Technical Report
USC/CSE-99-TR-514

March 10, 1999

University of Southern California

Center for Software Engineering

Qualifying Report

for partial fulfillment of Computer Science Department requirements

Integrating Architectural Views in UML

Alexander Egyed

Center for Software Engineering
Henry Salvatori Computer Science Building 328
University of Southern California
Los Angeles, CA 90089-0781, USA
aegyed@sunset.usc.edu

Acknowledgements: This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the affiliates of the USC Center for Software Engineering: Aerospace Corp., Air Force Cost Analysis Agency, Allied Signal, Bellcore, Boeing, Electronic Data Systems, E-Systems, GDE Systems, Hughes Aircraft, Interactive Development Environments, Institute for Defense Analysis, Jet Propulsion Laboratory, Litton Data Systems, Lockheed Martin, Loral Federal Systems, MCC, Motorola, Network Programs, Northrop Grumman, Rational Software, Science Applications International, Software Engineering Institute, Software Productivity Consortium, Sun Microsystems, TI, TRW, USAF, Rome Laboratory, US Army Research Laboratory, and Xerox.

Abstract

To support the development of software products we frequently make use of general-purpose software development models (and tools) such as the Unified Modeling Language (UML). However, software development in general and software architecting in particular (which is the main focus of our work) require more than what those general-purpose models can provide. Architecting is about:

1) modeling the real problem adequately
2) solving the model problem and
3) interpreting the model solution in the real world

In doing so, a major emphasis is placed on mismatch identification and reconciliation within and among architectural views (such as diagrams). We often find that this latter aspect, the analysis and interpretation of (architectural) descriptions, is under-emphasized in most general-purpose languages. We architect not only because we want to *build* (compose) but also because we want to *understand*. Thus, architecting has a lot to do with analyzing and verifying the conceptual integrity, consistency, and completeness of the product model.

The emergence of the Unified Modeling Language (UML), which has become a de-facto standard for OO software development, is no exception to that. This work describes causes of architectural mismatches in UML views and shows how integration techniques can be applied to identify and resolve them in a more automated fashion. In order to do so, this work introduces a view integration framework and describes its major activities – *Mapping*, *Transformation*, and *Differentiation*. To deal with the integration complexity and scalability of our approach, the concept of VIR (view independent representation) is introduced and described.

Table of Contents

1 OBJECT-ORIENTED SOFTWARE DEVELOPMENT 1

1.1 OO AND THE SOFTWARE CRISIS 1

1.2 MODELS AS A MEAN OF ABSTRACTION 2

1.3 MODELS AND VIEWS 3

1.4 THE UNIFIED MODELING LANGUAGE (UML) 5

1.5 FROM INCEPTION TO TRANSITION 8

1.6 SOFTWARE ARCHITECTING 10

2 VIEW INTEGRATION 14

2.1 MISSING INTEGRATION IN MODELS AND VIEWS 14

2.2 WHAT IS INTEGRATION? 15

2.3 THE VIEW INTEGRATION PROBLEM 17

2.3.1 WHY INTEGRATE VIEWS? 17

2.3.2 WHY INTEGRATE ARCHITECTURAL VIEWS? 19

2.3.3 WHY INTEGRATE ARCHITECTURAL VIEWS IN UML? 19

2.4 MOTIVATION 20

3 SCOPE AND LIMITATIONS 23

4 RELATED WORK 25

5 VIEW MISMATCHES 32

5.1 VIEWS 32

5.1.1 DIMENSIONS OF VIEW INTEGRATION 32

5.1.2 TYPES OF VIEWS 35

5.2 EXAMPLES OF MISMATCHES 36

5.2.1 MISMATCH BETWEEN CLASS LAYERS 36

5.2.2 MISMATCH CLASS AND SEQUENCE DIAGRAMS 38

5.2.3 CARDINALITY MISMATCH 38

5.2.4 MISMATCH BETWEEN STATE AND COLLABORATION DIAGRAMS 39

6 VIEW INTEGRATION FRAMEWORK 42

6.1 MODEL-BASED DEVELOPMENT 42

6.2 INTEGRATION ACTIVITIES 45

6.3 MAPPING 47

6.4 TRANSFORMATION 50

6.5 DIFFERENTIATION 52

6.6 IDENTIFIED VIEW MISMATCHES 53

6.7 ASSUMPTIONS 57

7 AUTOMATING THE MISMATCH IDENTIFICATION 58

7.1 PROVIDING TECHNIQUES TO IDENTIFY MISMATCHES 58

7.2 USING ROSE/ARCHITECT TO ADDRESS THE LAYERING MISMATCH 59

7.3 USING SCED TO ADDRESS STATE/SEQUENCE DIAGRAM MISMATCH 61

7.4 TRACE OBSERVATION 63

7.5 APPLYING INTEGRATION TECHNIQUES TO ACTIVITIES 67

7.6 SHORTFALLS OF THE VI TECHNIQUES 67

7.7 SYSTEM MODEL (BASE MODEL; REPOSITORY) 68

7.7.1 MISSING INFORMATION IN UML MODELS 69

7.7.2 AVOIDING INFORMATION DUPLICATION 70

7.7.3 EXAMPLE OF MISMATCH IDENTIFICATION IN THE BASE MODEL 73

7.7.4 BASE MODEL AS A VIEW INDEPENDENT REPRESENTATION 73

7.7.5 REAL VIEWS AND DERIVED VIEWS 75

7.7.6 OTHER ISSUES 75

7.8 EXTENDING THE NOTATION AND SEMANTICS 76

7.8.1 DESCRIBING RULES AND CONSTRAINTS 76

8 MISMATCH RESOLUTION 78

10 INTEGRATING THIS WORK 80

10.1 MBASE 80

10.2 ARCHITECTURAL MISMATCHES DURING SYSTEM COMPOSITION 81

10.3 ADL VIEWS AS MINI-SPIRALS 83

11 FUTURE WORK 85

12 CONCLUSION 86

13 REFERENCES 88

14 APPENDIX 92

14.1 COMPLETE SET OF RULES IN ROSE/ARCHITECT 92

1 Object-Oriented Software Development

The art of being wise is the art of knowing what to overlook – William James

Object-oriented (OO) software development has come a long way since the introduction of the first OO programming language *Simula* in 1967. Simula was first to present the idea that data and functions should be kept together. Not much later Smalltalk, the first pure object-oriented language, emerged. The potential of the object-oriented programming paradigm was, however, not fully realized until the late 1980s. Since then we have seen the object-orientation of many traditional programming languages like C, Pascal, and even ADA. Today, the object-oriented programming paradigm dominates the market and it is "in" to talk and to think object-oriented.

1.1 OO and the Software Crisis

The reasons for the OO hype are rooted deep in software engineering and its problems. [Siegfried 1996] writes that the "object-oriented paradigm gives us tools for fighting the software crisis: Object-oriented techniques make it possible for us to handle large systems, change them, reuse parts of old systems in new systems, ease the communication between customer and developer, and much more."

Even though OO development has indeed brought many advantages, the hopes of having found the solution to the software crisis have not become true. Even smaller aspects, such as increasing reuseability (which was and probably still is one of the strongest drivers of the OO movement), have not paid off as it was anticipated because "when we are fighting the software crisis we are fighting human nature and human inability to handle complexity." [Siegfried 1996]

However, this does not mean that object-oriented software development has failed in its mission. Quite to the contrary, there have been great successes. For instance, combining data and functions into objects led to cleaner interfaces, higher internal cohesion, and less coupling between the objects (all desirable attributes). Furthermore, the object paradigm reflects the real world more closely. In doing so, software objects do not only reflect the properties of real objects but also their behavior. Nevertheless, using an object-oriented programming language does not solve all our problems.

Object technology is not the famous *silver bullet* Fred Brooks refers to when he talks about the *software werewolf* [Brooks, 1995]. Brooks lists as the major reasons for the existence of the software crises its four inherent properties: complexity, conformity, changeability, and invisibility. None of these properties are eliminated though the use of the object-oriented paradigm, albeit some are eased by it. [Carmichael 1994]'s statement, that with little doubt "object technology is the latest in a long line of pretenders to the role of 'Silver Bullet,'" certainly gives us some hope that this technology may yet be indeed useful in fighting the software crisis.

1.2 Models as a Mean of Abstraction

What makes software so complex and so difficult to grasp is the fact that the number of information loaded onto a single person is vastly exceeding the capabilities of the human mind. We are not able to handle thousands of pieces of information at any given time. Instead it seems that the human short-term memory is quite limited in that respect. The 7±2 rule is a well-known example. This rule states that the human mind can usually only handle 7 distinct things (plus or minus 2) at the same time.

Given that restriction, one may ask how we could have possibly evolved humanity in general (and engineering in particular) as far as we did? Obviously, even the simplest piece of technology consists of more (much more) than 9 pieces! The answer to that is probably the concept of ***modeling***. If we encounter too much information, we consciously and/or unconsciously group this information in some way which makes it easier for us to recall it later on.

Thus, the field of *Software Engineering* uses this human capability of abstraction to create "theories, methods, and tools which are needed to develop […] software." [Sommerville 1996] Those theories, methods, and tools in turn use further abstractions until we have pieces which are small enough for us to comprehend without any further abstraction. [Siegfried 1996] describes this need for abstraction clearly when he says that "it is not the number of details, as such, that contributes to complexity, but the number of details of which we have to be aware at the same time."

Being able to abstract information does, however, not mean that we are able to solve a complex problem. To do that, we use modeling techniques not only to abstract but also to solve problems in the abstraction. Figure 1 shows this process in the field of mathematical systems theory. There a problem solver uses some mathematical formula (function f(x)) to translate a problem from the real world to the model. Then, the problem in model form, if it is simple enough, is solved to yield a model solution. Applying the translation backward will give us a real solution out of the model solution. If the model problem would have been to difficult to solve, we could have applied the same technique recursively again (the previous model problem now being the real problem). As long as we end up with a new model problem which is easier to solve than the real problem we will eventually find a model problem which is simple enough for us to solve. Figure 1 shows that in Mathematics finding a solution to the real problem is reduced to finding a solution for the model problem.

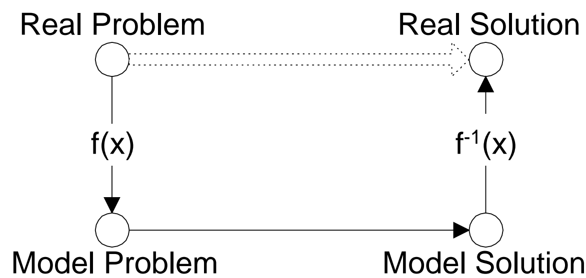

Figure 1: Mathematical Systems Theory

1.3 Models and Views

In software development we make use of models in a similar fashion. The introduction of the object-oriented development paradigm did neither stop their use nor their extensions. Quite the contrary, new models were created which could be used to represent new aspects which were unique to OO. Like in other (software) engineering domains, the models can be very distinct in their characteristics. Many models are (at least partially) graphical in nature, utilizing another advantage of the brain, that of being very efficient in handling pictures (a picture is worth more than a thousand words). Yet, other models are more textual, spanning the use of plain English to some type of formal or semi-formal language.

Most of those modeling techniques were found to be of great value for some aspect of software development, and, thus, based on them, a large number of development techniques emerged. It was only natural that people started to combine those techniques into development methodologies, which worked well together, and which seemed to cover the most important and interesting aspects of the development process. Over time, the community was able to standardize some of the development models, providing more general models, which in turn were applicable to a larger domain of software-intensive systems. The Unified Modeling Language (UML) [Booch-Jacobson-Rumbaugh 1998] is the result of the endeavor to unify object-oriented analysis and design techniques and their associated diagrams into a common model. The principles, which guided the mathematical model in Figure 1, are, however, still visible in software engineering models (see Figure 2).

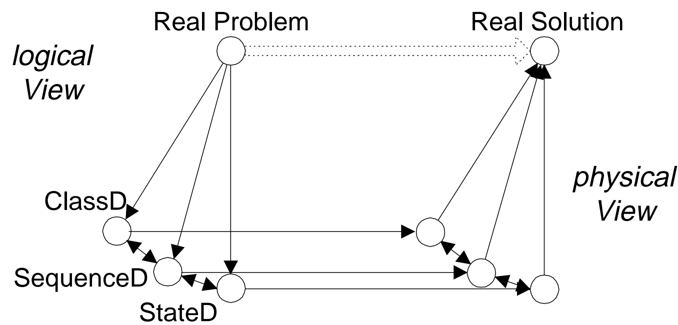

Figure 2: Software Engineering Theory

Figure 2 shows the task of going from a real software problem (via some high-level, logical model of the problem) to a software solution (which in turn is described in some formal way, e.g. programming language). As mentioned before, that task of solving the real problem directly is often too difficult to do without any models and thus, Figure 2 shows the usage of some diagrams (models), such as the Class Diagram, Sequence Diagram and the State Diagram to bridge the gap. The real picture is of course more complex since it usually involves multiple levels of abstractions. Nevertheless, the basic idea is still the same.

With that we are coming to the term *View*. So far we have used the word model in many ways. We said that a diagram is a model, a mathematical representation is a model, we had a problem and a solution model, and so forth. In this work we will not use the term *Model* this general but instead we will refer to a

model as the union of all its representations – or as a union of all it's *views.* The IEEE Draft Standard 1471 refers to a view as something which "addresses one or more concerns of a system stakeholder." By stakeholder we mean an individual or a group which shares concerns or interests in the system (e.g. developers, users, customers, etc.). Applied to our context, a *View* is a piece of the *Model* which is still small enough for us to comprehend and which also contains relevant information about a particular concern. As such, the diagrams depicted in Figure 2 really show views of the problem/solution model.

Given this, it would be ideal for us to only use views which can be used for all/most development concerns. The only drawback is that we do not have many of those views we could apply throughout the development process. One reason is the audience for which those views are intended. If the audience is a customer or user then the emphasis is to have something which is nice and simple to understand, and which models the 'what' of a system. This does not mean that the developers and other stakeholders involved in the development process would like to use views which are hard to understand, however, those people need to be able to specify in a higher level of precision using less ambiguity, which usually makes those descriptions harder to understand in order to describe the 'how.'

In an extreme case, we could say that the user and customer would like to use a natural language (e.g. English) to define a problem but the developer would have to use a structured and formal language to create a solution (e.g. programming language). The one being highly understandable but very ambiguous, the other being much less understandable but very precise. A partial solution to this problem is the use of some sort of semi-formal language which is still easy to understand but not very ambiguous. This option is the most useful approach currently available, but it has shortfalls both in general understandability and precision. Finding a language which is both easy enough for *any*body to understand and yet highly precise has not been achieved. This brings us back to our need of having multiple views for different stakeholders.

1.4 The Unified Modeling Language (UML)

In the remainder of this work we will primarily use the Unified Modeling Language (UML) as the foundation of our development model. In this section we will briefly describe UML, which is currently the leading object-oriented analysis and design model. UML's views are for the most part graphical diagrams.

However, it must be understood that UML is only **a part of** our development model (as the next chapter will explain) since it is missing important concepts.

UML [Booch-Jacobson-Rumbaugh 1998] is the result of a collaboration of numerous companies and OO modeling experts, led by Rational Software Corporation, and it borrows heavily from Booch [Booch 1994], OMT [Rumbaugh et al. 1991], and other OO models [Coad-Yourdon 1991a][Coad-Yourdon 1991b][Jacobson 1992]. The lead designers of UML were Booch, Jacobson and Rumbaugh.

"UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems" [Booch-Rumbaugh-Jacobson 1997]. These different but overlapping uses of the model (or language) can only be achieved by supporting a variety of views. Some of these views (representations) can be seen in Figure 3. This figure shows four of the most popular graphical design techniques which are currently supported by UML; the collaboration diagram, class diagram, use-case diagram, and state diagram.

These views and others are briefly explained in the following. A more detailed description of some of these views is given later. Please refer to [Booch-Rumbaugh-Jacobson 1997] for a detailed description of UML (Version 1.2).

- **Use Case** (e.g. Use Case Diagrams): Depicts the interaction between the user and the system or between one system and another. In doing so, use cases provide a high-level view of the usage of a system which frequently shows the interaction of multiple functions of the system. E.g. the task of editing a document involves the functions *open document*, *edit document*, and *save document*.
- **Interaction** (e.g. Sequence and Collaboration Diagrams): Sometimes also referred to as Mini-Uses. Interaction diagrams show concrete examples (e.g. test case) of how components communicate. They can often be seen as an actual test cases which involve the use of a single function (see use case). A function can refer to the GUI (e.g. open file) or to subcomponents of the system.
- **Objects and Classes** (e.g. Class Diagrams, CRC Cards): Classes are the most central view in UML (and all other OO models). They depict the relationships between classes and objects

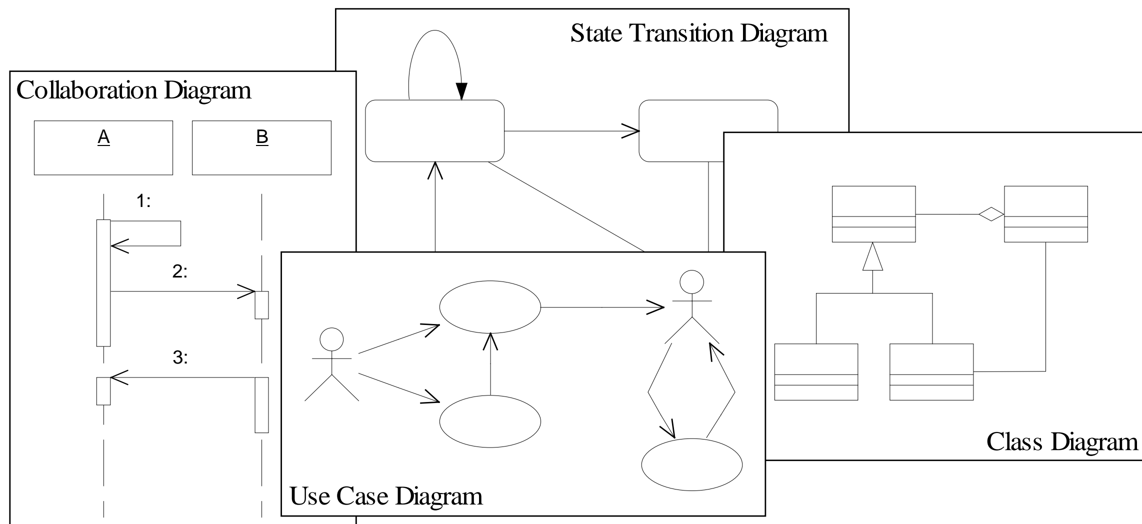

Figure 3: Some of Diagrammatic Views support by UML

which are the smallest stand-alone components in OO. Relationships can depict instances, part-of relationships, dependencies, and others.

- **Packages** (e.g. Package Diagram): Packages are used to group classes into layers and partitions. As such they show the functional decomposition of a system.
- **State Transition** (e.g. State and Activity Diagrams): This well-known technique is used in UML to describe the states a class can go through. In UML, state diagrams are restricted to a single classes only. Activity Diagrams are a generalization of state diagrams in that they can also be used to depict events or other 'transitional' elements.
- **Deployment** (e.g. Deployment Diagrams): Shows the relationship of the components of the system during deployment. As such it presents a physical view of the system. It is, therefore, frequently used to depict the component dependency of the actual implementation.

UML does, however, not cover the entire sot of useful stakeholder views. In addition to the views described above, the following are often considered to be very important extensions:

- **Information Flow** (e.g. Data Flow Diagrams): Shows the functional flow of information. This view is particularly useful to users and customers since the human mind tends to think in terms of flow of information (documents, etc.). However, the decomposition yielded through this type of view is not well matched to object-oriented design.
- **Interface/Dialog** (e.g. Interface Flow Diagrams): Describes the usage of the user interface of the system which again does not reflect any OO structure. Instead, it reflects the use cases in describing which functionality needs to be available where in the user interface (GUI). The Interface/Dialog Diagram reflects what the user sees of the system.

An object constraint language (OCL) [Booch-Rumbaugh-Jacobson 1997] supports the UML model and provides some limited integration within and between those views. OCL is a formal language for expressing constraints on model elements in UML. Since users can extend UML (e.g. through stereotypes), OCL can also help in integrating new techniques into UML. For instance, some work was done in integrating Architecture Description Languages, like C2 and Wright into UML [Medidovic 1998]. We will make use of OCL in further integrating architectural views in UML.

1.5 From Inception to Transition

Naturally, a software development model should contain enough information so that it may be useful throughout the entire development life-cycle. Thus, the development model should have views giving information about the requirements, the organization, the people, the technology, the history of changes, the structure of the product, its transition, and so forth.

Seeing the model solely as a collection of views is however not right either. A development model should be more than that. For instance, it should give information on how to use the model and its views. As such, it needs to answer questions like 'what has to be done first' and 'who needs to do what.' Traditionally, models which give guidelines are commonly referred to as process models or life-cycle models and the probably best-known but also most controversial process model is the *Waterfall Model* [Royce 1970]. It defines the development process as a rather sequential process (feedback loops are allowed) with an upfront planning stage, followed by high level design and implementation. Even though it

is widely acknowledged that a purely sequential process is generally infeasible to follow, the stages it defines are still visible in many newer process models.

In the literature we often read that the Waterfall model is dead and clearly no other model has as of this date gained a similar popularity. Especially the object-oriented community seems to hold the belief that OO development is different and that regular process models do not seem to apply here. Even though the latter may be true, this does still not mean that process models are obsolete. The wisdom which was gathered and incorporated into the Waterfall model (or any other process model for that matter) was useful not because we used those process models but because we needed assistance with the intrinsic properties of the software "werewolf"– complexity, conformity, changeability, and invisibility. As it was mentioned above, the object-oriented paradigm may ease some of those problems but the fact remains that we are still dealing with the same problems we have dealt with 30 years ago – even with OO.

