundant Contradictory Unrelated	Transmute Converge

The above table shows the model connectors discussed in Sections 4.2., 5.2, and 6.2, together with their properties presented in this Section 2 and revisited above. Describing model connectors in such a way serves several purposes:

1. it helps to better understand existing software development methodologies by characterizing the transitions between the various modelling approaches used,
2. it assists in identifying “missing links” inside methodologies, and
3. provides a roadmap for methodology developers who want to improve their approaches or need to adapt a methodology to specific needs (automation, high reliability, and so on).

8 Related Work

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

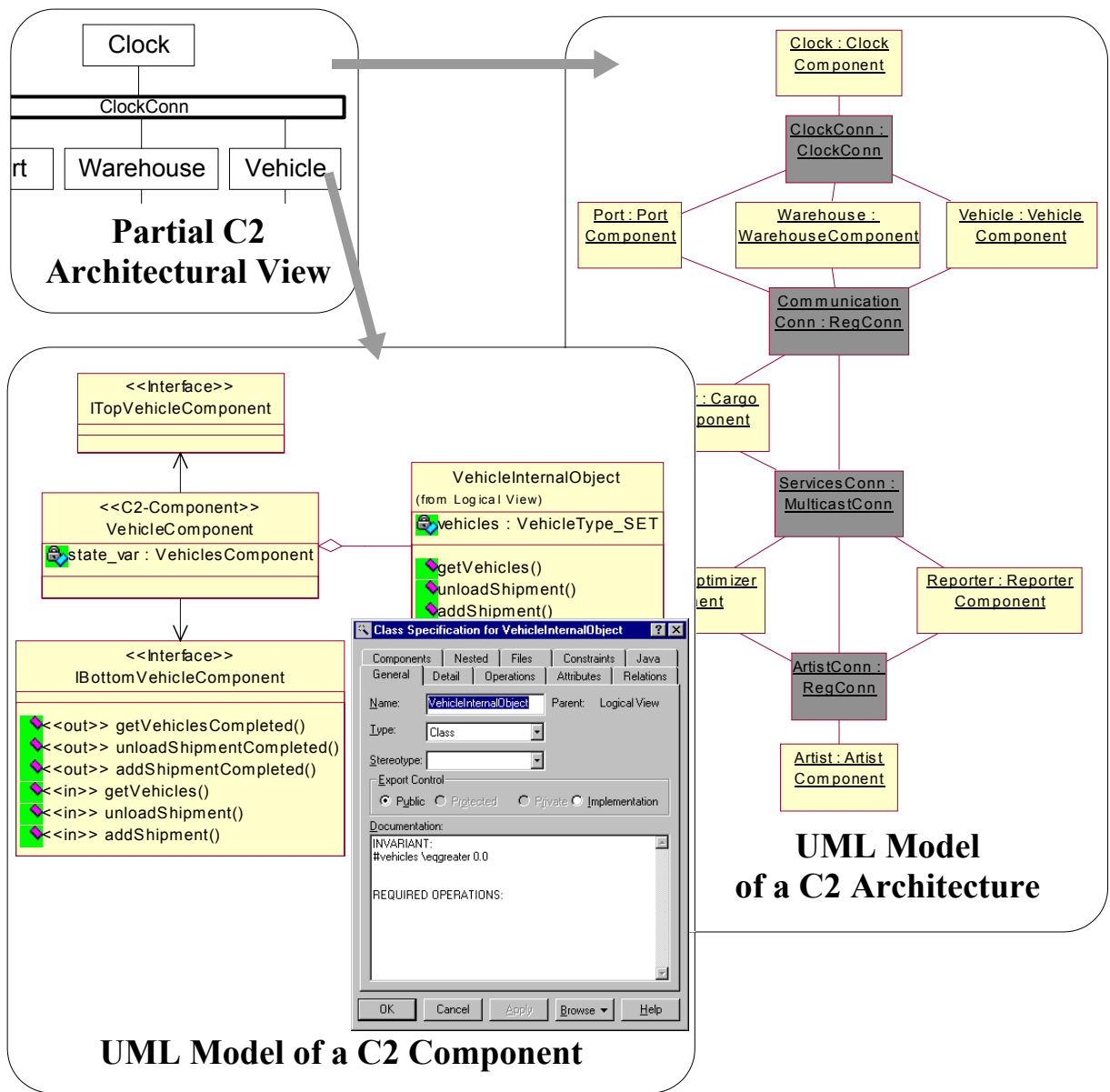


Figure 5. Synthesis of an intermediate UML model from the C2 architectural model shown in Figure 3.

architectures was applied to WinWin, an example of a class of techniques that focus on capturing requirements, their tradeoffs, and their refinements in a structured, but not always formal manner (Chung et al., 1999; Dardenne et al., 1993; Kazman et al., 1999; Mullery, 1979; Robertson and Robertson, 1999). For this reason, even though we have developed and applied our CBSP approach specifically in the context of WinWin, we believe CBSP to be more generally applicable.

The refinement of requirements into architecture and design is often discussed in the context of requirements capture. Generally, those discussions focus on processes (e.g., (Robertson and Robertson, 1999)), but not automatable techniques. Our work on refining requirements extends such a process with a structured transformation technique and tool support. A handful of other approaches exist that, at least in principle, also enable automated refinement of requirements. However, those approaches are predicated on a more formal treatment of requirements artifacts (e.g., (Nuseibeh et al., 1994)) than a technique such as WinWin would allow.

