Trace Observer: A Reengineering Approach to View Integration

Alexander Egyed
Center for Software Engineering
University of Southern California
Los Angeles, CA 90089-0781, USA
aegyed@sunset.usc.edu

Abstract

Developing software in phases (stages) using multiple views (e.g. diagram) is the major cause for inconsistencies between and within views. Views exhibit redundancies because they repeatably use the same modeling information for the sake of representing related information within different perspectives. As such, redundancy becomes a vital ingredient in handling complexity by allowing a complex problem (model) to be divided up into smaller comprehensive problems (closed-world assumption).

However, this type of approach comes with a price tag; redundant views must be kept consistent and, at times were more and more development methodologies are used, this task becomes very time consuming and costly. We have, therefore, investigated ways on how to automate the issue of identifying view mismatches and to this end we have created a view integration framework. This paper describes this framework and shows how scenario executions and their observations can help in automating parts of that framework. Trace Observer, as this technique is called, can assist in cross-referencing (mapping) high-level model elements and it may also be used for transforming model elements so that different types of views may interpret them.

Introduction

Integrating multiple views has been widely recognized as being an integral aspect of software development. “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.” [Siegfried 1996] Engineering in general, and software engineering in particular, have therefore made extensive use of views to separate stakeholder concerns into more comprehensive pieces. In doing so, a complex problem is split up into a collection of smaller, less difficult problems whose individual solutions can then be merged together to solve the original bigger problem.

The IEEE Draft Standard 1471 [IEEE 1998] refers to a view as something which “addresses one or more concerns of the system stakeholder.” With stakeholder we mean an individual or a group that 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* that is still small enough for us to comprehend but that also contains all relevant information about a particular concern. As an example, the diagrams (and instances of diagrams) offered in general purpose languages, such as the Unified Modeling Language (UML) [Booch-Jacobson-Rumbaugh 1997] represent types of views. Views address the following critical issues:

- Abstract and simplify the model
- Enable different stakeholders to work and to coordinate
- Compensate for different interpretations (different audiences/stakeholders)
- Extract relevant information about a particular concern.

What types of views should be used and when they should be used is, therefore, strongly dependent on the people who are using them and the tasks that are needed to be accomplished.

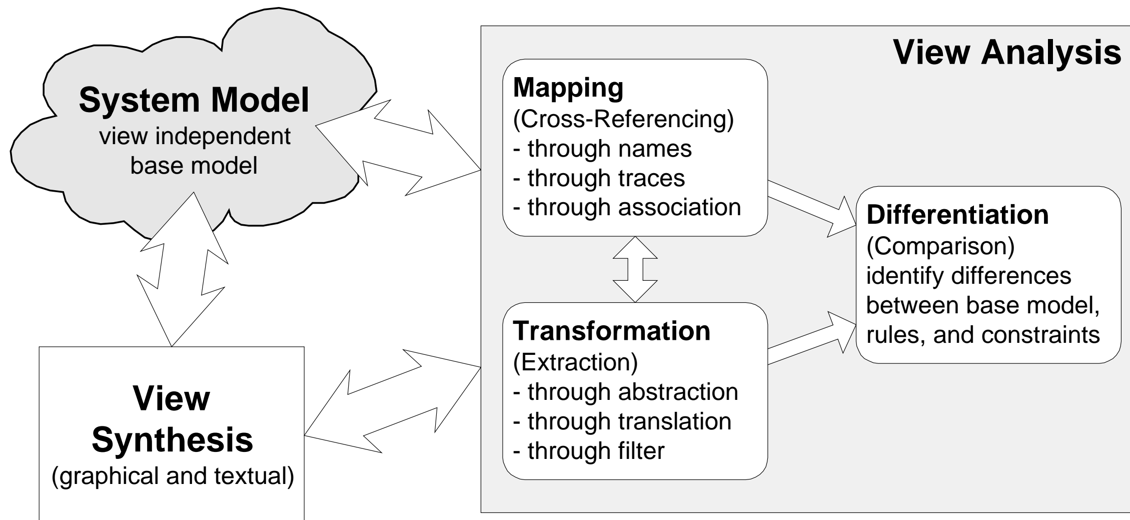

Figure 1. View Integration Framework

The View Integration Framework

Typical software development models in a project are compositions of numerous views (and smaller models) – the union of which capture the entire problem and solution. The task of developing software involves decomposing the model instances into its views and then, after having gone through some transitions, composing them again to yield a solution. This problem is very similar to that of decomposing and composing the actual software system. Just like software systems composition results in the need of resolving mismatches between components, we need similar techniques which do the same on a meta-level between models and views.