We therefore can only conclude that the myth that we do not need a life-cycle (process) model because we are using OO design and/or programming languages [Siegfried, 1996] is exactly that – a myth. Siegfried states this very nicely when he says that "using object-oriented methods does ***not*** mean that we can ignore what we have learned since the 1950s." Most of the lessons, best practices, guidelines, etc. which we have accumulated in the last 30 years are still as valuable and useful as they were before. It is just the way we have to use them which changed with the OO paradigm. Supporting a useful process model for OO development is therefore an essential step in integrating our OO views and there are a number of very good approaches. For instance, there is the WinWin Spiral Model [Boehm 1996] and the Rational Unified Process [Kruchten 1999]. Figure 4 shows an overview of the Rational Unified Process as it has integrated with the LCO, LCA, and IOC Anchor Points of the WinWin Spiral Model (defined later). It shows the major UML stages of an OO development process called *Inception*, *Elaboration*, *Construction*, and *Transition*. Each stage may use one or more iterations (e.g. spiral cycle) to complete it and each milestone must deal to some degree with the process components listed on the left. For instance, initially, we focus most on requirements capture and analysis whereas later we concentrate more strongly on implementation and test. However, all of the activities proceed concurrently, as it can be seen in the figure. Even during construction new requirements may be identified, thus, the iterative nature of the life-cycle model. It is out

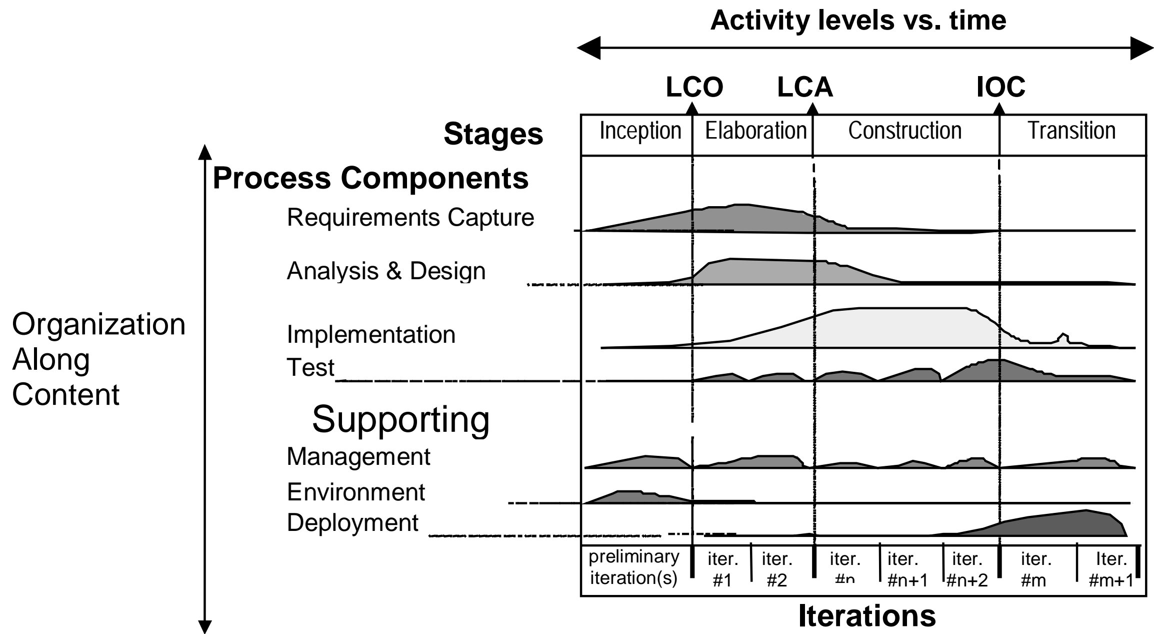

Figure 4: Rational Unified Process with the WinWin Spiral Model's Anchor Points

of the scope of this work to go deeper into this subject. For more an overview on process models also refer to [Chroust 1992].

1.6 Software Architecting

It is out of the scope of this work to incorporate all forms of development views. As mentioned above we will restrict ourselves mostly to UML views (which are explained in more detail later) and even there we will emphasize the high-level design and architectural views only. This section will therefore describe what we mean by the *Architecture* of a system.

To define what software architecture is, is already a problem in itself. Not many people can agree on a single definition. Thus, let me start by describing the common usage of the term which is based on the analogy to building architectures (an early definition is given by Wolf-Perry [1992]. The analogy between buildings and software may not be striking on first glance but the following may make it clearer.

Both buildings and software have an implementation representation; the actual building in bricks, stones, etc. and the implementation of the software product in a programming language. Both building and software architects use logical descriptions (blueprints) to describe the implementation. The building architect does so by describing the building's essential features without committing too much to specific

construction details such as what kind of building materials to use. Similarly we would like to architect software without overcommitting to implementation details (e.g. what programming language to use). The building as such may stand for many years but may undergo some changes – superficial ones or more profound structural ones. In general, software has to undergo more changes throughout its life time – some more to the eye (GUI) others more profound. Both buildings and software have in common that some design decisions (which must be done early on) may have considerable impact on other decisions later on. For instance, a wall that needs to support a ceiling cannot easily be moved or removed (if at all). Similarly, some design decisions in software (e.g. commitments to COTS products) may impose similar constraints.

Furthermore, building architects are able to reuse some of their architectural information. For instance, if a settlement with a number of similar (not necessarily identical) houses is built, the architect may still be able to use most parts of the blueprint of a house for the others. In times where software development concepts such as product lines are gaining in popularity, we clearly would also like to have something equivalent to that for software architecting. And finally, a building architect is able to describe a building in sufficient detail so that construction workers can build it without too much interaction with the architect. Thus, the architectural description of the building is complete enough to minimize additional mouth-to-mouth interaction between designers and builders – attributes we also would like to see in software architecture descriptions.

Clearly, software architecture and building architecture have many things in common, at least in principle. In practice, software architecture descriptions do not come anywhere close to the preciseness and general understandability of building architecture descriptions. However, in the long run we hope that we may yet come closer to that goal.

Recently, there has been some attempt in standardizing what architecting means. The IEEE Draft Standard 1471 is one of the latest. Let us provide their definition for architecture which illustrates some of the challenges in defining the term.

> *Every system has an architecture, defined as follows:*
> *An architecture is the highest-level conception of a system in its environment where: the 'highest-level' abstracts away from details of design, implementation and operation of the system to focus on the system's 'unifying or coherent form'; 'conception' emphasizes its nature as a human abstraction, not to be confused with its concrete representation in a document, product or other artifact; and 'in its environment' acknowledges that since*

systems inhabit their environment, a system's architecture reflects that environment. [IEEE 1998]

This definition's primary problem is vagueness: it could apply equally well to "architecture," "requirements," or "operational concept." Another problem we see with this definition is that it seems to emphasize too little onto the analysis and interpretation of architectural descriptions. We *architect* not only because we want to build but also because we want to understand. Thus, architecting has a lot to do with analyzing and verifying the conceptual integrity, consistency, and completeness of a design. This does not mean that above description excludes the analysis of architectures. Quite contrary. Everybody acknowledges its existence as part of architecting. However, it seems that builders of architectural descriptions only think about their analysis as a second thought and then mostly in vague terms. UML is certainly a very good example of that. This deficiency will be investigated shortly.

Architecting in UML may be seen in Figure 5. This figure shows UML views and other views which often are part of the major development stages (from the developers point of view) – the analysis, architecture, and design of a software system. The arrows depict the dependencies of the views onto information in other views. This figure should not be taken too literally since we tried to capture the major flows of dependencies only. For instance, the picture shows that the classes and objects affect the implementation (e.g. code in C++) but not vice versa. This is, of course, not always true. There are cases

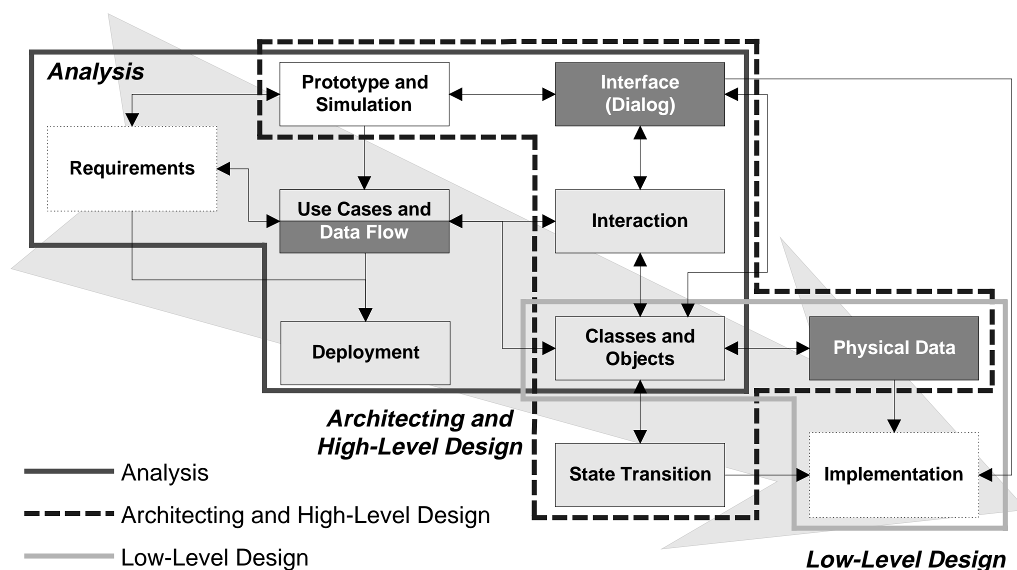

Figure 5: Architectural Views in UML

where the implementation may trigger changes in the architecture (e.g. due to choice of COTS product). As a general rule, it is good practice to anticipate these dependencies and address them via prototyping and analysis as emphasized in Figure 4.

Further, the associations of the development artifacts (such as classes, use cases, etc.) to the major phases of the life cycle can indicate primary associations at best. Again, we tried to capture the major associations of those development artifacts and the views in which they are frequently used. It is this ambiguity in how to associate and relate those artifacts which already poses our first problem in OO development. Traditional life-cycle models such as the Waterfall Model are less useful in OO development because of the activity and artifact overlaps we discussed in Figure 4. This fact can also be seen in Figure 5 where some development artifacts, such as classes and objects, are used and shared extensively during the entire development process. This ambiguity, in the definition of development stages and phases, is however also a good thing since it provides some continuity between the life-cycle stages and, thus, brings the development stages closer together. The conceptual breaks, which so frequently happen between the analysis and design stages, are eased.

2 View Integration

The beginning is the most important part of the work – Plato

2.1 Missing Integration in Models and Views

In the previous chapter we defined what it means to do OO software development. We talked about models and views, and used UML as an example. We confirmed the continuous need for a life-cycle model to guide us through the development process and we talked about architecting as being one of the most critical development stages from the software developer's point of view.

With that we basically described everything a well-equipped development team has access to these days. However, we still have a major problem. When we described the mathematical problem solving approach (see Figure 1) we concluded that modeling (architecting) replaces the finding of a solution to the real problem by the finding a solution to the model problem. For that very reason, software development models were devised which serve as counterparts to the mathematical model. However, are our software engineering models (like the one we showed in Figure 5) really equivalent to the mathematical model in solving the problem (see Figure 6 for a comparison)?

What if the (software) model which we created to represent the real world is not adequate? A solution we might find to that model problem would therefore not be correct. This implies that we are not only confronted with the challenge of finding a (model) solution to a model problem but also to find a

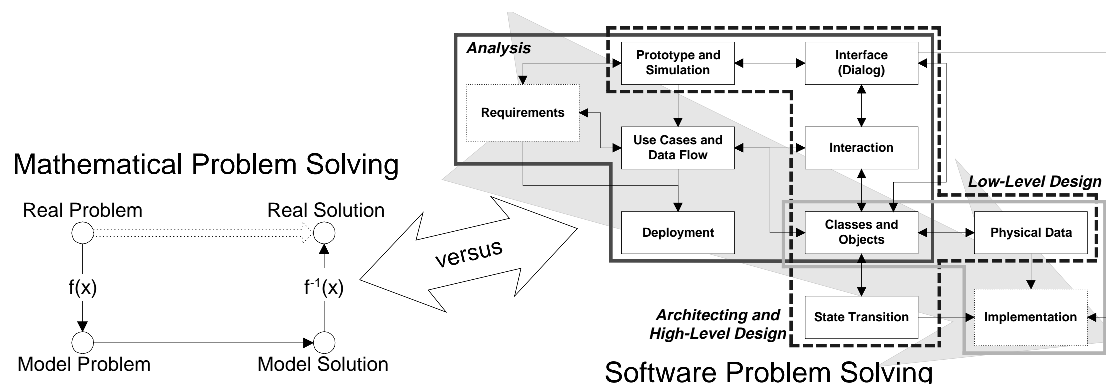

Figure 6: Two Problem Solving Approaches

model of the real world which adequately represents it for our needs. This is like solving the right problem vs. solving the problem right! As such the Mathematical Problem Solving Approach is really doing three things (corresponding to the three arrows):

- **Model the real problem adequately**
- **Solve the model problem**
- **Interpret the model solution in the real world**

Which part of the software model in Figure 6 is doing the modeling? Which part is doing the solving? And which part is doing the interpretation of the solution? None? But, what does this tell us about conventional software development models such as the UML? For instance, what is the best implementation of a software product if it does not reflect the architecture? What is the best architecture if it does not satisfy the requirements?

The only conclusion we can draw from that case is that Architecting is more than what conventional development models provide. ***Architecting is to model, to solve, and to interpret.*** And techniques such as the ones used by UML are just providing assistance. Therefore, this work is about integrating views so that they provide more than just structural assistance. In particular, we will investigate the integration of some architectural views in UML and what techniques we can deploy to bridge the gap between what architectural models are and what they should be.

2.2 What is Integration?

We have used the word *Integration* or 'what it means to *integrate'* but so far we have not described it. This section will do that. The term *Integration*, as such, is part of everybody's vocabulary. Therefore, let us check how the Merriam-Webster Dictionary defines that term. There, *Integration* is defined as:

> *1) The act or process or an instance of integrating: as a) incorporation as equals into society or an organization of individuals of different groups (as races) b) coordination of mental processes into a normal effective personality or with the individual's environment*
> *2) The operation of finding a function whose differential is known; the operation of solving a differential equation*

This set of definitions for the term *Integration* is very general. That should, however, not surprise us since this is how the word *Integration* is used. In Software Engineering it applies to Technology, Organization, and People; it affects management, products, humans, politics, standards, models, enterprises, and much more. Sage and Lynch's work about *Systems Integration and Architecting* [Sage-Lynch 1998] provide a very comprehensive overview of what integration in our context means. They found that "Systems Integration is an activity omnipresent in almost all of systems engineering and management." They further found that "the term lacks precise definition and is used in different ways and for different purposes in the engineering of systems."

In software engineering and software architecting, the word *Integration* is used frequently. It often refers to the process of assembling components (or subsystems) into a system. As such, the term integration stands for an activity that starts later on in the life-cycle once some components of the software system are developed. Another case, where the term *Integration* is used, refers to the unification of standards, processes, and models. For instance, the Integrated Capability Maturity Model (iCMM) of the FAA (which is a union of various CMM models (such as the SW-CMM [Paulk 1995], SE-CMM [Kuhn 1996], SA-CMM [Ferguson et al., 1996] and so forth) is one such attempt to combine standards to a more general one. The Unified Modeling Language (UML) is another such case, where various object-oriented development models (Booch, OMT, and pieces of many others) were combined into a integrated single OO development model.

In this work, the term *Integration* is used, in yet another way, to determine the semantic integrity of development models (or views, diagrams, etc.) in order to evaluate or improve quality aspects of the development model. Desirable qualities we would like to see in a development model (or its instances such as the product or domain models) are consistency, completeness, correctness, and so forth. So we may ask ourselves:

- What does it mean, for one view to be *consistent* with another view?
- When do I know, whether one view presents a *complete* picture of the entire system?
- How do I know that what I did is correct and faithfully represents what my customer wanted me to do?

On a close look, this form of *integration* is, however, not very different from the meaning described above. For instance, when we perform a component integration where we evaluate the integrity of components while assembling them to a bigger component (or even system) this is quite analogous to performing a view integration where we evaluate the integrity of views while assembling them to a bigger model. The one describes the product integration, the other the view integration. Both are facets of ***Integration*** (see also [Grady 1994] for an overview of these facets).

2.3 The View Integration Problem

2.3.1 Why Integrate Views?

Above we briefly described an object-oriented development model (UML) which satisfies the need of our stakeholders (such as users, designer, programmers, and maintainers) for views which can be used by them to describe and communicate. We further briefly described a process which can be used to guide them and advise them on how to use those views in creating a useful and feasible software product. And we also described the deficiency of that approach when it comes to solving the problem (to model, to solve, and to interpret).

This deficiency in views would not exist if we could have a few *perfect views* that could be used by all stakeholders (as described above) and which were precise enough but still easy enough to use. These views, unfortunately, do not exist. Instead, we are confronted with a number of loosely coupled, sometimes quite independent views. This is not really what we want. [Nuseibeh 1996] wrote that "multiple views often lead to inconsistencies between these views – particularly if these views represent, say, different stakeholder perspectives or alternative design solutions."

Thus, if we have to deal with multiple views we would like to have at least tightly coupled ones. Since a view represents only one aspect of the system to be modeled, they are meant to be together – only together can they fully describe the system. However, we also need those views to be different (and independent) enough to provide useful meaning to their respective stakeholders. Therefore, what we need

are views which are independent and can stand on their own, but with their contents being fully integrated with the contents of the other views to ensure their conceptual integrity. Thus, we need *View Integration*.

We also need integration, because the views often use different underlying paradigms and, thus, the results of modeling a system in one view may be different than modeling the same system in another view. For instance, a non object-oriented analysis and design stage would yield functional model elements as its major components (which are more suitable to be implemented in a functional programming language). Instead, using object-oriented design techniques (or views) would already structure the system in a more object-centered fashion and thus, its implementation will be more straightforward in an OO language.

In Figure 5 we showed both object oriented (classes, interaction) and functional views (data flow, state transistion) which are commonly used together in OO development. So if two different people would start creating a system, one using OO techniques and the other using functional ones, we would most likely get two different solution approaches for the same system. Even if each view were correctly solving the problem, they would still not make much sense together. This is because the one type of technique would yield a system which is structured by its functions whereas the other one would be structured by objects which have behavior (functions). Further, if modeling is done separately (one view at a time) we may get inconsistencies between them. The notation described above does not describe the semantics of the model and how it is (or is not) supposed to be used. Life-cycle process descriptions may help in that but they are usually not detailed enough and for the most part they are not supported by tool.

Thus, what we need is a development model which is not only defined syntactically but also semantically. Such a model would also need tool support which would not only enable the architects to create a model instance for a system which is syntactically correct but the tool should also be able to verify the semantic integrity of the model instance (at least to some degree).

The integration of architectural views (as the title says) is about adding semantics to our architectural views so that the integrity of the whole is improved.

2.3.2 Why Integrate Architectural Views?

The reason why we chose the integration of architectures was because it is the most important part of the design. This is best explained by [Siegfried 1996] who wrote that “there is no replacement for making a sound systems architecture early in a project.” Architecting is the start of a development process from a pure engineering point of view. Architecting is also early in the development life-cycle which means that problems and faults are still relatively easy (and inexpensive) to fix. Should architectural errors be carried into the implementation phase or even further, the cost of fixing them are some orders of magnitude higher [Boehm 1981]. Further, architectural descriptions are already low-level enough to be less ambiguous. Thus, there is more precise information available from which we can draw from. Integrating requirements (which is equally important) is much harder to do since we would require techniques such as natural language understanding which do not exist in a level of sophistication suitable for our needs.

2.3.3 Why Integrate Architectural Views in UML?

The beginning of this work already indicated why we chose object-oriented technology as another cornerstone of this work. OO is more and more dominating the market and UML has evolved into the most significant OO analysis and design methodology. The Object Management Group (OMG), furthermore, has standardized UML for the Object Management Architecture (OMA). “The adoption of UML provides system architects working on Object Analysis and Design with one consistent language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling.” [OMG 1997] We also chose UML because others have already made some progress in integrating UML views. Thus, summarizing previous sections we can say that views alone are not solving the consistent architecture representation problem because they:

- are standalone/independent
- involve different types of modeling elements
- are for different audiences/stakeholders
- are often used concurrently

This means that same or similar information is entered multiple times and that related information must be kept consistent manually. The *View Integration Problem* exists because it is often not apparent what information is duplicated or inconsistent. Therefore, finding means of ensuring the conceptual integrity are based on the ability of identifying duplicated model elements and integrating their properties.

2.4 Motivation

The absence of view integration is not a new discovery. Quite the contrary. As mentioned before, many model descriptions talk about the need of keeping the model(s) consistent. Sometimes, process models provide additional guidelines on what activities one can do to improve the conceptual integrity of architectures. For instance, a case study in using the WinWin Spiral Model [Boehm et al 1998] suggests using Architecture Review Boards [AT&T 1993] after the LCO (life-cycle objectives) and LCA (life-cycle architecture) stages to verify and validate the integrity of the analysis and design. A similar viewpoint can be seen in countless other research work:

- [Sage-Lynch 1998] describe various aspects of integration (enterprise wide). They frequently stress "the important role that architecture plays in system integration." They present the need for three major views: enterprise view, systems engineering and management view, and technology implementation view – and they stress to ensure consistency among these views.
- [Rechtin 1991] emphasizes strongly the validity and consistency of requirements as well as the interface definitions. He further suggests the need for problem detection and diagnosis.
- [Gacek et al 1995] present the results of a survey of people who are somehow involved in software development processes (developers, customers, maintainers, aquisitioners, etc.). There they found that, with respect to architects, the three major concerns were "1) requirements traceability; 2) support of tradeoff analyses; and 3) completeness, consistency of architecture."
- [IEEE 1998] speaks of *Architecture Evaluation*. "The purpose of evaluation is to determine the quality of an architectural description, and through it assess the quality of the related architecture." They further state the need of evaluation criteria against which the architecture can be verified.

- [Kuhn et al 1996][Humphrey 1995][Paulk et al 1995] who defined the Software and Systems CMM (Capability Maturity Model) stress the need for integration and quality control as part of the software life-cycle. Especially the SE-SMM (Systems CMM) identified *Integration*, *Validation*, and *Architectural Evolution* as key process areas.
- [Nuseibeh 1995] wrote that "inconsistency is an inevitable part of a complex, incremental software development process" and that "the incremental development of software systems involves the detection and handling of inconsistencies."
- [Wang-Cheng 1998] propose a more rigorous object-oriented design process to deal with the shortcomings of the OMT model. We share their view when they say that "the lack of a well-defined semantics for the individual [OMT] models and their integration hinders the overall development process."
- [Perry-Wolf 1992] realized the importance of software architectures early on and they state as one of the four major benefits of architectures that they are "the basis for dependency and consistency analysis."
- [Shaw-Garlan 1996] describe architecture very provocatively as being "a substantial folklore of system design, with little consistency or precision." They further state that "software architecture found its roots in diagrams and informal prose. Unfortunately, diagrams and descriptions are highly ambiguous."

These references, and many more, talk about the need for (or lack of) integration. Nevertheless, they usually do not describe the involved activities in detail (the works of [Nuseibeh 1995] and [Wang-Cheng 1998] are exceptions in that they provide some mechanism which we will discuss in Related Work). Sometimes this is done on purpose, such as in the CMM since they do not wish to favor a particular integration approach. However, in most cases it seems that the architects and designers have only few powerful tools when it comes to ensuring the integrity of their work.

On the other hand, those techniques which are sometime suggested are often aimed to make people talk to each other. For instance, the Architecture Review Board [AT&T] or the Inspection Process [NASA 1993] are primarily tailored at getting the most capable people together so that they may share their

information. These techniques may follow a defined process (e.g. checklists) and may yield very effective results but the actual activities of identifying and correcting defects are still done manually without much automated assistance.

This deficiency, the lack of automated assistance in identifying and resolving architectural mismatches, is the motivation for this work.

3 Scope and Limitations

If a man will begin with certainties, he shall end in doubts;
but if he be content to begin with doubts he shall end in certainties – Francis Bacon

The UML modeling technique has become a de facto industrial standard for object oriented software development. This was described previously as being an important reason why we chose it for this work. Nevertheless, UML is not quite complete. Its definition is ambiguous in many places and many interrelationships between views are not defined. Since it is not the scope of this work to formalize UML we will use other peoples interpretation and formalization whenever necessary and applicable (e.g. [Cheng et al. 1995] or [Övergaard 1998]).

Further, as mentioned before, we will restrict ourselves to some views. Thus, we will make no attempt to integrate the entire palette of UML views. Furthermore, we cannot guarantee that our integration approach will be able to cope with all forms of inputs. It is not our goal to be as complete/consistent as possible; but as complete/consistent as feasible. [Nuseibeh 1996] shares this view when he talks about viewpoint integration.

To address the challenge of integrating views, we will introduce a number of integration techniques. Similarly, these techniques may not work under all circumstances; neither will they cover all required aspects. This work does not emphasize individual integration techniques even though they are worthy research topics in themselves but instead it tries to emphasize how those techniques can be used together to identify potential mismatches and on how to resolve them.