A key issue in transforming requirements into architecture and design is effectively tracing development decisions across modeling artifacts. Researchers have recognized the difficulties in capturing such traces (Gieszl, 1992; Gotel and Finkelstein, 1994). (Gotel and Finkelstein, 1994) suggest a formal approach for ensuring the traceability of requirements during development. Our approach is less formal, but captures extensive trace information throughout the development process, thus satisfying many of the traceability needs defined in (Gieszl, 1992; Gotel and Finkelstein, 1994).

Software architecture researchers have studied the issue of refining an architecture into a design. An approach representative of the state-of-the-art in this area is SADL (Moriconi et al., 1995). SADL incrementally transforms an architecture across levels of abstraction using a series of refinement maps, which must satisfy a correctness-preserving criterion. While powerful, this transformation technique can be overly stringent (Garlan, 1996). It sacrifices design flexibility to a notion of (absolute) correctness. Furthermore, formally proving the relative correctness of architectures at different refinement levels may prove impractical for large architectures and numbers of levels.

Different elements of our model connectors between architectural and UML models can be found in existing work. (Cheng et al., 1995) enable transformations by converting models into a formal environment (e.g., algebraic specification) to allow precise reasoning. Likewise, our approach, defines C2 architectures in UML via formal OCL constraints to allow precise reasoning. Although a formal approach to transformation has a number of advantages, we have found that it is not always suitable or practical. Several of our design model connectors are therefore based on diagrammatic transformations of UML analogous to (Khriess et al., 1998; Koskimies et al., 1998). In fact, we adopted as one of our inter-design model connectors the approach for transforming sequence diagrams to statecharts introduced by (Koskimies et al., 1998).

The work described in this paper also relates to the field of transformational programming (Bauer et al., 1989; Liu et al., 1992; Partsch and Steinbruggen, 1983). The main differences between transformational programming and model connectors are in their degrees of automation and scale. Transformational programming is fully automated, though its applicability has been demonstrated pri-