This brings us to the next limitation. The rules and constraints we present in this work for identifying mismatches are not guaranteed to yield correct results under all circumstances (this restriction is a natural extension of above restrictions). Thus, mismatches that are identified with our techniques must be regarded merely as ***potential*** mismatches but not as factual ones.

Another restriction of this work is the issue of mismatch resolution. We will primarily investigate the issue of mismatch identification. We will only secondarily investigate how to automatically resolve mismatches with minimal or no feedback from the architect.

Having defined all the things this thesis will not do, we will now summarize what the scope of this thesis is. This thesis involves primarily four tasks:

1. Come up with a view integration framework as a foundation to support the definition and (automatic) identification of (potential) mismatches.
2. Finding techniques for the view integration activities defined in above framework. For this we will partially rely on existing technology but we will also introduce some new concepts.
3. Combining those techniques to make them work together (this step is primarily necessary because we use some techniques from other researchers; those techniques need some adaptation by us to be useful for our purposes).
4. Identify (potential) mismatches based on rules (constraints). Some mismatch resolution options will be investigated as well.

4 Related Work

All intelligent thoughts have already been thought; what is necessary is only to think them again – Johann Wolfgang von Goethe

In one form or another, the view integration problem has been worked on by numerous researchers. This section discusses their works and also discusses in what ways their works differ from ours. It is important to note that their works are not independent or in any way orthogonal to ours. [Sage-Lynch 1998] wrote that "unfortunately, there appear to be no detailed definitions that distinguish between various types of integration, and this may appear to make the subject disjoint. [… However] integration is generally always being performed, but it is not clear as to where it is performed or how to accomplish it successfully."

This means that integration is part of every aspect of the development life-cycle and, thus, our work and the related work presented in this section, fit somehow into the greater scheme of the view integration problem. *Integrating this Work* (a chapter later on) will briefly investigate this deeper aspect. Because of the depth of the integration problem it is far out of the scope of this work to present a complete survey, however, the following list should give the reader a reasonable detailed overview:

1. **Sage and Lynch [Sage-Lynch 1998]:** We start this overview with their work because they seemed to be one of the few sources who have attempted to summarize all key aspects of integration even across the boundary of software. Their work on *Systems Integration and Architecting* covers integration aspects, principles, and practices on the system level going far down into details of systems and software development. Their recent summary is an excellent work of 50 pages and we could not possibly provide a better one here. They talk about the need for integration on the systems engineering level and present the results and findings of numerous researchers. Systems engineering differs from software engineering in that it tries to cover all that is offered by the latter but more. It also covers hardware aspects and how software is integrated with it. With that in mind they address the need of (software)

architectural integration, however, the scope of their work did not permit more than an overview. If the reader of our work would like to know how our architectural integration approach fits into the 'bigger picture' we can only recommend their article.

2. **Abd-Allah and Gacek [Abd-Allah 1996][Gacek 1998]:** Their works address a very similar problem to ours – that of how to identify architectural mismatches. However, we take very different approaches in doing so. Their work investigates mismatches on a rather very high-level, that of software component composition. So for instance, if a software product is composed of a number of subcomponents – some of which may be COTS (Commercial-of-the-Shelf) components – then based on certain properties of these components there could be some potential mismatches. One of the key differences of their work to ours is that they treat components of the software product as black boxes. This is also their major weak point since it is unclear what property values to assign to components whose internal structure is unknown (e.g. COTS). Our work also differs in that we do not try to cover the integration of software components with COTS components but instead we are treating a software component as a white box and our work investigates primarily on how to avoid architectural mismatches within one such box. Nevertheless, their work can be used to extend ours in identifying mismatches if our architecture (box) should be integrated with other COTS products.

3. **Garlan, Monroe, and Wile [Garlan et al 1997]:** The last few years have brought several architectural description languages (ADL). Usually, each ADL stands for itself and in the context of this work can be seen as a view. However, the work of above researchers on an architectural interchange language called ACME (see Figure 7) has produced something else: an mechanism to share and exchange information between various ADLs (examples of ADLs are C2 [Taylor et al 1996], Darwin [Magee-Kramer 1996], Rapide [Rapide 1996], UniCon [Shaw et al 1995], Wright [Allen-Garlan 1996], just to name a few; [Shaw-Garlan 1996] provide a more comprehensive overview of what ADLs and styles are). Fundamentally, their

work is similar to what we are doing with respect to UML views, however, their work concentrates on ADLs. The set of ADLs supported by ACME does not necessarily exclude UML (there are some who argue that UML is another ADL), however, ACME itself does not provide mechanisms to check for inconsistencies across various views. Nevertheless, having a mechanism to share modeling elements is an integral step in identifying and resolving architectural mismatches. Thus, we think their work could be fundamental in fully integrating ADLs.

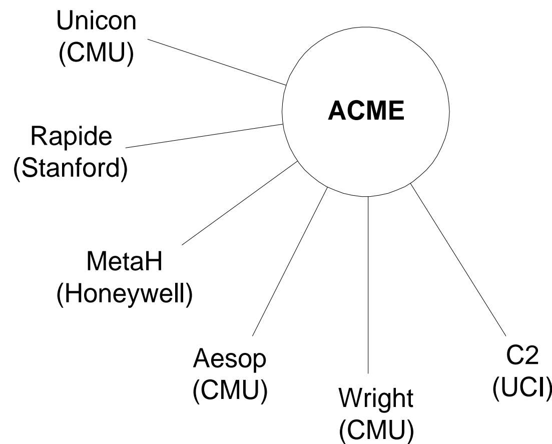

Figure 7: ACME and View Integration

4. **Robbins, Medvidovic, Redmiles, and Rosenblum [Robbins et al 1998]:** Their work presents a bridge between the ADLs presented in the previous bullet and UML. Here OCL (Object Constraint Language, which is part of UML [Booch 1997]) was used to represent two ADLs (C2 [Taylor et al 1996] and Wright [Allen-Garlan 1996]) in UML. The relationship is one sided in that UML is used to represent C2 and Wright but not vice versa. Even though this work gives a nice insight on how UML and other ADLs can be merged, it comes short of fully integrating those views. Nevertheless, their work may be seen as a foundation for further integration with the advantage of not having to define exchange mechanism between UML and C2/Wright views. With C2/Wright represented in UML the same integration techniques presented in our work may be applied to integrate them as well.

5. **Easterbrook, Finkelstein, Kramer, and Nusbeibeh [Easterbrook et al 1994]:** Their work is strongly based on their concept of *ViewPoints* which is similar to our views. Their work is very close to ours in that it presents some views and corresponding rules to identify inconsistencies within and between them. They further define a formal notation to represent those view. Even though their motivation and initial approach are very similar to ours their work differs from ours in many ways. For one, they focus primarily on other methodologies (other than UML) and sometimes on rather high-level (requirements specifications) views (e.g. the data flow diagram) and they discuss consistency rules primarily for the distributed aspects of software products. Thus, they do not address the problem of information transformation between views. Furthermore, they do not investigate this problem in very detail but instead are content in showing the feasibility of their approach and discuss the benefits of a more rigorous investigation of that problem. We see their work partially as a foundation of ours because it provides useful insight on how to approach this problem but we also see their work as an extension to ours in that they address some distributed/concurrency issues which we do not.

6. **Wang, Cheng, and Richter [Wang et al 1997]:** Their work acknowledges and addresses a deficiency of UML, which we have already discussed extensively above – the lack of precision and formalism. In various articles they propose ways on how to eliminate that problem by presenting formal methods which they integrate with OMT [Rumbaugh 1991]. So they substitute Object Models with Algebraic Specifications, various OMT semantics with Algebraic semantics and Instance Diagrams with Algebras. We consider their work as another foundation to ours in that we also needed to substitute UML with more precise formalism. However, they see the use of these formal extension less in automatic consistency checking but instead in combining the strength of both views – the readability of OMT with the precision of algebraic specifications. Thus, the issue of integrating their views with each other and with other views is not fully addressed; nevertheless recognized.

7. **Schönberger, Keller, and Khriss [Schönberger 1998] and Koskimies, Systä, Tuomi, and Männistö [Koskimies et al 1998]:** These two groups of researchers worked on a similar issue; that of view transformations. Both concentrated on UML Scenarios and how they can be transformed and merged into state diagrams. The latter group additionally developed a tool called SCED which automatically performs these transformations. Although the former group did not come up with an automated support, the algorithms they provide are somewhat more sophisticated in that they also address concurrency issues in the model. Even though both groups did not have view integration in mind when they build them, their work is fundamental when it come to view integration. We will make use of their techniques and a short description of it will follow later. Nevertheless, both their works come short in addressing transformations among other views but scenarios and state diagrams.

8. **Belkhouche and Lemus [Belkhouche-Lemus 1996]:** Their work reflects the opinion that views are independently created and analyzed, however, view transformation systems should be used to transform important design information for consistency checking. Their work lacks a well-formalized foundation and as such the types of inconsistencies and incompletenesses they can identify are very limited. Their work can also be used as an example of the state explosion problem which occurs by sequentially applying transformation algorithms without end. So they show a case of a Data Flow Diagram (DFD) and a State Chart (SC) and identify 15 relationships (!) between those two diagrams and 2 subsequent derivations (through transformation) of each of them. For instance, if the derivation (transformation) f(DFD) yields a SC and g(SC) yields a DFD, then the derivation f(g(SC) yields yet again a SC and so forth. These two derivations could be applied recursively forever, resulting in an infinite number of intermediate diagrams and relationships between them.

9. **Delugach [Delugach 1996]:** Parts of his work is similar to the ones discussed above. So he takes two types of diagrams, e.g. a data-flow diagram (DFD) and a class diagram, cross-

references all items which are named the same (the deficiency of this mapping mechanism is not addressed in his work) and transforms them onto a conceptual graph diagram. What makes his work interesting (and unique) is that he uses an algorithms which verbally describes the relationships of components in that conceptual graph. An architect or user can now read those descriptions (which are in plain English) and reason about whether they make sense. Thus, his work shows another interesting way on how the view integration problem can be addressed, however, his work provides no automatic way of conflict identification and resolution. Nevertheless, it could be very useful to apply this technique in parallel with some more automated means of analysis (such as ours).

10. **Grundy, Hosking, Mugridge, and Amor [Grundy et al]:** Like some of the works above, these researchers address the integration problem by translating diagrams (views) onto a conceptual model (called the repository or base view). Overlaps between the design components are identified through the usage of common names (!). Their work is also rather high level and does not only address software development but also other domains, such as views in building architectures. Another restriction of their work is that they do not address the problem of representing conceptually different views in their base model. Instead they show how this can done between a class diagram and the corresponding source code. Nevertheless, their work adds a very interesting dimension to integration which is not addressed by our work. That of identifying inconsistencies in following a software process model (e.g. some activities were not performed, etc.). As such, they address the product inconsistencies and process inconsistencies as very related problems. This view is also shared by us and we will briefly address this deeper aspect of integration later on in this work.

11. **Engels, Heckel, Taentzer, and Ehrig [Engels 1997]:** The final work presented here is the work of these researchers. Some aspects of their work are again similar to the ones discussed before. In their work their thrive to create a system model out of a number of related views. Like before, their work assumes that the same names are used or a name conversion

dictionary is maintained. Views are then combined stepwise to a system model by merging two views at a time recursively until all of them are merged. They fall, however, short in defining automated analysis mechanisms for consistency checking. Nevertheless, their work can be used to abstract views (currently only object models) into more general object models or class models. The most interesting aspect of their work is, however, something else. They also present methods for showing how an object diagram changes over time. For instance, each method (function) which is called might add, remove, modify, or not do anything to the current object model. Thus, they define methods in term what changes they cause in the object model (e.g. newAccount, if successful, will add one Account object to the object diagram). This kind of knowledge of the impact of methods can then be used for consistency checking; an aspect they did not explore in their work. Our work, however, will make use of that.

The works presented above do not cover the complete picture of what is going on in this field. However, it gives an overview of the major approaches. The diversity of the work above is another reason why this work thrives not to just add another technique. Many ideas described in their research are excellent and, thus, our work tries to also take the best of what already exists and build it into something bigger.

5 View Mismatches

If you fail, fail early – Philippe Kruchten

So far we have talked about the view integration problem in a very high level fashion. This chapter will explore this issue in more detail. First, we will categorize views followed by some examples of architectural mismatches between views of same or different categories (dimensions).

5.1 Views

5.1.1 Dimensions of View Integration

To get a better understanding of how views fit together, consider Figure 8. For our purposes we divide views into three major dimensions; horizontal, vertical and process. The vertical and horizontal dimensions are also often referred to as layers and partitions whereas the process dimension reflects the life-cycle (time). These three dimensions are briefly described in the following. It is worth noting that these

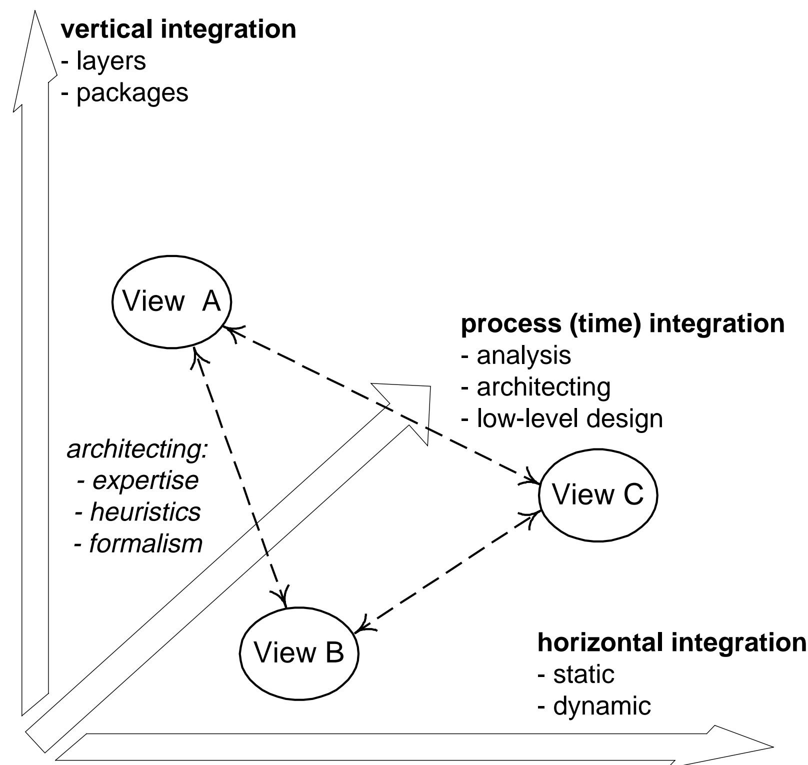

Figure 8: Dimensions of Views

dimensions are also three view integration dimensions since we would like to integrate views within their various dimensions but also across dimensions.

Vertical View Dimension

The vertical dimension reflects the system decomposition. Usually, higher levels (layers) show the system in a more abstract fashion whereas lower levels show the system in more detail. Each layer should represent a *complete* system, however, it might not do so in one diagram. Mismatches between vertical views are therefore mismatches where one layer does not represent the same interrelationships (within and between diagrams) as another layer. In UML, system decomposition is primarily achieved though class/object diagrams and associated packages. Other diagrams, such as state diagrams do not reflect the system decomposition. They may, however, still be used in each layer to repeat the same modeling constructs in an increasingly detailed and abstracted form.

Once a system is decomposed into subsystems, additional refinements (layers) of that subsystem must, then, not reflect the entire system any more but instead only that particular subsystem. If this is the case, other subsystems have to be similarly represented by their own layers and all those (sub) layers together should represent the complete system again. Thus, there may be another category of mismatches between (sub) layers if the partitions of those layers are not fully separated and consistently applied. In some cases, subsystems may not need further refinements because they already are detailed enough whereas in other cases subsystems (layers) may still be in need of further refinements. If this happens, the already adequately refinement subsystem(s) (which do not need further refinements) may also be used (e.g. accesses, called, etc.) by all subsequent refinements of other lower layers so that the entirety may still represent a complete system. This case also shows that the level of detail achieved by a layering is often not clearly defined and it may vary within each layer. The only important aspect is whether the sum of its part represents a complete picture again and that the logical divisions (partitions) of the system into subsystems is reflected consistently on all levels of abstractions.

There are however variations in how this can be done. For instance, the physical design should be a continuation of the logical design. This does however not mean that physical layers have to use the same type of views or the same set of features of logical layer. For instance, the physical design (as compared to

the logical one) should only use modeling elements that can be directly implemented. So, if a system layer is physically described in a class diagram and it is going to be implemented in C (a non-OO language) than some elements of class diagrams, such as inheritance, should not be used in the physical design anymore. Nevertheless, the partitions into which subsystems are divided are still valid and should be the same for both logical and physical views. Thus, it is not required that layers must use the same type of view (e.g. class diagrams).

Horizontal View Dimension

The only restriction we have from the vertical integration is that the set of views used in each layer must represent the entire system completely. However, each layer may still be represented by different sets of views. It may do so using different types views or even different decompositions (although the latter is not recommended as discussed before). Thus, in the horizontal layer, a view is not required to model a complete system (or subsystem depending on the layer).

Horizontal views are frequently further divided into static and dynamic views. The difference of these two groups is related to the presumed execution time. Dynamic views represent the system (or more likely a partition of it) at a particular point of time or time interval. For instance, object diagrams show the objects that exist at a particular time; sequence diagram show the calling dependencies between various objects during a time interval. Dynamic views often represent samples of the state or interaction of the system and its components during their execution. Class diagrams on the other hand are static representations of all allowed dependencies of a system and its components throughout the execution. Views in that category represent more general aspects of the system behavior and their constructs are always applicable.

Most forms of representations (diagrammatic or not) can be used in this horizontal manner. This fact has made some people belief that architectures are 'flat'. However, this is, as Philippe Kruchten said during his keynote address at the GSAW'98, one of the ten major misconceptions about software architectures [Kruchten 1998].

Process View Integration

Although, this form of integration is often ignored, it is actually very important. The integration over time (or process integration) reflects the integration of product artifacts during the life cycle. Note that

we are not speaking of making sure that a process model is followed consistently but to make sure that the changes a product artifact goes through over time are captured. This activity is also often referred to as version control or configuration management but it is also another dimension of the view integration problem. If we loose the rationale why things happen the way they did (or changed the way they did) we loose some important design information.

For instance, if we have two alternative design approach, each with unique advantages and disadvantages, clearly, we would like to make sure that both alternatives are reflecting the same problem and do so completely. So it may make sense to compare those two design approaches, and if it is only to ensure completeness. This integration aspect is, however, only of secondary importance to our work. We nevertheless list it because the activities and techniques presented in this work apply to this form of integration, as well.

5.1.2 Types of Views

Currently this work makes use of the following types of views. For a more detailed description please refer to the UML notation and semantics guide [Booch-Jacobson-Rumbaugh, 1997]. Later on in this work, we will expand on this list and show in a high level fashion how other architectural representations can be included as well (e.g. ADLs).

Diagrammatic views:

- Class/Object Diagrams
- Sequence Diagrams
- State Diagrams
- Interface Diagrams
- Collaboration Diagrams

Textual views:

- Object Constraint Language (OCL)
- Programming languages (C++)

- Attributes and other properties of UML model elements

5.2 Examples of Mismatches

Having defined views in terms of their dimension, we will now complement that by showing more concrete examples of (potential) architectural mismatches between and within those view dimensions.

5.2.1 Mismatch between Class Layers

The first example shows a simplified air traffic control system (see Figure 9). The system is presented in two layers and, as discussed above, each layer is supposed to presents the system in a complete fashion. The first layer shows the interaction of the *Flight* component which has some dependencies with *Mechanic*, *Pilot* and *Flight Controller*. The second layer refines this relationship by decomposing the *Flight* component into *Flight Plan*, *Aircraft*, and *Flight Authorization* - The *Flight Plan* being dependent on the *Pilot*, the *Aircraft* with its instance *Boeing 747* being dependent on the *Mechanic*, and *Flight Controller* being dependent on Pilot.

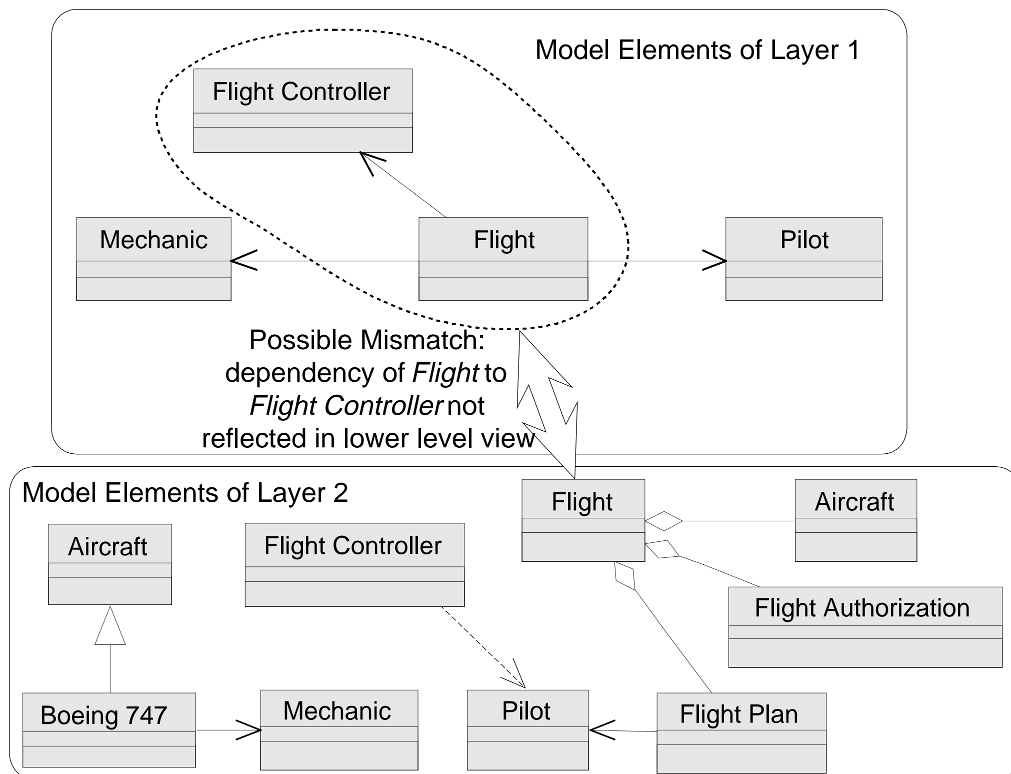

Figure 9: Potential Mismatch between two Layers (Completeness)

Although there is a *Flight Controller* in the lower level diagram, the higher level dependency from *Flight* to *Flight Controller* is not obvious. It would be dangerous to conclude (or in this case incorrect to conclude) that there is a dependency simply because there are lines going from *Flight*, via *Pilot* to *Flight Controller*.

Although, a human analysts would be able to detect this kind of architectural mismatch, for a computer to come to the same conclusion is not obvious. For a trivial example like this one, the need for automated assistance in identifying and resolving mismatches may not be obvious. However, more complex projects involve thousands of modeling elements. There mismatches between views cannot be seen this easily anymore and the task of finding and resolving them becomes very time consuming – frequently having strong effects on project schedule and cost. Thus, automated assistance in identifying and resolving them would result in major benefits.

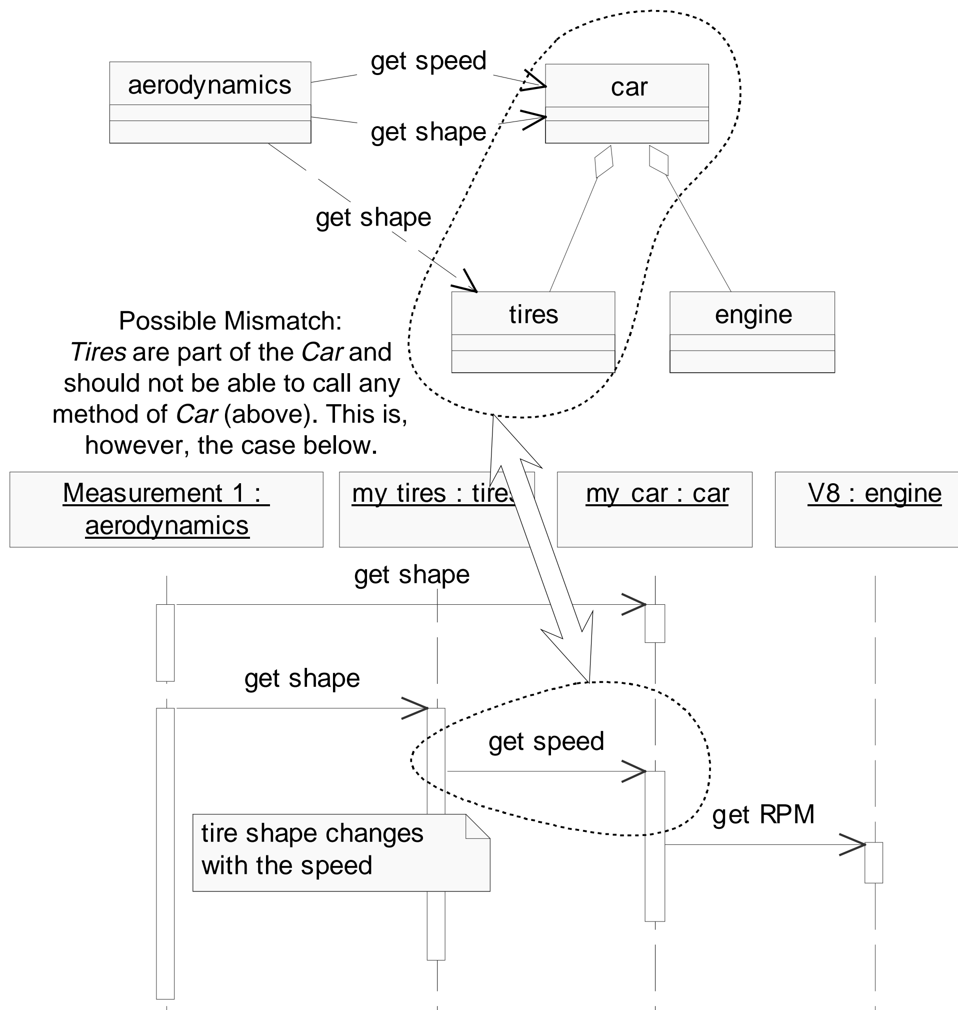

Figure 10: Potential Mismatch between Class Diagram and Sequence Diagram

5.2.2 Mismatch Class and Sequence Diagrams

Figure 10 is another example of an architectural mismatch. The figure shows a class diagram and a sequence diagram of a simplified Aerodynamics System. The *Aerodynamics* component acquires information about a *Car*, which consists of the parts *Tires* and *Engine*, and calculates some aerodynamic behavior of the car. The sequence diagram (below) shows this for a particular instance of *My Car*. A potential mismatch between the two diagrams arouses around the components *Tires* and *Car*. Since *Tires* are part of the *Car*, only the Car object (*My Car*) should be able to call methods in *Tires*. The sequence diagram violates that rule. Possible ways of resolving that issue are either to change the relationship between *Car* and *Tires* or to change the calling sequence (e.g. the latter can be done by having *Aerodynamics* pass along the speed of the car as a parameter).

5.2.3 Cardinality Mismatch

Figure 11 shows an example of a mismatch between a static and some dynamic views. The class diagram (top) shows the static view between a *Patient* and his/her *Visiting Record* during a stay in a

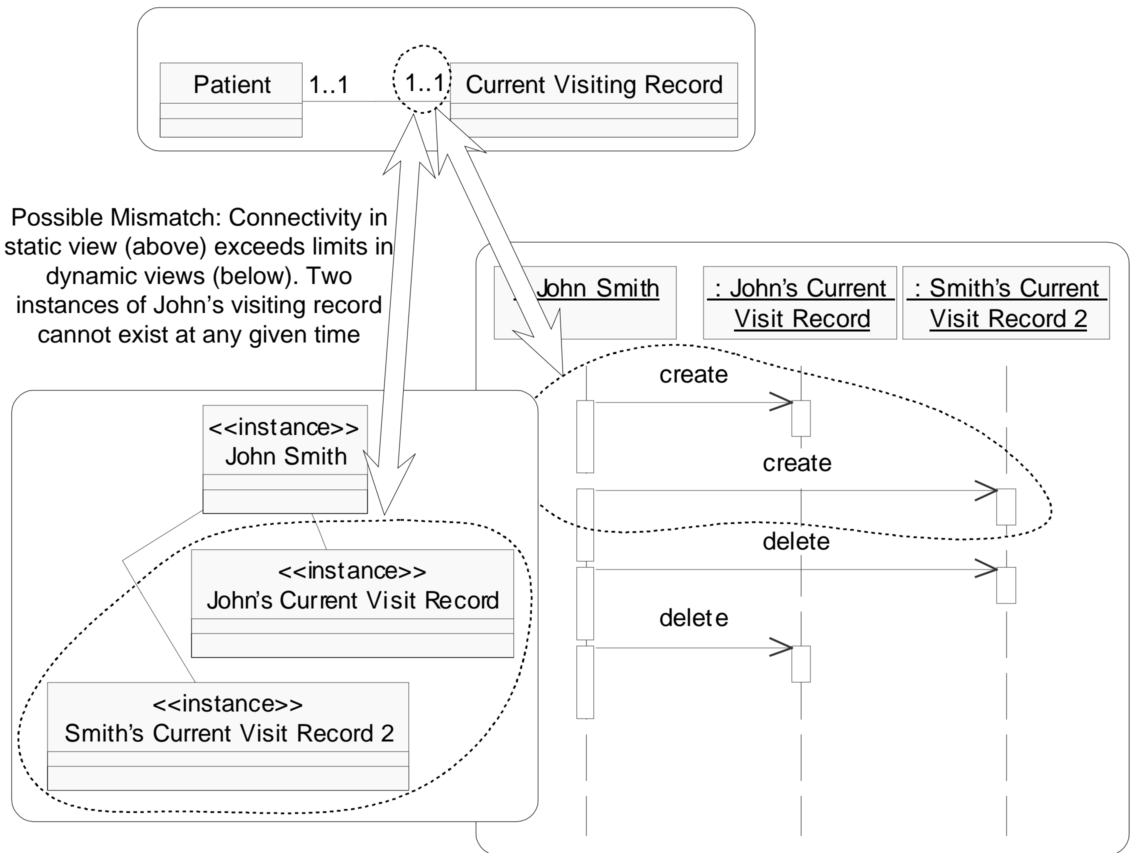

Figure 11: Potential Mismatch between a Static View and two Dynamic Views

hospital. Even though a patient may have stayed in the same hospital more than once before, he/she should nevertheless have only one current visiting record at any given time. This static rule is violated in both, the object diagram (lower left) and the sequence diagram (lower right). The object diagram shows an instance of Patient *John Smith* and it also shows that he has two current visiting records. Similarly, the sequence diagram shows that a new visiting record for John Smith is created even though one already exists. Please note that both dynamic diagrams are consistent. The inconsistency is only between the class and object diagram as well as between the class and sequence diagram.

5.2.4 Mismatch between State and Collaboration Diagrams

The last mismatch sample, discussed in this chapter is depicted in Figure 12. It shows a more complex setting in that it becomes less obvious that there are potential mismatches. It shows another perspective of the hospital system we discussed above. Here we see the system from a clerk's point of view, who is using the screen to create visiting records for patients. The state diagram of the *Screen* class (top) shows that information about a patient is entered and validated. Afterwards, a visiting record for that patient is created.

The sequence diagram (bottom) shows that data is validated, patient information is retrieved and, for a given patient information not found, a patient and a visiting record is created. On the other hand, the state diagram of the *Screen* class (top) shows that information about a patient is entered and validated and after the patient database is checked a visiting record is created.

The sequence (lower-right) and collaboration (lower-left) diagrams conflict because either *request patient data* is missing or named differently (assuming that *get patient data* is an equivalent method). There are further potential mismatches. So for instance, both the collaboration diagram and *Screen's* state diagram have one component each which is not accounted for in the other diagrams. In case of the collaboration diagram, the class *DB* is introduced which could be the database for *Patient* and *Visiting Record*. However, this interaction is not shown in the sequence diagram which may indicate some incompleteness. Furthermore, the *User Input* of the state diagram is not reflected in both, the sequence and collaboration diagrams.

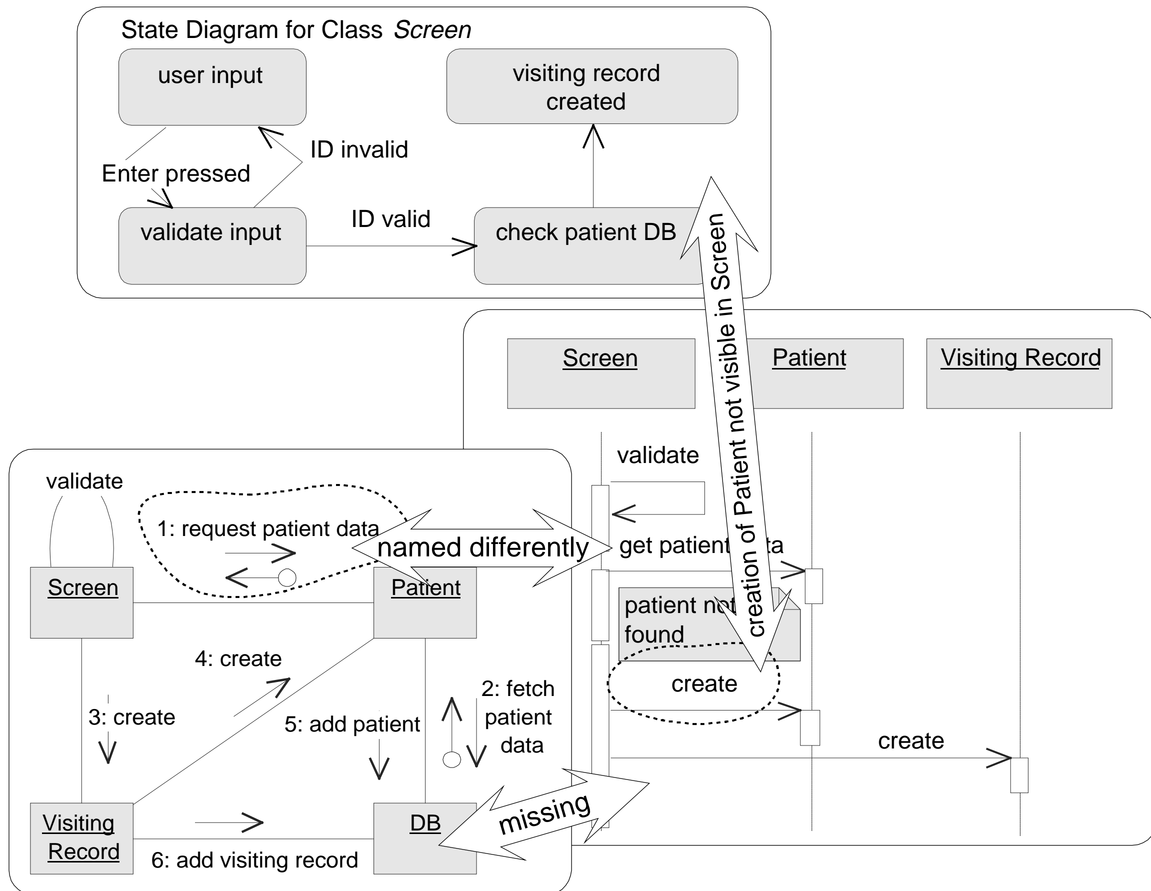

Figure 12: Potential Mismatch between State-, Sequence- and Collaboration Diagrams

Figure 12 also adds another problem to our view integration challenge. This example does not always use the same names for same/similar things. For instance, *get patient data* and *request patient data* may be identical and for us, the architects, this may be obvious, but for the computer this is not. Thus, we are confronted with the challenge of automatically identifying this relationship.

The other problem we can see in Figure 12 is even more severe in that even if we use consistent naming, it is still not obvious what parts correspond to what. For instance, to which model element in the sequence or collaboration diagram does the state *Visiting Record Created* relate to? If we would say the create arrow that calls *Visiting Record* from *Screen* we might be close but this is not completely true. The *create* arrow just calls the method. No *Visiting Record* and no *Patient* have been created at that point. So the state *Visiting Record Created* clearly does not correspond to that arrow but instead it corresponds to the point when the *create* method is finished and execution control is returned to *Screen*. So the state *Visiting*

Record Created corresponds to the void after the *create* method when the control is returned to *Screen*. In other words, that state does not relate to any sequence diagram model element in particular.

Whereas the naming problem above can be approximated with the use of a naming dictionaries (which keeps track of all synonymous model element names), finding an adequate way on how to relate state diagrams with sequence diagrams is less simple.

6 View Integration Framework

No model is correct, but some are useful – Albert Einstein

Having identified views and their dimensions, as well as having shown examples of possible mismatches, we will now elaborate further on what we have to do to integrate those view. As we mentioned before, integration is primarily about identifying inconsistencies. [Nusbeibeh et all 1994] refined that and wrote that the term inconsistency indicates that some form of rule has been broken which expresses the relationship between development objects (model elements). It is these kind of rules we are aiming for. However, rules alone are not useful if they cannot be applied automatically to check for consistencies. What this mean is, that there is more to view integration than consistency rules. What we need is an environment where those rules can be applied. This section will present such a framework and necessary activities.

6.1 Model-based Development

As discussed previously, views are nothing more than an abstraction of relevant information from its model. Views are necessary to present those information in some meaningful way to the user (developer,

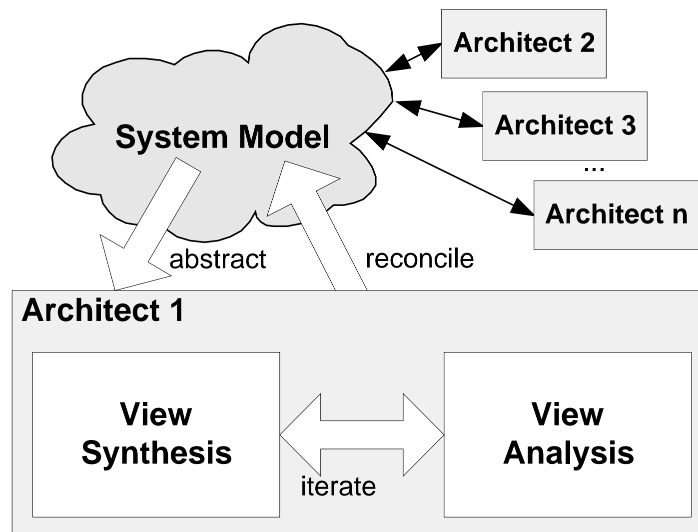

Figure 13: Model-based Development - a view independent representation

architect, customer, etc.) – something the model does not do. When we talk about the need for tightly coupled views, we are really talking about the need of having an integrated model that is adequate in representing views (and their various instances). There, views may be abstracted (derived) from the model, worked on, and then reconciled with the model to ensure that the changes are still consistent with the model. So the model contains a union of the information of all its views, however, with as minimal duplication as possible. In that respect we are talking of the model as a ***View Independent Representation*** (see Figure 13).

Views and the View Independent Representation (VIR) is a paradigm shift which is similar to that of functional vs. object oriented programming. In OO development we do not see functions as the components of the system but instead we see the system as objects that have functions (behavior). Similarly, in VIR we do not see views as components of the system representation but instead we see the system representation as consisting of models which have views (or viewpoints). As functions give meaning to objects, views give meaning to models. We are speaking of model-based software development.

This implies that the model is more than the sum of its views (contrary to what most development models are today, e.g. UML). The stakeholders (e.g. architects) can then derive views from that model, fill in the missing blanks, and reconcile the changes with the model. This means that all information about a software system is captured with as little redundancy as possible in the model even though the views, which are derived from that model, may repeatably use the same information and, thus, have redundancy.

Naturally, this concept of a view independent representation is not an easy one to come by. Actually, one can see this approach as being the opposite of the standalone views (analogy to stovepipe systems) we have these days. Brig. Gen. Mitchell, outlined these two extremes in a different setting during his GSAW'98 keynote address. So he spoke of the Etruscan people, who lived in Northern Italy between 1000BC and 505BC. They were eventually conquered by the Romans because their clans did not unite against their common threat but instead tried to face it everyone for itself. As the other extreme, he spoke of the Autoshave system which an inventor tried to patent. The Autoshave system is a mask with razors inside. The idea is to only put on the mask and the face would get shaved automatically. When confronted with a side effect of his mask – that his invention would cut anybody whom the mask would not fit precisely – he just replied 'well yes, but only once' (!).

The view independent representation (VIR) is the counterpart to the standalone views in that it tries to represent all views. Some views may, however, be just too different from the other ones and, thus,to difficult to integrate into a common system model (e.g. ADL components and connectors). Nevertheless, model-based development may still work for most views (or at least most information within those views). Thus, we aim to find a reasonable compromise between the Autoshave and Etruscan approach.

There is, however, also another reason why we would like to have a common base model. Consider, for example, Figure 14, where we can see the complexity involved in integrating views with each other. In order to share information gained from one view with all other views, each of them would have to be somehow integrated with **all** the other views. In our example we have six views which, in absence of a common reference model, would require to be integrated in 15 (!) different ways. Seven views would already result in 21 different ways of integrating them. Thus, each additional view would force (n-1) additional ways of integration if n is the number of all views to be integrated. In total n (n-1) / 2 ways of integration are required for n views to be fully integrated.

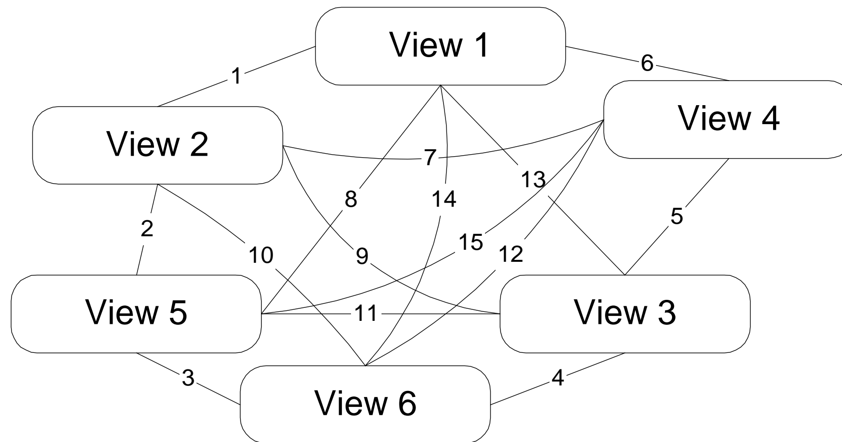

Figure 14: Complexity in Integrating Views

Clearly, we are confronted with a non-linear explosion of integration work (n^2 for a big n). The concept of VIR (or model-based development) would, however, reduce the view integration complexity to a linear problem as depicted in Figure 15. With the existence of a VIR, the integration work would be reduced to mapping or translating each view so that it is fully (or sufficiently) represented in the VIR and then we could define consistency and completeness rule based on the VIR. Thus, each view needed only to

be translated once and all consistency and completeness rules needed only to be represented in one type of style (language, etc.) and not in a view-dependent form. Therefore, this work will utilize the concept of VIR and build on that.

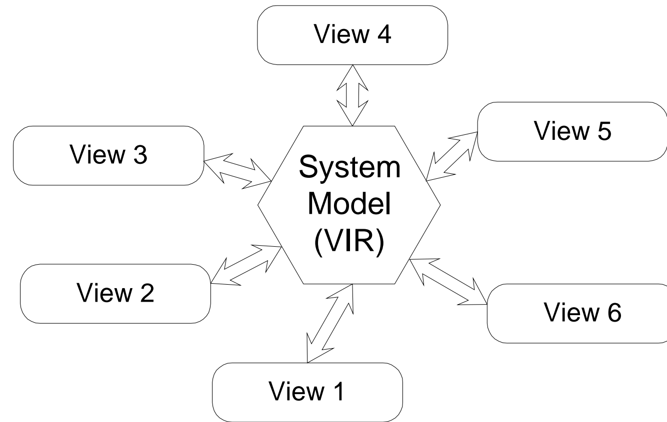

Figure 15: Linear Integration Work using VIR

6.2 Integration Activities

In the beginning of this section we explained that there is more to view integration than consistency rules. What we need is an environment where we can apply those rules in a meaningful way. This work, therefore, introduces a view integration framework that describes integration activities. We will present some corresponding integration techniques that may be seen as instances of activities in the next chapter.

Our integration framework was already illustrated in a high-level fashion in Figure 13. There, a system model was used to represent the knowledge base of the designed software system. Software developers use views to add new data to the knowledge base and to review existing data (view synthesis). Interacting with both, the system model and the view synthesis, is the view analysis. As soon as new information is added, it can be validated against the system model to ensure its conceptual integrity. Figure 16 shows how the view analysis can be further subdivided in three major activities – *Mapping*, *Transformation*, and *Differentiation*.

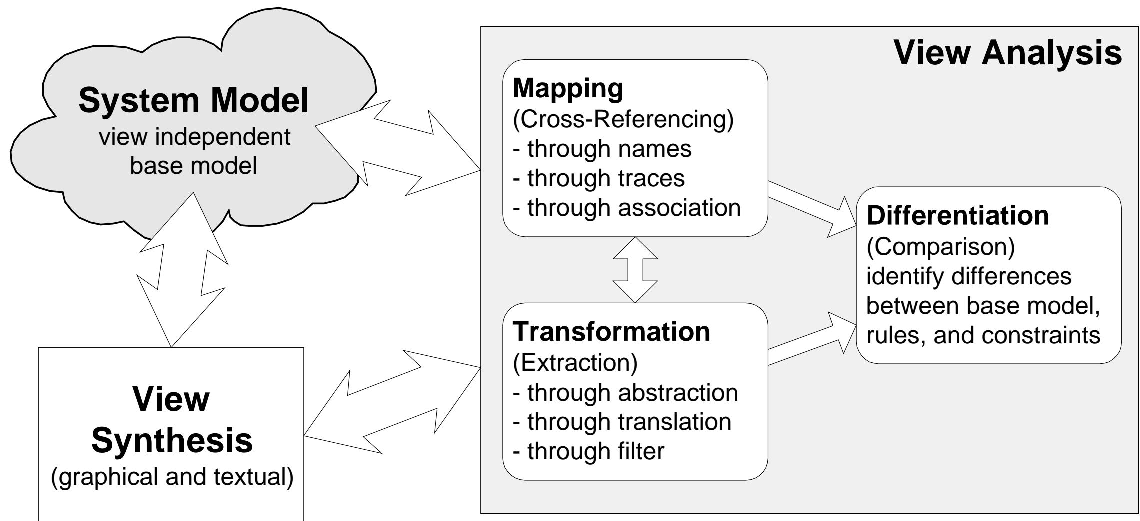

Figure 16: View Integration Activities

- **Mapping:** Identifies related pieces of information through the use of naming dictionaries (manual process), traces and trace simulation (e.g. the use of same physical classes and methods), and certain forms of associations/patterns (e.g. common interfaces).
- **Transformation:** Manipulates model elements in views so that they (or pieces of them) can be shared with other views (or in representing the system model itself). For instance, we may use abstraction techniques to generalize a detailed diagram, we may use view translations to exchange information, or we may rearrange model elements (or pieces) in different manners to create new perspectives (e.g. merging or splitting).
- **Differentiation:** Traverses the system model to identify (potential) mismatches within the system model. (Potential) mismatches are described in form of rules and constraints. Furthermore, mismatch resolution rules can be associated with the mismatch identification rules to propose options on how to resolve them. Differentiation is only possible because new information about model elements is made available through *Transformation* and their relationships are defined through *Mapping*.