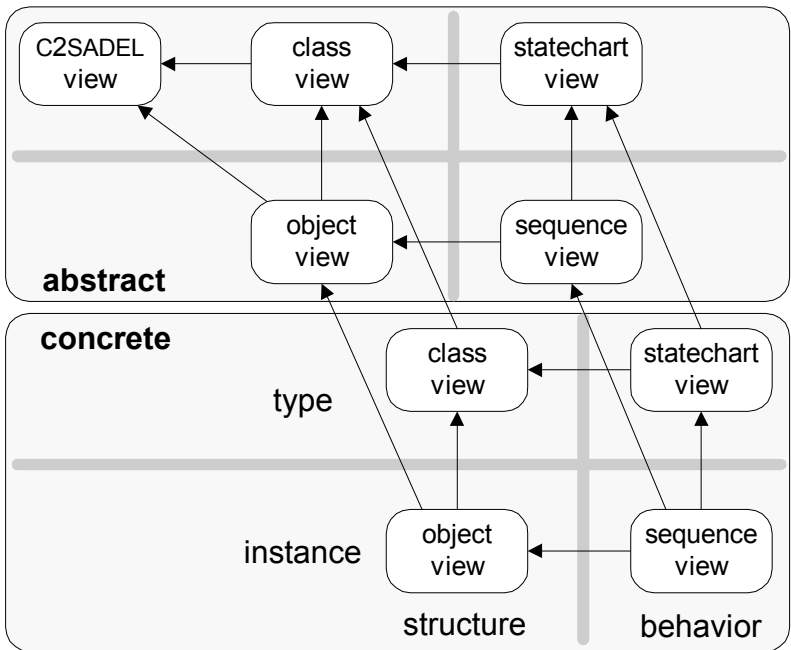


Figure 6. Design model connectors.

Generalization x Class x Generalization → Generalization
Generalization x Class x Association → Association
Generalization x Class x Aggregation → Aggregation
Association x Class x Generalization → Association
Association x Class x Association → Association
Aggregation x Class x Association → Association

Figure 7. Partial rule set used by UML/Analyzer to simplify class and object diagrams. These rules have been created in collaboration with the Rational Software Corporation. Rational also implemented our abstraction method in a tool called Rose/Architect (Egyed and Kruchten, 1999).

marily on small, well defined problems (Partsch and Steinbruggen, 1983). Our approach, 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.

9 Discussion and Conclusion

In this paper, we have discussed a set of model connectors whose ultimate goal is to facilitate the consistent transformation of a system's requirements into its implementation. We believe that this is an important contribution in that our approach provides some novel solutions to a difficult problem, studied extensively by software engineering researchers. For example, the CBSP model connector provides a good balance of the structure and flexibility needed to address the problem of deriving an effective architecture from a system's requirements. System quality requirements in particular tend to drive the choice of architecture (Kazman et al., 1999); at the same time, the "optimal" architecture is often a discontinuous function of the required quality level. Highly formal approaches are typically unable to adequately deal with this discontinuity, while the collaborative CBSP approach can handle it more readily. CBSP addresses the issue by involving experts in a voting process to determine the architectural relevance of negotiation artifacts and to identify incomplete and inconsistent requirements.

Another contribution of this paper lies in its identification of a set of underlying principles needed to enable a series of model connectors: all connectors discussed in this paper rely on the use of intermediate models, the coupling of analysis and synthesis of varying degrees of automation, and a set of shared properties (recall Section 7). While we have developed and applied these principles in the context of specific requirements, architecture, and design modeling approaches, we have taken special care to ensure their broader applicability. Thus, for example, the CBSP approach does not depend on the use of WinWin, but can instead be applied to a wide range of requirements model artifacts. Similarly, we have already applied our ADL-to-UML model connector to several ADLs (Medvidovic et al., 2001). Other well understood software model connectors also appear to adhere to these principles. For example, compilation is a fully automated, typically unidirectional, highly reliable synthesis model connector whose intermediate models include abstract syntax trees.