It must be noted, however, that these activities are not orthogonal to each other. Obviously, we can only make useful transformations if we know the proper mapping of model elements. However, what may not be obvious on first glance, this relationship is also true in reverse. Information derived through view

transformation can clarify many ambiguities in the mapping. Thus, one view/activity may be used to clarify ambiguities in other views/activities. To apply the ADL framework onto our view integration approach, *Transformation* corresponds to components (e.g. boxes and arrows in diagrams) and *Mapping* corresponds to Connectors (e.g. relationships between boxes and arrows). The configuration derived through *Transformation* and *Mapping* is the foundation for analyzing the conceptual integrity (*Differentiation*) of the system.

6.3 Mapping

In order to deal with inconsistent naming of modeling elements some form of mapping activity is needed. In a simple case, mapping is done through some traces or name dictionaries. However, since both of those have to be done manually, they not only add an additional source of possible defects but also increase the manual overhead in applying integration techniques. The goal is therefore to have some automated mapping technology. The following methods can be useful in identifying mapping.

Mapping through Common Names

This should work at least for all components of the same type. E.g. if there are two diagrams that both use a class with the same name than it can be assumed to be the same. This technique does not necessarily work for connectors or components of different types (e.g. name of class vs. name of state). In case of connectors, the situation is similar (same names may still imply same behavior for connectors of the same type) If a component (e.g. class) exhibits the same set of connectors (methods, attributes) this may also be used to derive some relationships between them (e.g. instance or inheritance). We distinguish between basically four cases: multiple connectors same/different type; multiple components of same/different type. Mapping has to deal with all those in different ways.

Mapping through Design Features

Certain configuration characteristics such as feedback loops or shortcuts (e.g. in state diagrams) can be used to reason about mapping. The SCED technique we will present later on shows a nice example of that. There we use a tool to transform sequence diagrams into a state diagram with the purpose of comparing the

transformed view with existing state diagrams. In an ideal case, where the original view and the derived/abstracted view are consistent, both views should exhibit patterns like the ones above in the same sequence (not necessarily at the same position) regardless of what names are used.

Mapping through Design Patterns

If a high (or low) level diagram uses a component of a particular pattern then characteristics of that pattern can be used to identify its corresponding components in the lower (or higher) level diagram. For instance, [Gamma et al 1994] defined various types of design patterns such as *Iterator*, *Observer*, and *Command*, to name a few. These patterns exhibit unique characteristics that may be used to identify them in other level diagrams. It is assumed that their existence is known, e.g. through the use of stereotypes in UML (see Figure 17).

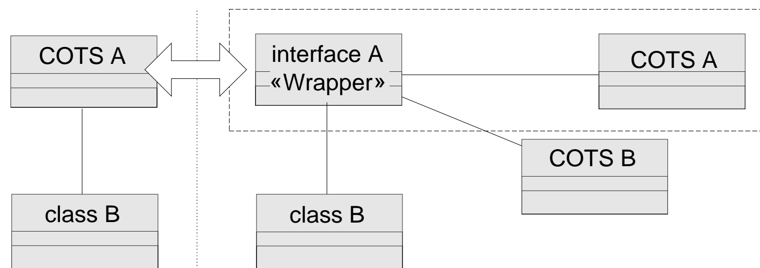

Figure 17: Mapping through Design Patterns or Styles

Mapping through Common Interfaces

In cases where some mapping for some components exist but not for others, the interconnectivity may be used to identify corresponding model elements. For instance, if there are two views which contain components A and B and both views are connected via some helper classes which are named differently then it could be possible that those helper classes are identical.

Also the type of connectivity (e.g. dependency relationships in class diagrams) can be used to identify mappings. For instance classes may be grouped into three categories, *Actor*, *Agent*, and *Server*, corresponding to the whether they provide only services, trigger requests or mediate requests. This information in turn may be used to identify the layering of components (e.g. a actor class is probably not part of the functional framework, or a server class is probably not a user interface component).

Mapping through Similarity in Name

Frequently it can be seen that the names of components and connectors are not identical, but nevertheless similar. Using the similarity of names is, therefore, another mapping technique.

Mapping through Observing Test Scenarios during System Execution

Observation of test cases while they are executed on a software system (or part of it) can as well help in establishing mapping. In this case we combine the system model with the actual implementation and execution of the software product to identify dependencies between them. The identified dependencies are useful for many things but foremost for the verification of the conceptual integrity of the software model by cross-referencing the product implementation with its corresponding system/software model. As such the most significant byproduct are traces between the model elements and the actual implementation of the software product (the solution). Using these traces enables us to further reason about architectural interdependencies based on what implementation pieces they share (or have in common). We will discuss the TraceObserver technique in more detail later.

Having described techniques for mapping, it must be stressed that those techniques must not always yield correct results. It could very well be that different interpretations are possible upon which either all are automatically investigated or a human being must be consulted to make a decision as to what is correct. Furthermore, no single mapping technique will likely be sufficient. It is more likely that a combination of above techniques and others are necessary to yield a higher collective effectiveness. All that also applies to view transformation.

One of the major challenges of integrating views is to figure out where they overlap, what information they can exchange, and what might be useful for other views. Since, we do not know in advance what naming conventions architects follow (if any) while designing their systems we cannot make the simplistic assumption of the existence of consistent names. This deficiency was already visible in many integration approaches we discussed in *Related Work* before.

Mapping can also be integral in avoiding (or minimizing) the state explosion problem. For instance, if a model element in one view is changed, propagating the changes to all affected other model elements require the identification of all affected components (=> mapping). Summarizing, we can say that *Mapping* depends heavily on the existence of view redundancy. Without view redundancy, we would not be able to establish mapping. This shows the dilemma we are in. For one, we need redundancy to identify relationships between views but on the other hand it is this redundancy which is the cause of inconsistencies in the first place. Thus, redundancy is a necessary evil.

6.4 Transformation

The issue of transformation is often considered the core of view integration. Although this is not entirely true as we discussed above, view transformation nevertheless constitutes a major challenge because the modeling information provided by one view are not readily compatible with other views. Thus, we need transformation to translate views in such a way so that their elements can be shared, exchanged, or compared (same types of model elements and the same level of abstraction). As a rule of thumb there are basically two categories of translation techniques:

1. The transformed view uses modeling elements of the same type as the source (e.g. source was a class diagram and so is the transformed view)
2. The transformed view uses modeling elements of different types as the source (e.g. source was a sequence diagram and transformed view is a state diagram)

Transformation through Abstraction

Abstraction ensures that modeling elements express information in the same/similar level of detail. We will present an abstraction technique in the next chapter when we introduce Rose/Architect. There we have a technique which parses lower-level class/object diagrams and abstracts (simplifies) them to yield higher-level (less complex) diagrams. Although, the Rose/Architect tool only works for class diagrams this concept can easily be expanded to capture other views, such as sequence diagrams, state diagrams, etc.

Transformation through Translation

The SCED technique we will describe in the next chapter will be an example of how information can be translated into another view. Key problem in this approach is that the semantic meaning of information is not the same any more and this may cause problems in the process of translating modeling information. This means that components (boxes) in one view do not match components in other views. The same applies for connectors. More often than not, translation requires input from various resources, sometimes even humans since one source view does not often provide all necessary information.

Transformation through Patterns

Patterns are again useful in transforming information. In *Mapping* we introduced them in order to find related information based on the patterns they exhibit. However, patterns can also be used in translating information. For instance, we could abstract information based on the patterns it recognized in a lower-level diagram and, thus, replacing them with simpler (higher-level) patterns. Patters are, however, also useful in transforming between different types of views. For instance, if we have a database containing standard patterns and how they would be represented in different styles (or views) then we can use that information for view transformation.

Transformation through Common Components

In cases were modeling information about a particular aspect of the design is scattered in different locations (diagrams, views), we can create new views out of combining information from various locations in a different manner. Take for instance the state diagram in Figure 18 that has two state transition scenarios (A->C->E and B->C->F). Both scenarios have the state C in common. This state can be used as a point of interaction between both scenarios. Thus, we can create a new state diagram A->C->F. This concept also works for other views such as class diagrams, sequence diagrams and collaboration diagrams.

Another example would be when a single diagram does not depict all related modeling information. There, a new diagram could be created that merges information (interaction, properties, etc.) of that modeling element in a different way.

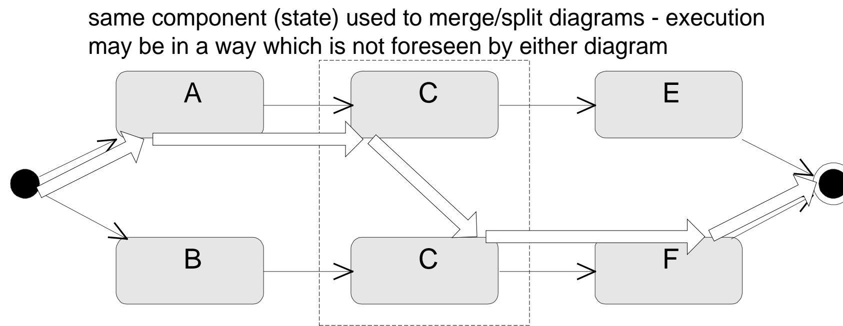

Figure 18: Transformation though common components and rearranging substructures

Replacing a substructure of a diagram (or a modified version of thereof) by another structure is another example. For instance, we could replace a high-level component by its corresponding lower-level components or vice versa without changing the remaining elements in that view.

Transformation through Trace Observation

This technique was discussed in mapping but it can also be applied to view transformation. So for instance, we can create object models and object interaction scenarios based on observations of the system execution. Multiple object models can then be abstracted into class models and so forth.

6.5 Differentiation

Transformation enables us to share modeling information with other views and *Mapping* enables us to classify modeling information to identify relationships among them (correspondence of information). Having this established, the last integration activity is *Differentiation*, where views and derived views (viewpoints) are compared in order to identify (potential) mismatches between them.

The term *Viewpoint* was adopted from [Nuseibeh et al 1994] who defined viewpoints as being pieces of views. In our case, viewpoints are mainly derived or abstracted views that provide another viewpoint or perspective to the view they are compared with.

Since it would be too difficult to create differentiation techniques for all possible views and viewpoints, we will make use of the VIR concept introduced previously. Thus, our work uses translation and mapping to create a view independent representation (VIR) model of the system which captures view and viewpoint information in some defined manner. The *Differentiation* activity then traverses that model

in order to identify mismatches within it. This latter comparison can be done using two principal techniques:

1. (Graph) Comparison Algorithms: In case a viewpoint shows a configuration of several modeling elements of similar type and level of abstraction, then a comparison may be done by simply traversing its nodes (components) and connectors.

2. Constraint/Rule Checking: Viewpoints often only express pieces of information about a modeling element but not necessarily their configuration. Thus, these pieces may be captured in form of rules and constraints (e.g. using UML's Object Constraint Language) which in turn may be validated against each other and other modeling information.

6.6 Identified View Mismatches

Having talked about view integration activities, the following will now focus on what kind of architectural mismatches exist. The next chapter will then describe (semi) automatic ways of identifying them. With mismatches we understand primarily inconsistencies (conflicting constraints). Incompleteness is a refinement of inconsistency in that it represents an inconsistency with the corresponding higher-level software system representation (e.g. software system requirements, architectures, etc.).

Section 5.2 already showed a number of mismatch cases. This section will now generalize them,

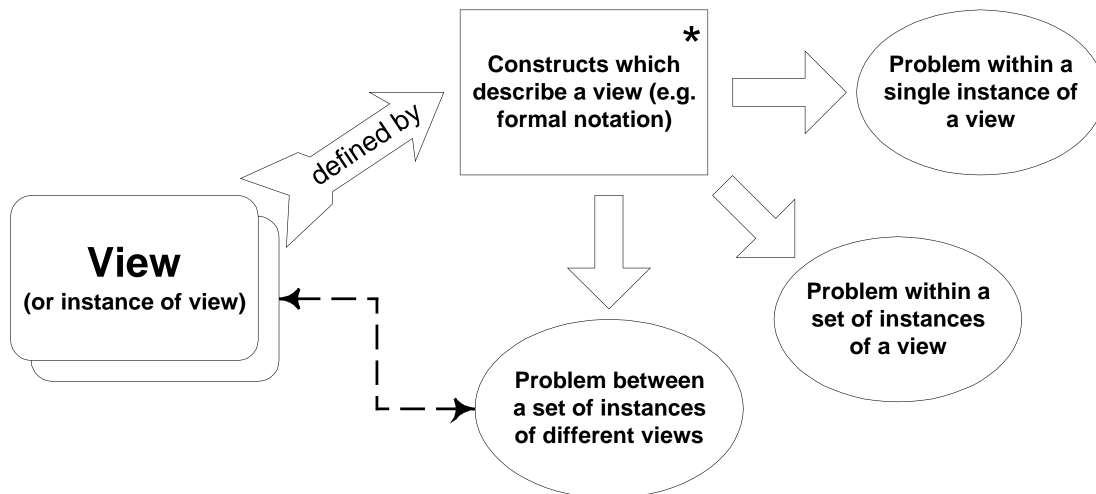

Figure 19: Categories of Mismatches

describe them, and group them. As such this section 1) identifies constructs (e.g. what is a layer) and 2) identifies constraints/mismatches within and between these constructs (e.g. object interaction may not skip layer) – see also Figure 19. With respect to view mismatches we distinguish between three basic types: mismatches between a single instance of a view, mismatches between a set of view instances of the same view type (e.g. high and low level class diagram – e.g. Figure 9), and mismatches between a set of view instances of different view types (e.g. sequence and state diagram – e.g. Figure 12).

It must be stressed, however, that the following list does not represent a complete representation of mismatches. It is not our goal to be as complete as possible at this stage but instead to be as a complete as necessary in order to create a meaningful integration model. This list will be expanded as the research progresses.

Layer

- If there are layers – observe whether the objects in these layers communicate through the proper channels. E.g. their interactions are not supposed to skip layers and are only allowed to call components in the same layer and the one underneath. Issues: Can we automatically detect layers? And if yes, which objects belong there?
- Missing Interaction of Components: Ensure whether component interaction in higher-level diagrams is reflected in lower-level diagrams. For instance, Figure 20 below shows two class diagrams. The upper

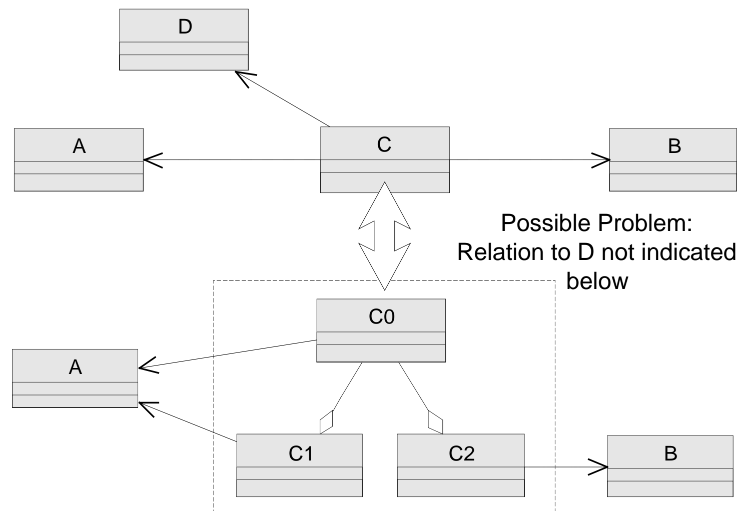

Figure 20: Potential Interface Mismatch

one shows the interaction from class C with the classes A, B, and D. The lower diagram shows a more detailed C, which is broken up into its subcomponents C0, C1, and C2. The lower diagram also shows the interaction with classes A and B (as the one above) but fails to show the interaction with class D. This missing interaction must not necessarily be a fault since the interaction of C0-C2 with D may be omitted in the lower view (e.g. it may not be relevant for the lower view). Thus, it must also be verified that none of the components C0-C2 are referring to D in any other view.

- Diagram does not depict components of the same importance: If we assume that D would be a part of C in above example then this may indicate that the higher level is depicting a low-level component (D). D is only a sub component of C and thus should not be relevant in the higher-level view.

Component/Connector Incompatibility

- Wrong Direction of Call/Trigger/Message: Figure 11 showed the case of a connector between two classes (aggregation) which do not match the calling direction in the sequence diagram.
- Wrong Cardinality: Figure 21 shows a case were the cardinality between two classes is not reflected in object diagram.

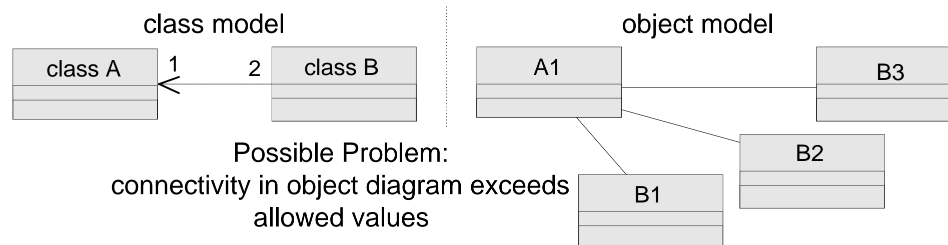

Figure 21: Potential Connectivity Mismatch

- And/Or connections (e.g. in state diagrams) must also be reflected in CollD.
- Verify 1:1 (or 1:many) mapping between Analysis, Design and Implementation: This can be done via the traces which were observed during execution. If design traces are ‘part of’ analysis traces then the one-to-one mapping has been observed (-> proper partitioning)

- Bi-directional calls are not allowed for some types of connectors: E.g. Object A calls object B and vice versa although A is part of B (aggregation relationship).
- Labeled connectors do not match attributes: E.g. a class diagram may describe methods as attributes. A sequence diagram may then use those methods. Figure 22 shows such a case (it is assumes, however, that both diagrams correspond to the same level of abstraction).

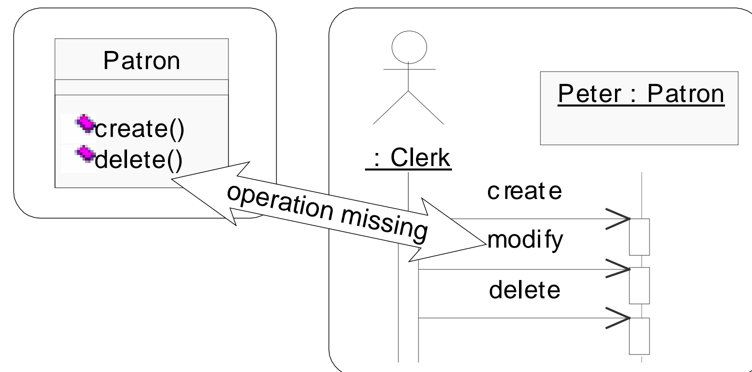

Figure 22: Mismatch between Class Operator and Sequence Diagram

Styles and Patterns

- Style or Pattern Characteristics Violated: For instance, if a view is categorizing components (e.g. classes) into layers then all other views (with their classes and interactions) must also observe the calling dependencies of that layered architecture.
- Pattern definition of higher-level diagram is not reflected in lower-level diagram.

Process

- Type of view or modeling element should not be used at some development stages: E.g. the physical view limitation. In the physical view only those model elements should be used that are reflected in the implementation program language. E.g. if the inheritance connector is used in the physical view but the implementation language is ‘C’ than there is no simple (one-to-one) mapping between the physical view and its implementation.

6.7 Assumptions

In order for some mismatch rules to be meaningful, assumptions about views and their usage will have to be made. For instance, the following are possible assumptions about layering.

Layers

- If a component is not refined in layer n+1 then layer n's version is latest.
- A lower-level component may be connected to a higher level component if and only if there is also a connection between that higher-level component and the corresponding higher-level component of the lower-level element.

This list needs to be expanded and therefore more assumptions of this nature will be specified at a later stage of this work. The list of assumptions can be seen as constraints opposed by the view or system to be modeled and against which mismatch rules are compared. The next chapter will talk about this issue in more detail.

7 Automating the Mismatch Identification

The whole of science is nothing more than a refinement of everyday thinking – Albert Einstein

The goal of the view integration framework presented in the previous chapter was not only to point out integration activities and concepts on how to deal with mismatches but also to create a foundation for automating the mismatch identification and resolution process.

7.1 Providing Techniques to Identify Mismatches

This section introduces several view integration techniques. The reason why we distinguish between view integration activities and view integration techniques is because the techniques often tend to overlap in their abilities to satisfy mapping, transformation, and differentiation. They also often tend to be too imprecise, and thus, we need to make use of different techniques in order to satisfy integration activities. Like before, we are confronted with two primary choices on how to use and integrate techniques for our purpose.

1. Integrate all techniques with each other
2. Integrate techniques with activities and all activities with each other (system model)

These choices are not coincidentally similar to the choices we had with respect to integrating views. Again, we see the need for applying a system model as a foundation to integrate views simply because integrating techniques would result in a non-linear increase in integration complexity.

We furthermore present those techniques in the context of the mismatch examples in section 5.2 above. There we asked the question whether it is possible for a computer to identify those mismatches automatically. Because of the different semantic meanings of the modeling information in views, this did not seem to be an easy task. This section will introduce three techniques, called SCED, Rose/Architect, and TraceObserver that can be used to transform above examples in such a form that a direct and automatic comparison of their modeling information becomes possible.

7.2 Using Rose/Architect to address the Layering Mismatch

In order to deal with the layering mismatch example in section 5.2.1, we need a mechanism which is able to abstract (simplify) class diagrams. For that we can make use of Rose/Architect (RA), which utilizes patterns and heuristics to deal with that issue. This technique takes advantage of the fact that some structures in views (e.g. collections of classes and their relationships in class diagrams) exhibit some recurring characteristics or patterns that can be identified and replaced.

Rose/Architect [Egyed-Kruchten 1999] does that by identifying patterns of three classes and replacing them with simpler patterns using transitive relationships. In class diagrams, a transitive relationship describes the relationship between classes that are not directly connected. A relationship may, however, exist through other classes (e.g. helper classes) which form a bridge between them (e.g. in case of our example, *Aircraft* and *Mechanic* are not directly connected but a relationship is still given through the helper class *Boeing 747*). Thus, if some formula is discovered which could derive transitive relationships with sufficient accuracy, then some automatic support in simplifying and abstracting class diagrams could be provided in tool form. This would allow architects to abstract important classes from existing, more detailed models by eliminating the 'helper classes' and, thus, would further enable them to portrait and

Table 1: Some Abstraction Rules from the Rose/Architect Model

<table>
<tr><th>Rules</th><th>Component</th><th>Connector →</th><th>Component</th><th>Connector →</th><th>Component</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">Class</td><td>Generalization →</td><td>Class</td><td>Generalization →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">Generalization →</td></tr>
<tr><td rowspan="2">2</td><td rowspan="2">Class</td><td>Generalization →</td><td>Class</td><td>Dependency →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">Dependency →</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">Class</td><td>Generalization →</td><td>Class</td><td>Association →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">Association →</td></tr>
<tr><td rowspan="2">4</td><td rowspan="2">Class</td><td>Generalization →</td><td>Class</td><td>Aggregation →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">Aggregation →</td></tr>
<tr><td colspan="6">[…]</td></tr>
<tr><td rowspan="2">55</td><td rowspan="2">Class</td><td>← Generalization</td><td>Class</td><td>Aggregation →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">Aggregation →</td></tr>
<tr><td rowspan="2">56</td><td rowspan="2">Class</td><td>← Dependency</td><td>Class</td><td>Generalization →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">×××</td></tr>
<tr><td rowspan="2">57</td><td rowspan="2">Class</td><td>← Dependency</td><td>Class</td><td>Dependency →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">×××</td></tr>
<tr><td rowspan="2">58</td><td rowspan="2">Class</td><td>← Dependency</td><td>Class</td><td>Association →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">×××</td></tr>
<tr><td rowspan="2">59</td><td rowspan="2">Class</td><td>← Dependency</td><td>Class</td><td>Aggregation →</td><td rowspan="2">Class</td></tr>
<tr><td colspan="3">← Dependency</td></tr>
</table>

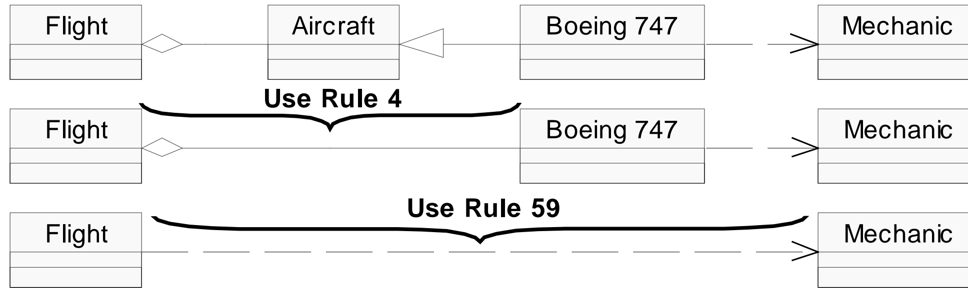

Figure 23: Using Rose/Architect to abstract (simplify) class diagram

analyze the interrelationships between classes even if the classes were scattered in different locations throughout the model (e.g. in different diagrams, or in different packages and name spaces).

RA provides this mechanism and [Egyed-Kruchten 1999] describes this technique in much more detail. Table 1 shows an excerpt of the rule set defined in RA. Rule 4, for instance, describes the case of a class which is generalized by a second class (opposite of inheritance) and that parent class is an aggregate (part) of the third class. This three-class pattern can now be simplified by deleting the middle class and creating a transitive relationship (an aggregation in this case) which goes from the first class to the third one. The underlying RA model describes these rules and how they must be applied to yield an effective result.

Figure 23 shows those RA refinement steps using above rules in the case of the *Flight* to *Mechanic* relationship of our example. After applying two rules (rules 4 and 59 respectively) we get a simplified pattern of two classes and a dependency relationship between them. If this is also done for the other classes *Pilot* and *Flight Controller*, we get an abstracted version of the layer 2 class diagram (see Figure 24). This abstracted view can now be compared directly with the original class diagram we used in Figure 9. Thus, through the use of RA we are now able to convert one view so that it represents information in a very similar manner as in the other view. Comparing views can now be done by simply using some graph comparison algorithm.

This integration technique has, however, the drawback that it can only be used to abstract information of the same type of view (e.g. class diagram). Although, this technique could also be applied for other types of views, it still does not help us when we want to compare modeling element of different types of views as this is the case in with the other examples. The next section addresses that issue.

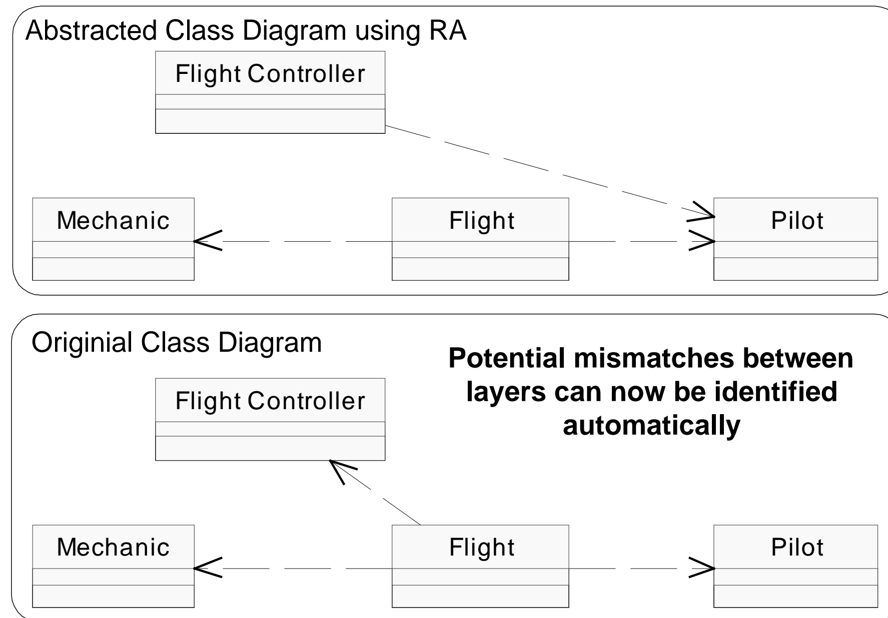

Figure 24: Layering Mismatch more obvious through common base line

7.3 Using SCED to Address State/Sequence Diagram Mismatch

Example 2 in Figure 12 showed an architectural mismatch between a state diagram and a sequence diagram. In this view, it is even less obvious how information can be compared, mainly because the meaning of the components and connectors (boxes and arrows) are not the same anymore. Although it may appear that the states correspond roughly to the arrows in the sequence diagram, this is not correct. Take for instance, the state *checking patient DB* which seems to correspond to *get patient data*. It is clear that the former (*checking patient DB*) includes the action of getting the patient data but this function also needs to check whether this operation was successful. There is also another distinction between those two diagram types. The state diagram shows the generic case of what states the *Screen* may go through. On the other hand, the sequence diagram is a sample (test) case. It is perfectly legal to have different sample cases (e.g. say another one where patient is found in the DB) matching the same state diagram.

In order to be able to compare those two types of diagrams we will make use of a diagram mapping technology developed by [Koskimies et al 1998] and [Schönberger et al 1998]. Both have independently created technology for the purpose of transforming data between sequence diagrams and state diagrams, the former group also having built a support tool called SCED.

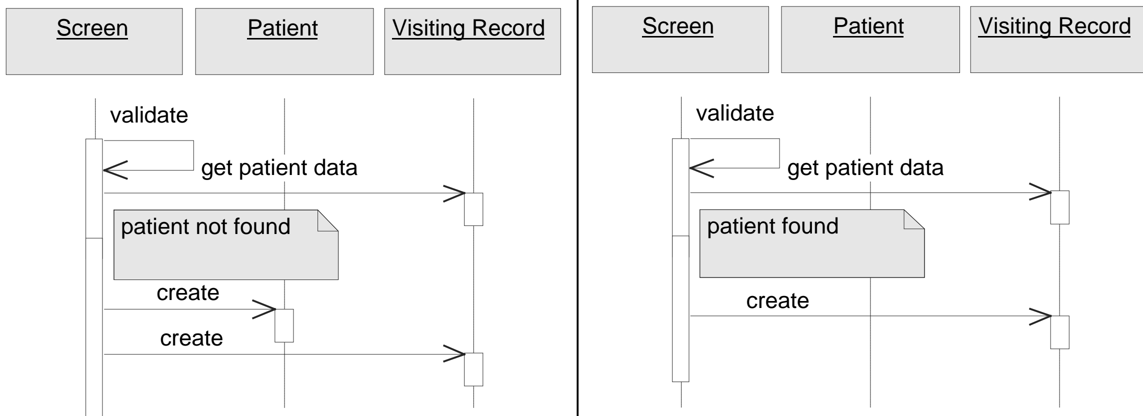

Figure 25: Another Scenario for SCED Mismatch Identification Approach

Koskimies et al [11] wrote that "the problem of synthesizing a state diagram on the basis of a set of scenario diagrams resembles the problem of learning a program from its sample traces." This implies that a single scenario diagram (as depicted in Figure 12) would not represent a general 'scenario' (except maybe in the most trivial of cases). Thus, to make this problem more interesting (and useful) we need more scenarios to be able to validate our state diagram. For that purpose, Figure 25 introduces an additional scenario (right side).

State diagrams for all involved classes may now be created by feeding above scenarios into a tool like SCED. The core of that tool consists of a state diagram synthesizer which can automatically create a state diagram reflecting the information of the individual scenarios. In case of the hospital scenarios above, the resulting state diagram may be seen in Figure 26 (upper half). The lower half shows the original state diagram from Figure 12. Like in the previous example, it is now possible to compare both views more easily because of the use of the same diagrammatic view type.

However, this example also shows that transforming information alone may not be sufficient in enabling automatic comparison. It would be hard for a computer to automatically identify mismatches simply because it is still unclear whether the boxes and arrows mean the same thing. For instance, both diagrams have four states. Thus, an automated tool could make the wrong assumption that *user input* corresponds to *do:validate*, *validate input* to *do:get patient data* and so forth. Thus, it may be necessary to manually adjust the mapping between views using the knowledge of the architects.

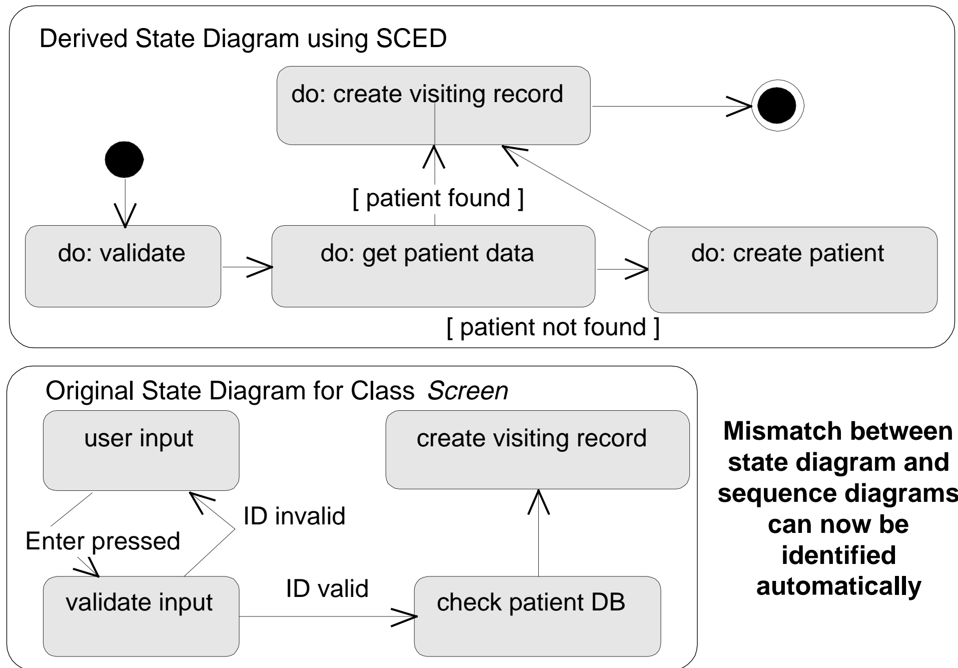

Figure 26: Sequence/State Diagram Mismatch more obvious though common base model

Nevertheless, even without manual adjustment, automatic comparison of these views would still identify potential mismatches. For instance, the original state diagram exhibits a feedback loop between the first two states that does not exist in the second one. Similarly, the derived state diagram has a shortcut from the second state to the last (forth) state which cannot be seen in the original one. Thus, the fact that these patterns in one view cannot be seen in the other view may already indicate potential mismatches. On the other hand, if both state diagrams would exhibit the feedback loop then this pattern could be used to guess which model elements correspond to each other. Finally, there may be even other diagrams (not seen here) which, after being transformed, may shed additional light onto this situation.

7.4 Trace Observation

The next technique shows a reengineering approach to view integration. This approach combines the model with the actual implementation and execution of the software product to identify dependencies between them. The identified dependencies are useful for many things but foremost for the verification of the conceptual integrity of the software model which is achieved by cross-referencing the product

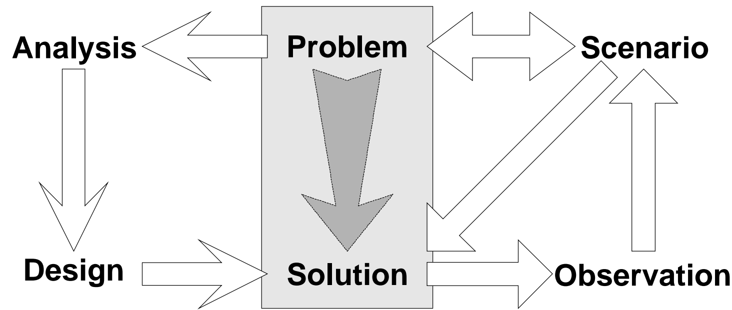

Figure 27: Combining Top-Down and Bottom-Up

implementation with its corresponding system/software model. As such the most significant byproduct are traces (mapping) between the model elements and the source code of the software product (the solution).

The left side of Figure 27 shows the typical life-cycle view of software development. Once a solution (implementation) has been created, using the top-down life-cycle approach, the behavior of the solution can be observed using the test scenarios which were created in the process. The captured observations in turn, which correspond directly the test scenarios, can be mapped back to the problem (requirements, architectures). Test scenarios are preferably created during the problem description stage and are refined in the analysis and design stage.

This techniques only works once an executable product, prototype, or simulation is available so that scenarios may be tested against it and the resulting internal activities of the product (or prototype) can be observed. Since observations correspond directly to scenarios and scenarios in turn corresponds directly to problem descriptions (requirements, architecture) a trace from implementation to requirements or architecture is established. It is our belief that this alternative process makes the tracing less difficult and it even fits nicely into existing testing activities. What remains is establishing and maintaining scenarios.

Figure 28 shows a simple example. It shows some high-level artifacts such as a class diagram and associated state and sequence diagrams (e.g possibly using SCED to derive the one from the other). Since the sequence diagram models the behavior of a system (component) it can be used as an input to test the system. There are various tools which perform static and dynamic analysis based on the source code and the execution and the output of these tools then reveal what happened during the execution. Thus, using sequence diagrams we can established a relationship (mapping) between the scenario and the physical low-level representation of the system. Since we also know what high-level components the scenario

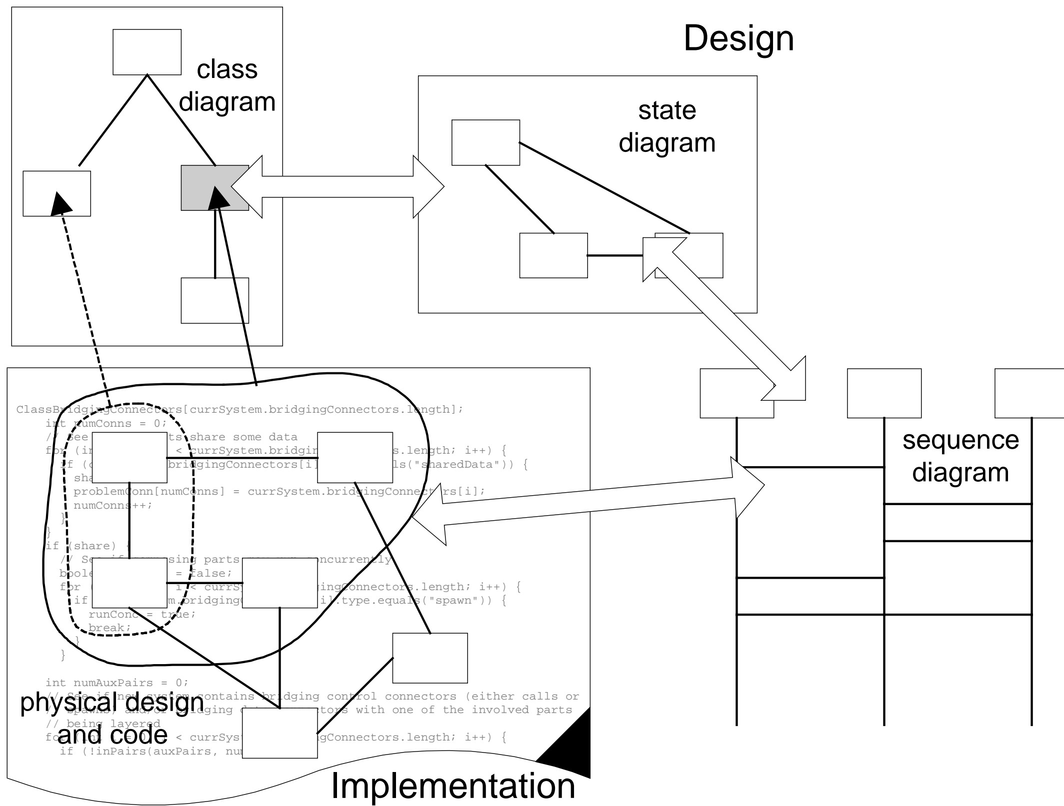

Figure 28: Using TraceObserver for Mapping and Transformation

corresponds to, we also get the relationship between high-level components and the implementation. In the figure this is indicated through fat arrows. Indicated also through the dotted circle is the result of another test execution of another class.

Figure 29 expands on that idea and shows two high-level diagrams and their corresponding scenarios. Through trace observation we again get the relationships to their lower-level components. Using both execution results we can now also make additional assumptions about the relationship of the high-level components. For instance, the observation of the second execution (c1 on the right hand side) showed that the same classes and additional ones were used as in the first execution (x1). Thus, for the high-level design this could imply an aggregation relationship between the involved components C and X. This type of information can then be used to validate components of the higher-level diagrams. Thus, the trace observation approach serves both mapping and transformation:

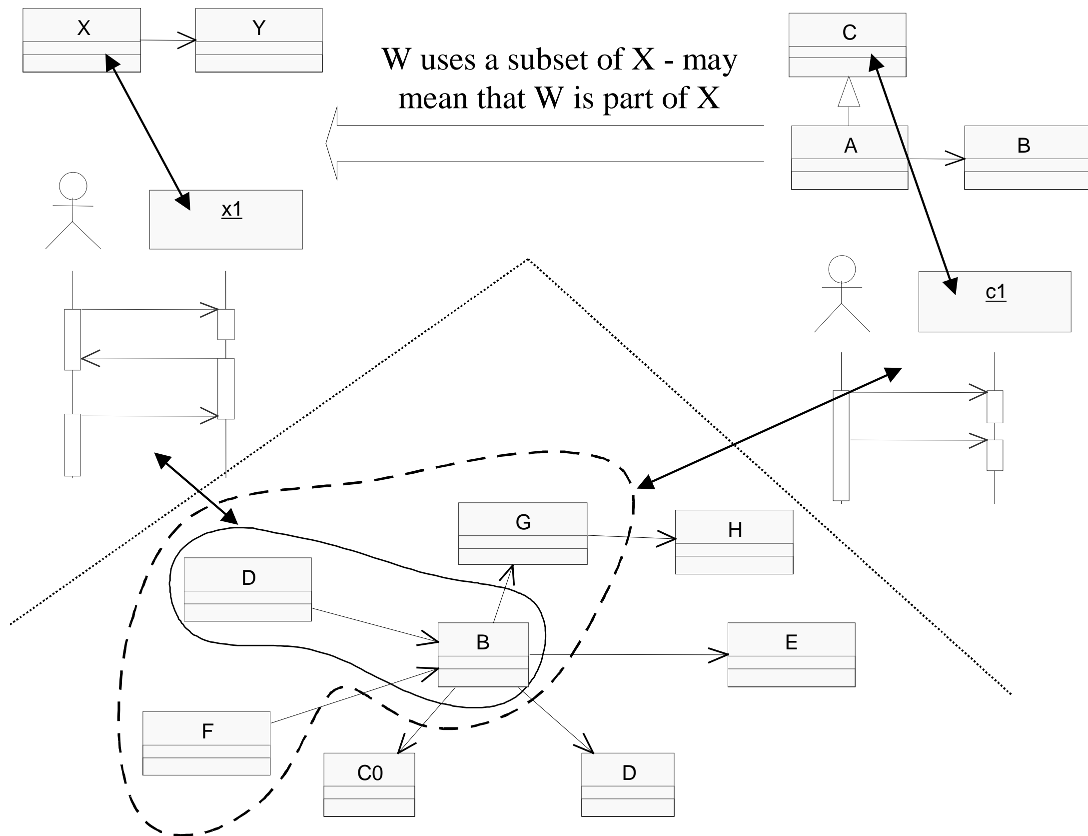

Figure 29: Mapping between high-level Components

- Mapping between low-level and high-level components
- Mapping between various high-level components
- Transformation of behavior diagrams to structural diagrams (the sequence diagram is mapped to a physical class diagram which in turn could be refined using Rose/Architect)
- Transformation of low-level structural/behavioral diagrams to corresponding high-level structural/behavioral diagrams.

7.5 Applying Integration Techniques to Activities

Above techniques showed ways on how to address architectural mismatches between and within views. As we discussed in the beginning of this section, we distinguish between three major dimensions of views – the vertical dimension (different view layers), the horizontal dimension (different view types) and the process dimension (different view configurations). Above examples showed one of each. Rose/Architect is an example for a vertical integration technique, SCED is one for horizontal integration, and TraceObserver is a process integration technique (although it also falls into the other categories). In a real development environment it is very likely that a combination of those techniques (and others) is necessary to yield effective results. For example, a state diagram may be derived using SCED but it must still be abstracted (simplified) in order to be comparable to the original state diagram (e.g. in our hospital example, the *create visiting record* state in the original view could be broken down to *do:create visiting record* and *do:create patient*). Table 2 summarizes the integration techniques in their usefulness in addressing the integration activities discussed before. It basically shows that the issues of mapping and

Table 2: Usefulness of Integration Techniques for Integration Activities

Integration Activities / Integration Techniques	Mapping	Transformation	Differentiation
Rose/Architect	+	++	-
Tracing through Execution	++	+	-
Collaboration to State Diagram	-	++	-

transformation are addressed but not the issue of mismatch identification itself which has to be done through other means. We will discuss *Differentiation* in more detail shortly.

7.6 Shortfalls of the VI techniques

The hospital system example (Figure 12) also showed that the assumptions of a common name base are not only unrealistic but also impossible between some types of views. For instance, as it was mentioned before, the boxes and arrows in state diagrams are not semantically the same as the ones in sequences diagrams. Thus, it is clear that the names derived through transformation will most likely not

match any other name. This is also true for the Rose/Architect model when it comes to the naming of transitive relationships. This is the reason why we talked about the need for *Mapping* in view integration.

Further, SCED, the sequence diagram to state diagram conversion is automatic one direction but not in the other. Possible ways of addressing that are:

1. Base model can store additional information about the origin of the state diagram viewpoint (=> the state diagram which was derived from sequence diagrams) so that the missing information is available when a backward comparison is applied. It is the view's responsibility to ignore that additional information so that the state diagram view still looks the same.
2. Verify which other objects offer the same/similar operators (if they are know). E.g. in the case of our patron and visiting record example, if the state says checking patron DB than it is clear that the patron object is referred to. Similarly the name of connectors may be used too.

This implies that the integration techniques are addressing some aspects of view integration but not others. This is not surprising since they were not created for this use. To compensate for their deficiencies we again have to look to the base model which we will discuss next. It is save to say that no individual technique will probably be sufficient in the long run – just like the software crisis where the cumulative set of best practices are the answer, techniques for integration need to be applied together.