Our work in this arena continues along several dimensions. The MBASE approach (Boehm and Port, 1999) and its support for multiple model categories is used as the conceptual integration platform for this work. We also integrate the tool support provided by EasyWinWin, DRADEL, and UML/Analyzer to facilitate easier development and implementation of model connectors; we intend to leverage all three tools' existing interfaces to Rational Rose to this end. We are also investigating additional model connectors that will, in particular, allow the use of multiple ADLs to enable architectural modeling of different system characteristics. Finally, we are exploring the suitability of open hypertext engines (Anderson, 1999) for automatically maintaining the numerous traceability links produced by our model connectors, a task our work to date has not supported.

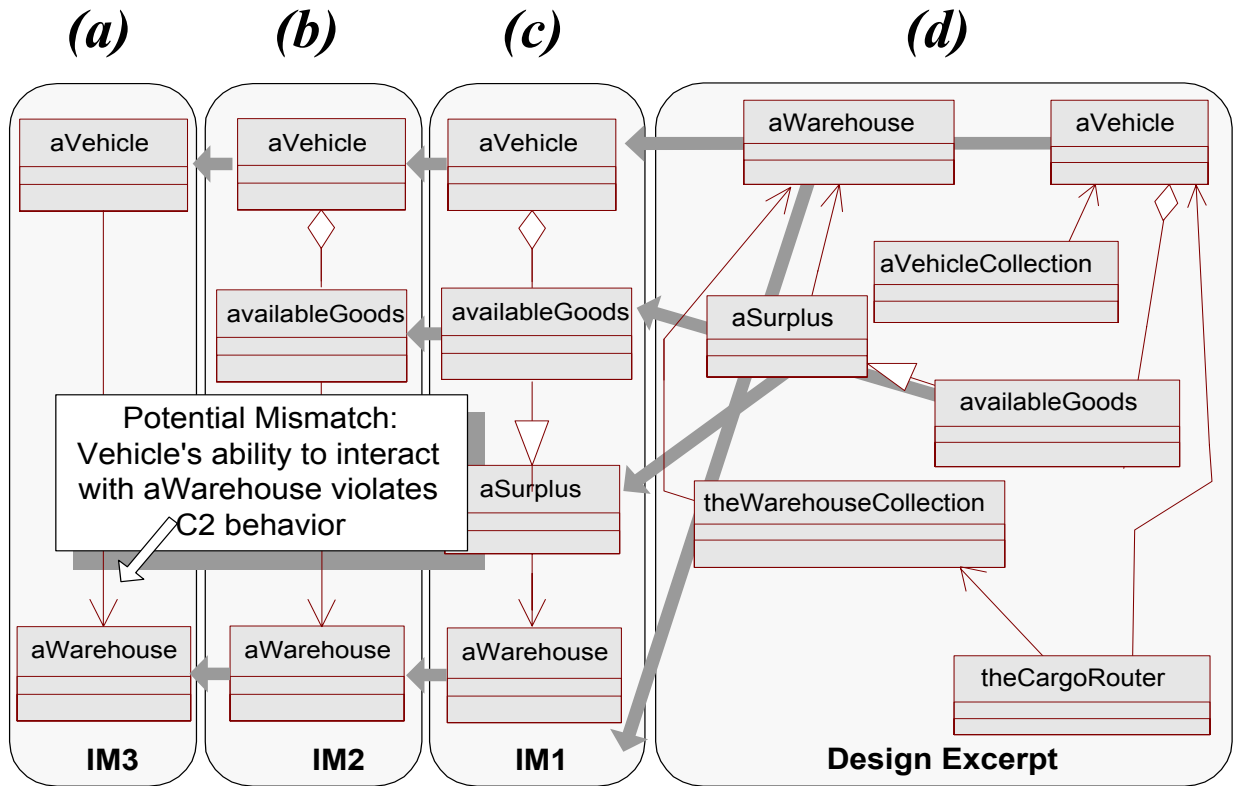


Figure 8. Series of intermediate models (from right to left) produced to identify behavioral mismatch.