7.7 System Model (Base Model; Repository)

The system model has been mentioned many times in this work but so far we have not elaborated on it. This section will therefore focus on this very important aspect of view integration. The following summarizes the needs and goals of the system model. We need it in order to:

- Avoid Redundancy (view independent representation)
- Reduce View Integration Complexity/Scalability
- Provide a common data foundation for *Differentiation* (rules and constraints)

- Hide additional information about modeling elements not visible in views but that are needed for mapping and transformation

Since UML already has a well-defined model foundation (described in UML itself) it seems natural to use it for our purposes. However, the UML model is only adequate in describing the syntactic structure of its views. It was not tailored towards two key criteria of above list – that of view redundancy and information hiding. This does not necessary invalidate the UML models for view integration but view integration would loose some of its strength if it is used as is. It is therefore the goal of this work to create a model which is compatible to the original UML model but which does not exhibit the deficiencies of the former.

On the other hand, it is also unlikely that the model presented in this work will satisfy all our needs, not to speak about all view integration needs. Thus, with respect to the base model we aim to find something which satisfies most or needs as well as being open enough for other techniques to be integrated. The following describes the proposed model.

7.7.1 Missing Information in UML Models

The UML model captures all relevant view information for all views it incorporates, however, not many of the inter-view relationships between them. Consider for example Figure 30. This figure depicts a three-class-diagram setting. Although, UML is able to describe each diagram adequately, it fails to capture any of the relationships between them. The rounded boxes and dashed lines are examples of such missing information. Only when we know these kinds of inter-view relationships, we are better able to understand the bigger picture outlined here.

Figure 30 also shows the layer extension which categorizes modeling elements (or even whole diagrams) into layers/partitions/etc. This figure further shows the refinement extension where components and connectors are associated with higher-level components and connectors. Not all this information must be entered manually. The mapping techniques mentioned before can be used to populate the system model with this kind of information. For example, we see that C0 – C3 are a refinement of C and we also see that both C (the higher-level component) as well as C1 (a part of the lower-level structure) have a dependency

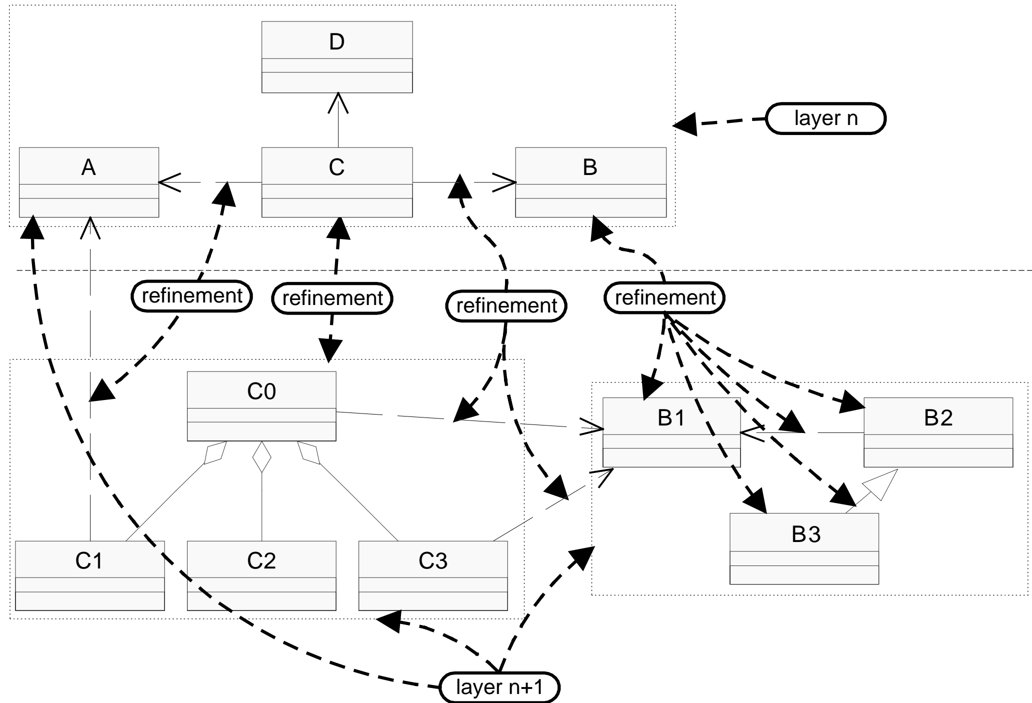

Figure 30: Annotated UML model with VIR extensions

relationship to A. Thus, the fact that the one connector is a refinement of the other could be concluded through some automated ways. Nevertheless, it is very unlikely that automated techniques could derive this sort of conclusion without at least some basic inter-view information available.

7.7.2 Avoiding Information Duplication

Figure 31 shows another example of an annotated UML diagram. It shows two class diagrams and uses the aggregate connector to show refinements between the two class diagrams (upper-left and middle). The upper-right corner shows a layering diagram (those diagrams are not directly supported by UML, however, we can use UML's extension mechanisms to represent it) that categorizes classes into layers. Finally, the lower-half of the figure shows a sequence diagram, depicting some of the interactions between these classes. In the current state, above model contains extensive redundancy. Consider for example class C which is contained in all four diagrams. Each diagram adds another piece of information – the first defines it, the second refines it, the third categorizes it, and finally the forth instantiated it. Although, the UML model would be able to deal with this type of redundancy to some limited degree, it is not sufficient for our needs.

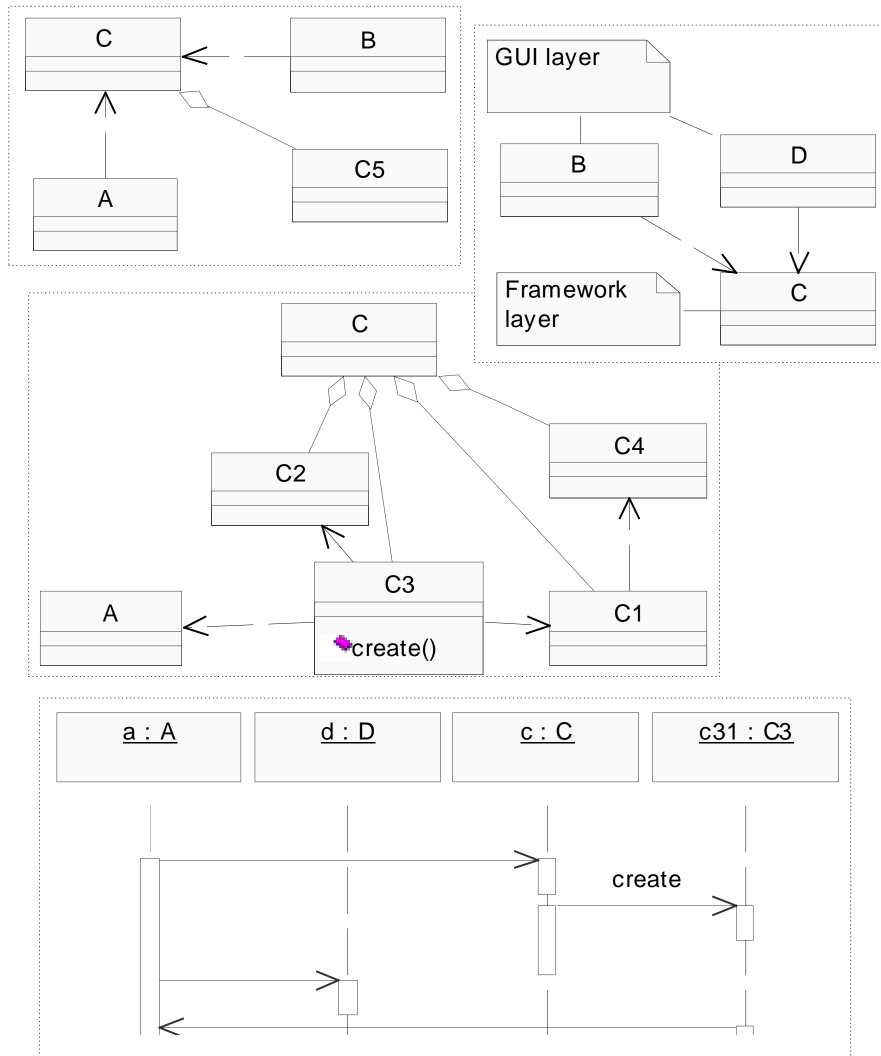

Figure 31: Class Diagram and Sequence Diagram in UML

Thus, another way of representing above figure would be in form of a base model. One such base model was described by Grundy [Grundy et al 1996]. However, he did not create a base model to capture different views but his model was limited to class diagrams and source code only. Nevertheless, this type of model seems highly beneficial since it breaks down the model into all its atomic pieces of information.

Figure 32 shows an example of a base model for above UML diagrams. Components in this base model are linked to other model elements in such a way that if a component is changed other components affected by that change can be notified. Note that the connectors in UML diagrams are components in the base model, making them first class citizens. In case of a modification to a component, a message is sent to

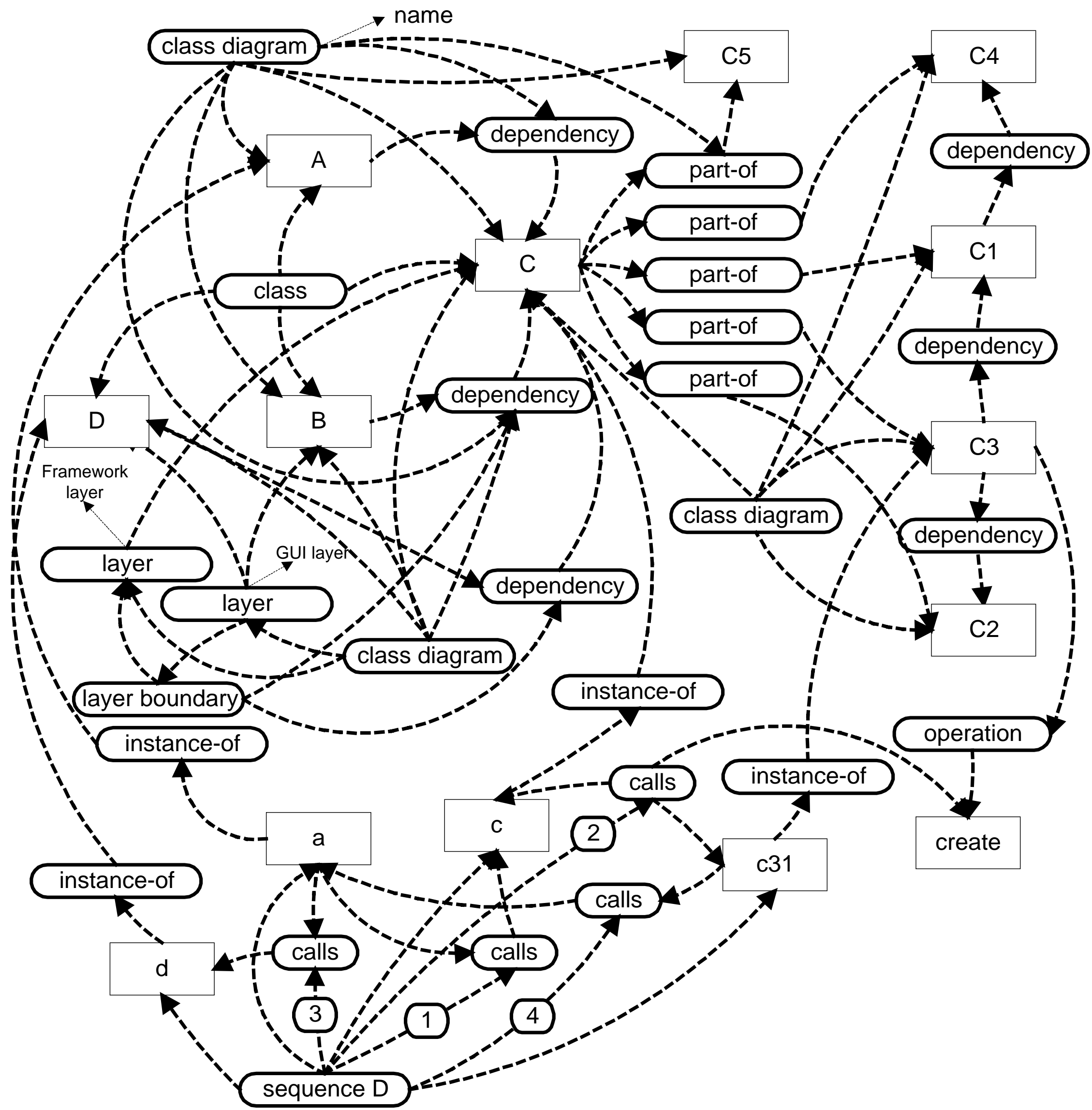

Figure 32: Corresponding Base model

the others (neighbors) and if the other element changes it will likewise propagate a change request to all its neighbors.

The rectangular boxes in Figure 32 correspond to the boxes in Figure 31 (this is done for simplistic reason, the next section will give another more atomic base model example) and the rounded boxes show interrelationships between them. This base model exhibits the following features:

- Relationships of components and connectors to diagrams (e.g. class diagram box) are shown in the same style as other relationships (e.g. dependencies between components)
- Components and connectors are interrelated through multiple dimensions. Consider for example class C again which is connected to 14 (!) other modeling elements in the base model.
- No component or connector that means the same is repeatably stored in the base model. Thus, redundancy is minimized.
- Related components are often directly associated with each other. For instance, interactions in the sequence diagram can readily be verified with interactions in the class diagram since their components are directly connected.

7.7.3 Example of Mismatch Identification in the Base Model

Figure 33 shows another example of UML views (a class diagram on the right and a sequence diagram on the left) and its corresponding base model. Instead of representing classes in boxes as we did in the last example, this example of a base model refines those into more atomic structures. Again, modeling elements are represented with as little redundancy as possible. This figure shows also an example of an architectural mismatch between the class diagram and the sequence diagram. The sequence diagram calls both the patient and the visiting record with a ‘create’ operator (which is supported by the class diagram), however, only the visiting record component has one such operator defined – not so the patient component. Since components and connectors are directly interrelated in the base model, this form of mismatch can be easily identified there.

7.7.4 Base Model as a View Independent Representation

The base model extends a view in that it also provides all information about modeling elements that were stripped away by a view. This is like architecting a building where a 2D view of a building does not reveal the depth of that view. The base model may be seen as being a 3D model of the building and views are 2D interpretations of it. Changes in the 2D views must not affect other views but likely will.

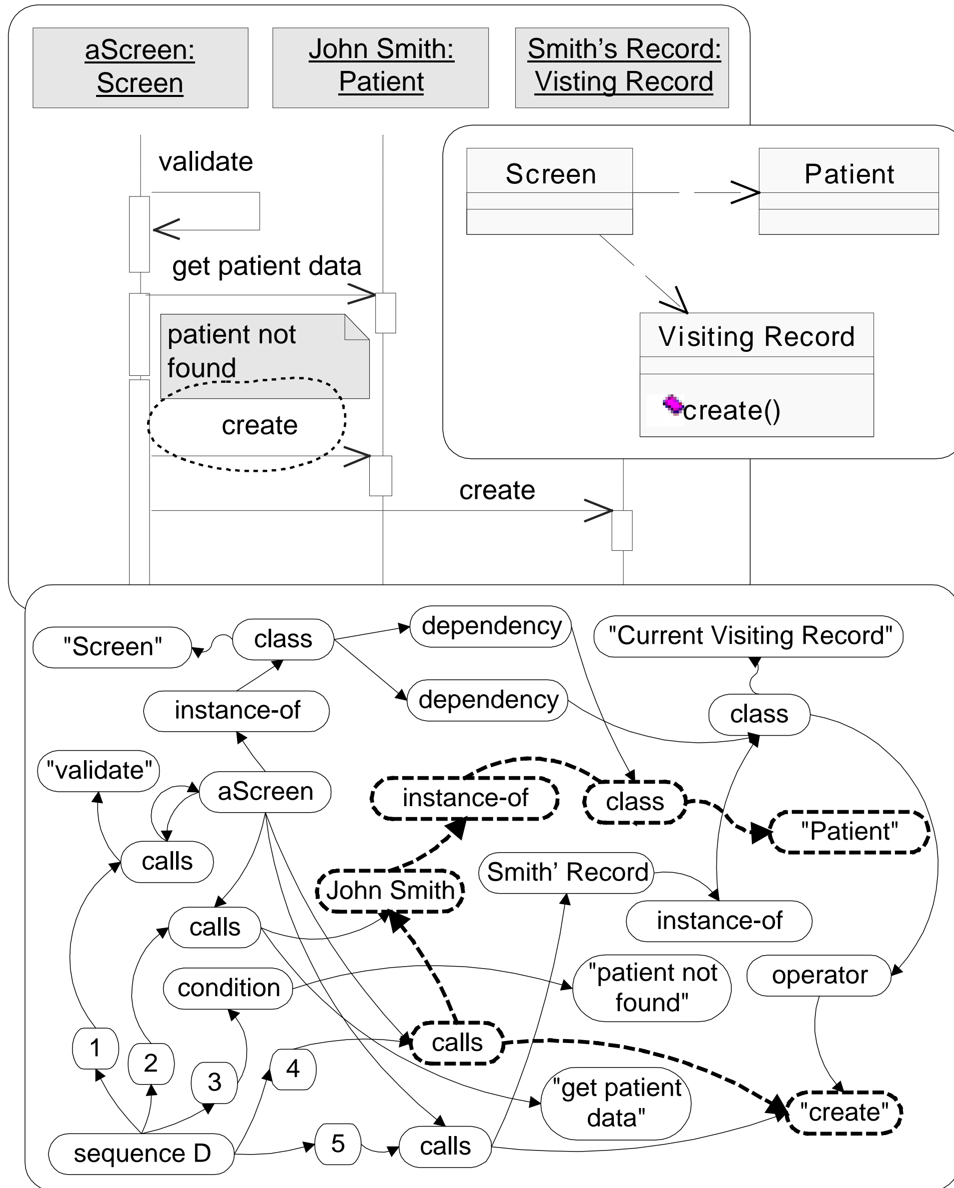

Figure 33: Views, Base Model, and a (potential) Mismatch

Thus, changes in the views must be propagated to the 3D model to ensure consistency. The base model is therefore a meta model of all its views. Thus, we have to:

1. Transform views onto the base model. All views that can be mapped onto this model may be compared there.
2. Transform views into other views if they cannot be represented in the same base model.

3. Transform derived/abstracted views (viewpoints) onto the base model.

In the latter two cases we also have to expand the base model in such a way that information which might be lost through transformation are still captured. Basically we need to store all information to be able to reverse the conversion and again yield the same (semantically identical) original model.

7.7.5 Real Views and Derived Views

The base model must, however, not only deal with view information and their known interrelationships but also with all sorts of derived and abstracted information. For instance, if a technique such as SCED or Rose/Architect is used to identify mismatches then the base model should also capture their results. Here, special care must be taken as to not to confuse real architectural information (entered by architects) with hypothesized information (automatically generated) since the latter is less faithful. Nevertheless, there are good reasons why derived information should be captured in the base model.

- Similar information may be derived sometimes in the future
- Manual assistance given while deriving view can be considered 'real' information, however, those might only be useful in the context of the derived view.
- They further constrain the existing base model and thus they are needed for *Differentiation* (which is based on the base model).

7.7.6 Other issues

- Undo/redo changes
- Renaming/changing information and how to propagate that
- Algorithms for mapping
- Algorithms for transformations
- Algorithms for differentiation
- Life-time and Extend of mismatches or related mismatches

7.8 Extending the Notation and Semantics

Although we can create the base model any way we wish, we nevertheless are bound by what UML views (or others) are capable off. It is not the intent that people use the base model and add information to it, although, this options should be possible since some more experienced people might find it more powerful. Thus, information in the base model must somehow be entered through views. This implies that this work also needs to extend UML views to accommodate the needs of the base model. Fortunately, UML incorporates extension mechanisms (e.g. stereotypes) and we will make use of that. However, since a UML component may only have one stereotype attached, we will have to investigate other extension mechanisms as well. Issues are:

- How to represent Rules and Constraints?
- Can the UML model be integrated with our base model (OO base model)?
- How to deal with non-UML and non-architectural specifications? E.g. more detailed specification from code?

7.8.1 Describing Rules and Constraints

Since the system model must reflect the synthesis and the analysis of a product, it has to be able to handle structural information as well as semantical ones. The former are already captured in some detail in UML, however, the latter need to be described somehow else. For that we use rules and constraints. We distinguish two types of constraints in a system model:

1) Constraints opposed by a view or style used (e.g. calling hierarchy of layers) and
2) Constraints opposed by a domain (e.g. no two current visiting record shall exist for one patient)

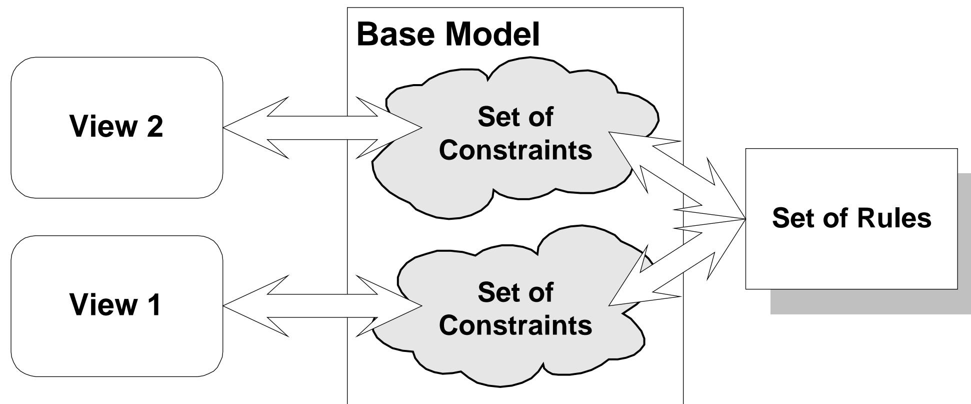

Figure 34: Rules and Constraints

Rules describe the legal or illegal aspects of views and the system then compares these rules with the constraints opposed by the views and domains in order to identify mismatches (see also Figure 34). The base model we discussed previously can also be described and interpreted as constraints. For instance, the fact that *John* is an instance of *Patient* constraints *John* to the structure and behavior of *Patient*. The mismatch example in Figure 33 violated the rule that definition and instance must use the same operators and attributes. In order to describe rules and constraints, we are currently considering the use of either OCL (Object Constraint Language) [Booch-Jacobson-Rumbaugh 1997] or Z [Bowen-Hinchey, 1995]. Since the former is supported by UML it is a more likely choice.

8 Mismatch Resolution

Good order is the foundation of all good things – Edmund Burke

Previously we focused our attention on how to find mismatches – but once found, how can we deal with incomplete/inconsistent information? Can approximation concepts be used to fill the gap? In case of the mismatch examples in Figure 33 the violated rule can be resolved by either implementing all its constraints or eliminating them. Thus, the two choices we have are 1) create an operator named *create* for *Patient* or 2) delete the call from *Screen* to *Patient*. Since these options are direct derivations of the rules and constraints, they could be derived automatically and presented to an architect to make the final decision.

We don't belief that the actual mismatch resolution should be done fully automatic. This issues is related to a previous endeavor of self-correcting source codes for compilers which ultimately failed because of the social and technical complexities involved. However, we belief that presenting the architect not only with (potential) mismatches but also with ways on how to resolve them is highly beneficial in both dealing with mismatches and in understanding them. Furthermore, it might be very beneficial to also devise techniques that deal with the issue of which options are better. To come back to our example of Figure 33 and our two choices discussed above, are both options equally good? We can investigate this by actually executing those options.

1) Remove the *create* call to *Patient* from the sequence diagram: A side effect of that would be that although our immediate mismatch rule is resolved, we find that now we do not have an example any more which shows the dependency of *Screen* to *Patient* in the class diagram. Or another way of saying this is, we had two arguments in favor of a call from *Screen* to *Patient*: the dependency connector in the class diagram and the call connector in the sequence diagram; but only one argument against it.

2) Add a *create* operator to *Patient* in Class Diagram: This option also resolves our immediate mismatch rule and no further side effects are created – meaning no other mismatch rules were violated.

Thus, it seems that option number 2 is the better one and should be presented to the architect with a higher confidence. Mismatch resolution is not the central focus of our work here. Nevertheless, we think that our framework is also very suitable for incorporating it.

10 Integrating this Work

When we build, let us think that we build forever – John Ruskin

In Chapter 4 we introduced related work of other researchers. There we claimed that integration is part of every aspect of the development life-cycle and, thus, our work and the related work presented in that chapter, fit somehow into the greater scheme of the view integration problem. This section will elaborate on that.

10.1 MBASE

In order to determine whether a software/system architecture is satisfactory (meets stakeholders' expectations) one needs considerably more than a specification of components, connectors, configurations and constraints. Considering the architecture as an island, as we did to some degree in this work, puts one at a serious disadvantage in evaluating its adequacy.

[Boehm, et al., 1999] has been developing, applying and refining an approach called MBASE (Model-Based Architecting and Software Engineering) [Boehm-Port, 1998] to address this issue. It focuses on ensuring that a project's product models (architecture, requirements, code, etc.), process models (tasks, activities, milestones), property models (cost, schedule, performance, dependability), and success models (stakeholder win-win, IKIWISI (I'll Know It When I See It), business case) are consistent and mutually enforcing.

Figure 35 summarizes the overall framework used in the MBASE approach to ensure that a project's success, product, process and property models are consistent and well integrated. At the top of Figure 35 are various success models, whose priorities and consistency should be considered first. Thus, if the overriding top-priority success model is to "Demonstrate a competitive agent-based data mining system on the floor of COMDEX in 9 months," this constrains the ambition level of other success models (provably correct code, fully documented as a maintainer win condition). It also determines many aspects of the product model (architected to easily shed lower-priority features if necessary to meet schedule), the

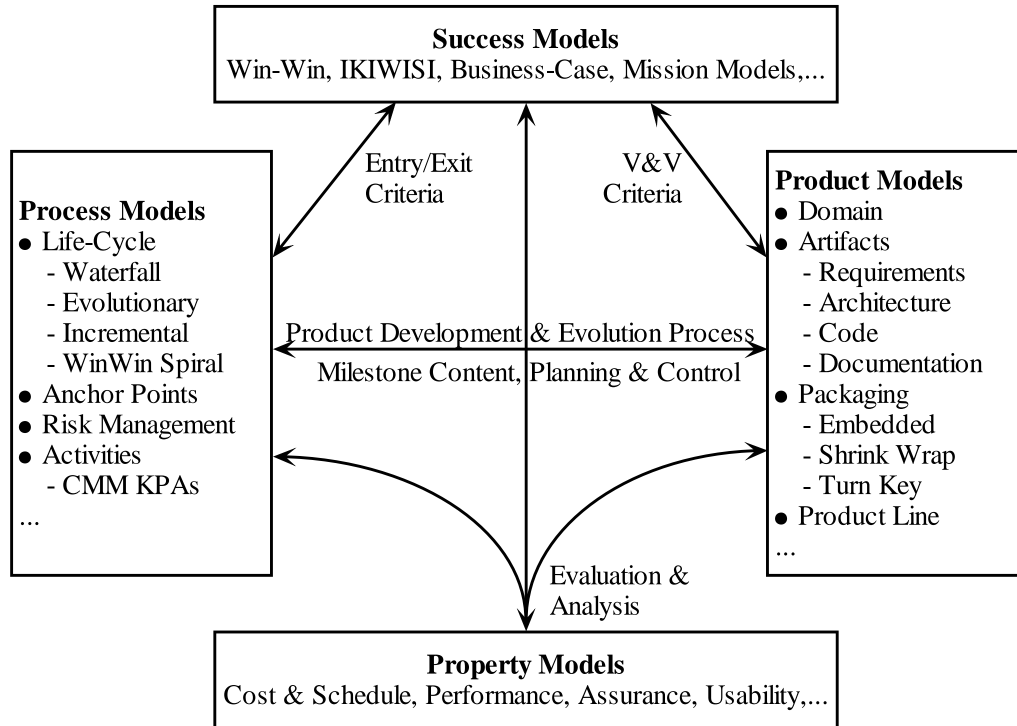

Figure 35: MBASE Overview

process model (design-to-schedule), and various property models (only portable and reliable enough to achieve a successful demonstration). In the context of our work, we are primarily concentrating of ensuring the integrity of the product model. However, for our work to be truly useful, it also needs to be integrated with the other types of models.

10.2 Architectural Mismatches During System Composition

Furthermore, our work did not address a wide range of architectural views but instead focused on UML views only. As such we did not investigate architectural description languages (ADL). As mentioned in Chapter 4, there are numerous groups working and ADLs and their integration (e.g. [Garlan et al 1997], [Robbins et al 1998], and [Gacek 1998] to list just a few).

Our view integration effort fits into theirs in a number of ways. For one, UML views are considered to be more general-purpose views and thus ADL views may be used to analyze some aspects in more detail. Other ADL integration efforts are more high-level then ours and, thus, they may be used in advance to identify possible mismatches earlier.

For instance, the work of [Gacek 1998] fits into the latter category. Figure 36 is an example that shows components of systems that are described through some properties and styles. The information management system described here consists of five major components. Two of those components, *Existing Information System* and *Oracle*, are part of the current management system. New capabilities are added through *My Application,* which uses another database. On top of that, a component called *My Sync* is in charge of ensuring consistency between both databases.

The components *Existing Information System*, *Oracle*, and *Object-Oriented Database* are COTS (commercial off the shelf) products and their internal structure is largely unknown – although the general type of style and some architectural properties are known. The remaining components, *My Application* and *My Sync* on the other hand are very well know since those are applications to be developed in-house. Knowing some of the properties of systems and their components allows us now to reason about potential mismatches that could occur while composing them. This is basically what the work of [Gacek 1998] and

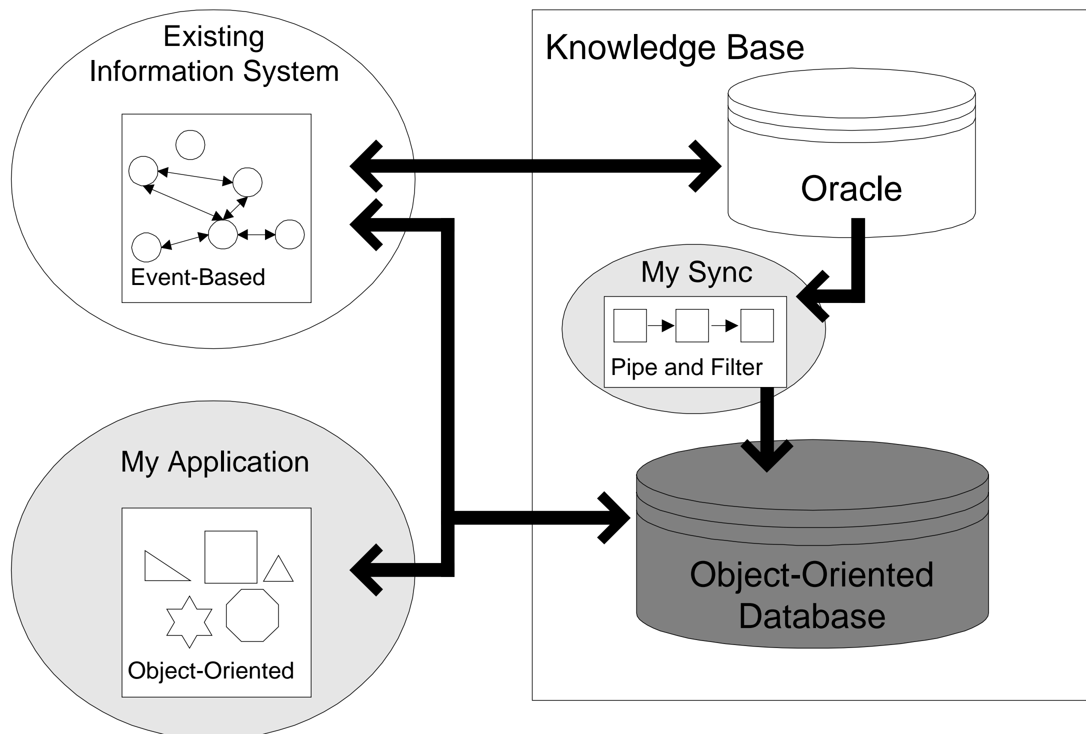

Figure 36: System Composition Example

[Abd-Allah 1996] characterizes. With their tool, named AAA (for Architect's Automated Assistant), we can now describe our proposed system in terms of known properties and the tool then infers potential mismatches based on those properties. Thus, their work represents and analyzes architecture although their components remain black boxes.

On the other hand, our work can be used in a more detailed level to investigate mismatches within one such box. For instance, in above example, the component *My Application* is an object-oriented component where UML is used to describe the details. Therefore, our work may be used to describe mismatches **within** the *My Application* box, whereas their work can be used to describe potential mismatches when we combine *My Application* with other (COTS) components.

10.3 ADL Views as Mini-Spirals

Another way of integrating ADLs with UML (or the software development process at large) is to substitute the general-purpose language whenever appropriate. This was indicated above when we said that UML views are considered to be more general-purpose views and, thus, ADL views may be used to analyze some aspect in more detail. [Medvidovic 1998] provides an interesting analysis of that aspect (see

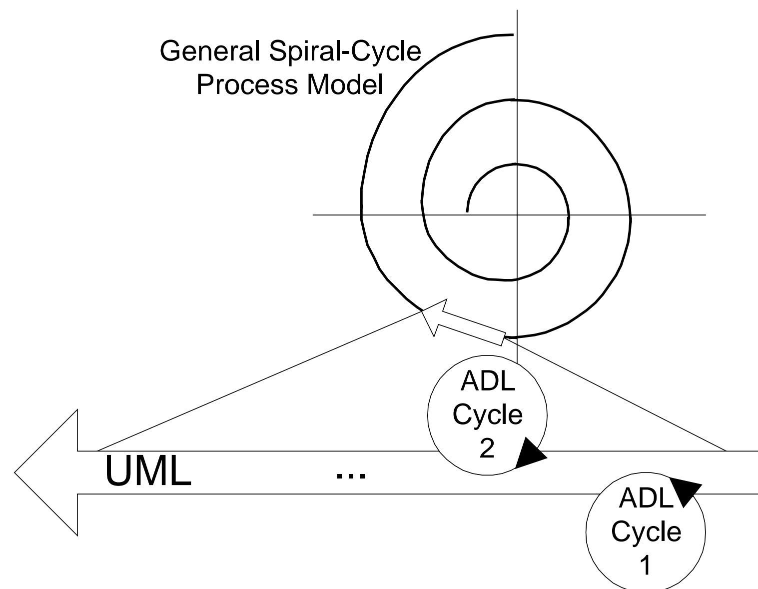

Figure 37: General-Purpose Models and ADLs

Figure 37). This figure depicts the use of ADLs (and other more formal design methods) as being mini-spirals within the existing process model (e.g. another spiral model in this case). View integration in this type of example has to ensure the conceptual integrity of the pre- and post conditions of those mini-spirals.

It is out of the scope of this work to do a more comprehensive analysis on how other integration approaches fit together. Nevertheless it would be interesting to know how fundamentals of our approach would work with other integration ideas – for instance, how well does our base model (system model) work with ADLs which use conceptually different types of components and connectors.

11 Future Work

Even when the laws have been written down, they ought not always to remain unaltered - Aristotle

Issues concerning future work were already addressed throughout this work. This section will summarize the key aspects again. We basically distinguish two types of future work: 1) what we hope to accomplish as part of our thesis work and 2) what we belief this general area should accomplish.

Future Work for this Thesis:

- Define the structure and constraints of the base model
- Define mismatches in form of consistency and completeness rules
- Describe extensions to the UML model
- Identify and Describe *Mapping*, *Transformation*, and *Differentiation* techniques
- Provide more comprehensive mismatch examples using several integration techniques
- Define mismatch resolution options and their accuracy

Future Work for View Integration in general:

- Find a more Universal Model – a true View Independent Representation
- Automating the mismatch identification and resolution process
- Reduce the state explosion problem
- Process, Property, and Success model integration (MBASE)
- ADL integration

12 Conclusion

The secret of success is constancy to purpose – Benjamin Disraeli.

Views are nothing more than an abstraction of relevant information from its model and, thus, views are necessary to present information in some meaningful way to stakeholders (developer, architect, customer, etc.) – something the system model does not do. Because of views, "inconsistencies are inevitable in software development […] processes and products. They provide a focus for further development […], and can be regarded as 'desirable' in that they highlight issues that need further attention. As such, they should be tolerated, analyzed and acted upon." [Hunter-Nuseibeh 1997] Therefore, we talk about the need for view integration, and thus, we also talk about the need of having a system model integrated with its views. Although, a full integration effort may seem impossible at this point, it can still be attempted for significant parts.

With that purpose in mind, this work presented a framework and some techniques to help software developer in ensuring the conceptual integrity of their models, and thus ensuring the consistency of views. Major problems we have not yet fully addressed are:

- Finding (or developing) integration techniques covering a wide range of views
- Dealing with the state explosion problem in mapping and transforming views and as such with the issue of scalability
- Addressing the issue of automatically supported mismatch resolution (as compared to just automated mismatch identification)

Despite those problems, we feel that the potential benefits of using integration techniques, such as the ones above, are immense. We have shown that it is possible to (at least partially) automate the task of mismatch identification and since computers are clearly much more efficient in comparing views, this

implies that substantial manual labor can be saved (currently, consistency checking is probably still the largest development activity that is done almost entirely manually). Another benefit of our approach is that mismatches may be identified as early on as they are created. Every time new data is added to the model, tools (e.g. agents) can validate them. This in turn can save substantial rework cost later on.

13 References

Abd-Allah, A. (1996) "Composing Heterogeneous Software Architectures," Doctoral Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA.

Allen, R. and Garlan D. (1996) "*The Wright Architectural Specification Language*," 24 September (http://www.cs.cmu.edu/afs/cs/project/able/ftp/wright-tr.ps).

AT&T (1993) "Best Current Practices: Software Architecture Validation," AT&T, Murray Hill, NJ.

Cheng, B.H.C., Wang, E.Y., Bourdeau, R.H., Richter, H.A. (1995) "Bridging the Gap Between Informal and Formal Approaches to Software Development," Proceedings of Software Engineering Research Forum, November.

Chroust, G. (1992) "Modelle der Software Entwicklung," Oldenbourg Verlag

Coad, P. and Yourdon, E. (1991a) "Object-Oriented Analysis," Yourdon.

Coad, P. and Yourdon, E. (1991b) "Object-Oriented Design," Yourdon.

Belkhouche, B. and Lemus, C. (1996) "Multiple View Analysis and Design," Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development.

Boehm, B.W. (1981) "Software Engineering Economics," Prentice Hall.

Boehm, B.W. (1996) "Anchoring the Software Process," *IEEE Software*, July, pp.73-82.

Boehm, B., Egyed, A., Kwan, J., and Madachy, R. (1998), "Using the WinWin Spiral Model: A Case Study," *IEEE Computer*, July, pp. 33-44.

Boehm, B., Port, D. (1998) "Conceptual Modeling Challenges for Model-Based Architecting and Software Engineering (MBASE)," Proceedings of Conceptual Modeling Symphosium.

Booch, G. (1994) "Object-Oriented Analysis and Design with Applications," Second Edition, Addison-Wesley.

Booch, G., Jacobson, I., and Rumbaugh, J. (1997) "The Unified Modeling Language for Object-Oriented Development," Documentation set, version 1.0, Rational Software Corporation.

Brooks, F. P. (1995) "The Mythical Man-Month," Addison Wesley.

Bowen, J. P., Hinchey, M. G. editors (1995) "ZUM'95: The Z Formal Specification Notation," 9th International Conference of Z Users, Volume 967 of Lecture Notes in Computer Science.

Delugach, H.S. (1996) "An Approach to Conceptual Feedback in Multiple Viewed Software Requirements Modeling," Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development.

Ehrig, H., Heckel, R., Taentzer, G. and Engels, G. (1997) "A Combined Reference Model- and View-Based Approach to System Specification," International Journal of Engineering and Knowledge Engineering, Vol.7 No.4, pp. 457-477, World Scientific Publishing Company.

Ferguson, J., et al. (1996) "Software Acquisition Capability Maturity Model," Technical Report, CMU/SEI-96/TR-020, ESC-TR-96-020.

Finkelstein, A, Kramer, J., Nusibeh, B., Finkelstein, L., and Goedicke, M. (1991) "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal on Software Engineering and Knowledge Engineering*, March, pp. 31-58.

Gacek, C. (1998) "Detecting Architectural Mismatches During System Composition," Doctoral Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA.

Gacek, C., Abd-Allah, A., Clark, B.K., and Boehm, B. (1995) "On the Definition of Software System Architecture," in *Proceedings of the First International Workshop on Architectures for Software Systems - In Cooperation with the 17th International Conference on Software Engineering*, Seattle, WA, 24-25 April 1995, pp. 85-95.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) "Design Patterns – Elements of Reuseable Object-Oriented Software," Addison-Wesley.

Garlan, D., Monroe, R.T., Wile, D. (1997) "ACME: An Architecture Description Interchange Language," Proceedings of CASCON'97, November.

Grady, J. O. (1994) "Systems Integration," CRC Press, Boca Raton, FL.

Grundy, J.C., Hosking, J.G, Mugridge, W.B., Amor, R.W. (1996) "Support for Constructing Environments with Multiple Views," Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development.

Humphrey, W.S. (1995) "A Discipline for Software Engineering," Addison-Wesley, Reading, MA.

Hunter, A., Nuseibeh, B. (1998) "Managing Inconsistent Specifications: Reasoning, Analysis, and Action" TOSEM 7(4): 335-367 (1998)

IEEE (1998) "Recommended Practice for Architectural Description," Draft Std. P1471, IEEE.

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992) *Object-Oriented Software Engineering-A Use Case Driven Approach*, Addison-Wesley Publishing Company.

Koskimies, K., Systä, T., Tuomi, J., and Männistö, T. (1998) "Automated Support for Modelling OO Software," IEEE Software, January, pp. 87-94.

Kruchten, P. B. (1998) "The Common Misconceptions about Software Architecture," Proceedings of the 2nd Ground Systems Architecture, El Segundo, CA.

Kruchten, P. B. (1999) "The Rational Unified Process," Addison-Wesley.

Kuhn, D.A. (1996) “A Discription of the Systems Engineering Capability Maturity Model Appraisal Method Version 1.1,” CMU/SEI-96-HB-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

Magee, J. and Kramer J. (1996) “Dynamic Structure in Software Architectures,” in *Proceeding of the ACM SIGSOFT’96: Fourth Symposium on the Foun-dations of Software Engineering (FSE4)*, San Francisco, CA, October, pp. 24-32.

Oreizy, P., Medvidovic, N., and Taylor, R. (1998) “Architecture-Based Runtime Software Evolution,” Proceedings of the 20th International Conference on Software Engineering.

Övergaard, G. (1998) “A Formal Approach to Relationships in the Unified Modeling Language,” Proceedings PSMT’98 Workshop on Precise Semantics for Software Modeling Techniques.

Mitchell, H.J. (1998) “An NRO Vision for Ground Systems Architecture,” Proceedings of the 2nd Ground Systems Architecture Workshop, El Segundo, CA.

NASA (1993) “Software Formal Inspection Process Standard,” NASA-STD-2202-93.

Nuseibeh, B., Kramer, J., and Finkelstein, A. (1994) “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification,” IEEE Transactions on Software Engineering, pp. 760-773, October.

Nuseibeh, B. (1995) “Computer-Aided Inconsistency Management in Software Development,” Technical Report DoC 95/4, Department of Computing, Imperial College, London SW7 2BZ.

Nuseibeh, B. (1996) “Towards a framework for managing inconsistency between multiple views,” Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development.

OMG (1997) “Object Management Group Adopts Unified Modeling Language and Meta Object Facility Specifications,” Press Release of the OMG, November 17.

Paulk, M.C., Weber, C.V., Curtis, B., and Chrissis, M.B., Eds. (1995) “The Capability Maturity Model: Guidelines for Improving the Software Process,” Addison-Wesley, Reading, MA, 1995.

Rapide (1996) “Guide to the Rapide 1.0 Language Reference Manuals,” Stanford University, 5 December (http://anna.stanford.edu/rapide/lrms/overview.ps).

Rechtin, E. (1991) “System Architecting, Creating & Building Complex Systems,” Prentice Hall, Englewood Cliffs, NJ.

Perry, D. E. and Wolf, A. L. (1992) “Foundations for the Study of Software Architectures,” *ACM SIGSOFT Software Engineering Notes*, October.

Robbins, J.E., Medvidovic, N., Redmiles, D.F., Rosenblum, D.S. (1998) “Integrating Architecture Description Languages with a Standard Design Method”, Second EDCS Cross Cluster Meeting in Austin, TX, ftp://www.ics.uci.edu/pub/arch/papers/TR-ICS-UCI-97-35.pdf.

Royce, W. W. (1970) “Managing the development of large software systems: Concepts and techniques,” Proceedings of ICSE 9.

Rumbaugh, J., Blaha, M., Premerlani, W., and Eddy, F. (1991) "Object-Oriented Modeling and Design," Prentice Hall.

Sage, Andrew P., Lynch, Charles L. (1998) "Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives," Systems Engineering, The Journal of the International Council on Systems Engineering, Wiley Publishers, Volume 1, Number 3, pp.176-226.

Schönberger, S., Keller, R.K., and Khriss, I. (1998) "Algorithmic Support for Model Transformation in Object-Oriented Software Development," Technical Report GELO-82, Université de Montréal, Montreal, Quebec, Canada, February.

Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., and Zelesnik, G. (1995) "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, vol. 21. no. 4, April 1995, pp. 314-335.

Shaw, M. and Garlan, D. (1996) "Software Architecture: Perspectives on an Emerging Discipline," Prentice Hall.

Sheard, S.A., Lake, J.G. (1998) "Systems Engineering Standards and Models Compared", Proceedings of the Eighth International Symposium on Systems Engineering, Vancouver, Canada, pp. 589-605.

Siegfried, S. (1996) "Understanding Object-Oriented Software Engineering," IEEE Press.

Sommerville, I. (1996) "Software Engineering," Addison-Wesley.

Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D. (1996) "A Component- and Message-Based Architectural Style for GUI Software," IEEE Transactions on Software Engineering, pp. 390-406, June.

Wang, E.Y., Richter, H.A., and Cheng, B.H.C (1997) "Formalizing and Integrating the Dynamic Model within OMT," Proceedings of the IEEE International Conference on Software Engineering, May.

Wang, E.Y., Cheng, B.H.C. (1998) "A Rigorous Object-Oriented Design Process," Proceedings of the International Conference on Software Processes (ICSP5), June.

14 Appendix

14.1 Complete Set of Rules in Rose/Architect