10 Acknowledgements

This material is partly based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-99-1-0524, F30602-00-C-0200, F30602-00-C-0218, and F30602-00-2-0615. 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 the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Paul Gruenbacher was supported by a grant from the Austrian Science Fund (1999/J 1764 “Collaborative Requirements Negotiation Aids”).

11 References

- Abi-Antoun, M., and Medvidovic, N. Enabling the Refinement of a Software Architecture into a Design. *UML'99*, Oct. 1999.
- Anderson, K. M. Supporting Software Engineering with Open Hypermedia. *ACM Computing Surveys' Electronic Symposium on Hypermedia*, Dec. 1999.
- Batory, D. and O'Malley, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992.
- Bauer, F. L., Moller, B., Partsch, H., and Pepper, P. Formal Program Construction by Transformations – Computer-Aided, Intuition-Guided Programming. *IEEE Transactions on Software Engineering*, 15(2), Feb. 1989.
- Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., and Madachy, R. Using the WinWin Spiral Model: A Case Study. *IEEE Computer*, 7:33-44, 1998.
- Boehm, B., and Gruenbacher, P. Supporting Collaborative Requirements Negotiation: The EasyWin-Win Approach. *International Conference on Virtual Worlds and Simulation*, San Diego, Jan. 2000.
- Boehm B., Grünbacher P., Briggs B. Developing Groupware for Requirements Negotiation: Lessons Learned. *IEEE Software*, pp. 46-55, May/June 2001.
- Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
- Cheng, B. H. C., Wang, E. Y., Bourdeau, R. H., and Richter, H. A. Bridging the Gap Between Informal and Formal Approaches to Software Development. *Software Engineering Research Forum*, Nov. 1995.

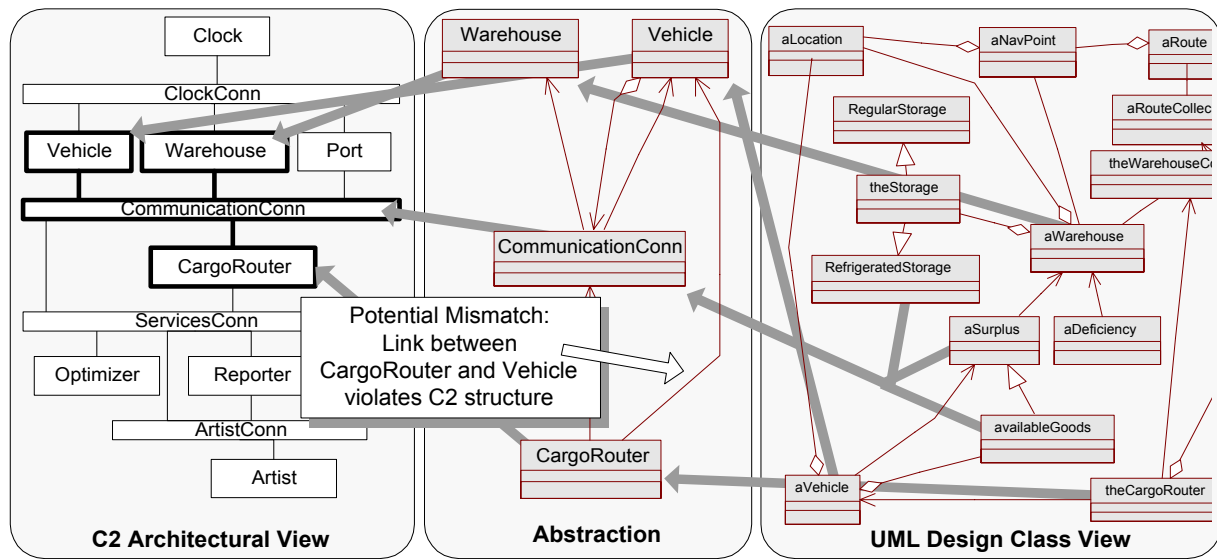


Figure 9. Use of an intermediate model to find a structural inconsistency between architecture and design models.

- Chung, L., Gross, D., and Yu, E. Architectural Design to Meet Stakeholder Requirements, In P. Donohoe, ed., *Software Architecture*, Kluwer Academic Publishers, 1999.
- Dardenne, A., Fickas, S., and van Lamsweerde, A. Goal-Directed Concept Acquisition in Requirement Elicitation. *6th International Workshop on Software Specification and Design (IWSSD 6)*, Oct. 1993.
- Egyed, A., and Kruchten, P. Rose/Architect: A Tool to Visualize Architecture. In *Proceedings of the Hawaii International Conference on System Sciences*, Jan. 1999.
- Egyed, A. Heterogeneous View Integration and Its Automation. Ph.D. Dissertation, University of Southern California, Aug. 2000.
- Egyed, A., and Medvidovic, N. A Formal Approach to Heterogeneous Software Modeling. *Conference on the Fundamental Aspects of Software Engineering*, Mar. 2000.
- Garlan, D. Style-Based Refinement for Software Architecture. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 72-75, San Francisco, CA, Oct. 1996.
- Gieszl, L. R. Traceability for Integration. *2nd International Conference on Systems Integration*, pp. 220-228, 1992.
- Gotel, O. C. Z., and Finkelstein, C. W. An Analysis of the Requirements Traceability Problem. *First Int'l Conference on Requirements Engineering*, pp. 94-101, 1994.
- Gorlick, M. M., and Razouk, R. R. Using Weaves for Software Construction and Analysis. *ICSE 13*, Austin, TX, May 1991.
- GroupSystems.com. <http://www.groupsystems.com/>
- Grünbacher P., Briggs B. Surfacing Tacit Knowledge in Requirements Negotiation: Experiences using EasyWinWin. *Proceedings Hawaii International Conference on System Sciences*, IEEE Computer Society, 2001.
- Kazman R., Barbacci M., Klein M., Carriere, S.J., Woods S.G., Experience with Performing Architecture Tradeoff Analysis, *ICSE '99*, Los Angeles, CA, May 1999.
- Khriss, I., Elkoutbi, M., and Keller, R. Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams. *UML '98*, Jun. 1998.
- Koskimies, K., Systa, T., Tuomi J., and Mannisto, T. Automated Support for Modelling OO Software. *IEEE Software*, Jan. 1998.
- Liu, J., Traynor, O., and Krieg-Bruckner, B. Knowledge-Based Transformational Programming. *Fourth International Conference on Software Engineering and Knowledge Engineering*, 1992.
- Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a Taxonomy of Software Connectors. *ICSE 2000*, Jun. 2000.

- Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE '99*, May 1999.
- Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., and Robbins, J. E. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 2001. To appear.
- Medvidovic, N., and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, Jan. 2000.
- Moriconi, M., Qian, X., and Riemenschneider, R. A. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356-372, Apr. 1995.
- Mullery, G. CORE: A Method for Controlled Requirements Specification. *ICSE 4*, Munich, Germany, Sep. 1979.
- Nuseibeh, B., Kramer, J., and Finkelstein, A. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, Oct. 1994.
- Partsch, H., and Steinbruggen, R. Program Transformation Systems. *ACM Computing Surveys*, vol. 15, no. 3, Sep. 1983.
- Perry, D. E. Generic Descriptions for Product Line Architectures. *2nd International Workshop on Development and Evolution of Software Architectures for Product Families (ARES II)*, Spain, Feb. 1998.
- Perry, D. E., and Wolf, A. L. "Foundations for the Study of Software Architectures." *Software Engineering Notes*, Oct. 1992.
- Robertson, S., and Robertson, J. *Mastering the Requirements Process*. Addison-Welsey, 1999.
- Shaw, M. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Proceedings of the Workshop on Studies of Software Design*, 1993.
- Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.