Software development is therefore about modeling the real problem adequately, solving the model problem, and interpreting the model solution in the real world. In doing so, a major emphasis is placed on mismatch identification and reconciliation within and among views (such as diagrams). We often find that this latter aspect, the analysis and interpretation of (architectural/design) 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.

We have therefore investigated ways on how to describes and identify causes of architectural/design mismatches between views. To address that problem, we make use of the very problem of view inconsistencies – that of redundancy between. Redundancies between a set of views implies that one view contains information about another view. That information, once identified and transformed can be then be used to constrain both views. Since there is more to view integration than constraints and consistency rules, we created a view integration framework (see Figure 1) where we can apply those rules in a meaningful way. Therefore, view integration defines *what* information can be exchanged and *how* it can be exchanged. Only after the *what* and *how* have been established, can inconsistencies be identified and resolved.

The system model (e.g. UML model) in Figure 1 represents the repository of the designed software system. Software developers use views to add new data to the system model or to review existing data (view synthesis). Interacting with both, the system model and the view synthesis, is the view analysis activity. As soon as new information is added, it can be validated against the system model to ensure its conceptual integrity. View Analysis involves the following major activities:

- **Mapping:** Identifies related pieces of information and thereby describes *what* information is overlapping. Mapping is often done manually through naming dictionaries or traceability matrices. We can automate mapping through the use of patterns, interfaces, trace observations, and others.

- **Transformation:** Extracts and manipulates model elements of views in such a manner that they (or pieces of them) can be interpreted and used by other views (*how* can information be exchanged). Transformation can be categorized into basically two dimensions: abstraction (simplify or generalize detailed views) and translation (convert information for use in different types of views).
- **Differentiation:** Traverses the model to identify (potential) mismatches within its elements. (Potential) mismatches can automatically be identified through either the use of rules and constraints or direct (graph) comparison. Mismatch identification rules can frequently be complemented by mismatch resolution rules. Automated differentiation is strongly dependent on *Transformation* and *Mapping*.

The Challenge of Mapping (Traceability)

To date we have identified numerous techniques for automating the activities of *Transformation* and *Differentiation* [Egyed 1999a] and [Egyed 1999b]. However, automating the *Mapping* activity of our framework has been a particular challenge. In previous papers we have usually deferred that activity as being mostly manual since *Mapping* has a lot to do with the traceability problem. We need *Mapping* because we need to know what information is repeatably used by multiple views. Thus, *Mapping* is critical when it comes to the scalability of our automated view integration approach in that it constrains what information is needed to be transformed where, and what information is needed to be compared (differentiated) with what. The following approaches can be used to automate *Mapping*:

- **Same or Similar Names:** Names of model elements may sometimes be used to infer relationship. Even similarity in names may give helpful hints (see examples in [Egyed 1999b]).
- **Design Patterns/Features:** Knowledge about patterns (both in structure and behavior) may allow their identification on different levels of abstraction or types of views. Also, design features in views generated through *Transformation* may serve as 'pattern' to already existing views (see examples in [Egyed 1999c]).
- **Common Interface:** Model elements exhibiting the same interconnectivity with other model elements may indicate correspondence (e.g. two differently named sets of classes interacting/interfacing with the same set of other classes).
- **Trace Observation:** Observation of test cases while they are executed/simulated can help establishing dependencies between high and low-level development artifacts. These observations of the executable combined with model knowledge of the system enables us to further reason about dependencies between high-level elements.

There are probably many other approaches not yet discovered by us, however, above list gives a reasonable overview. This paper will focus on the *Trace Observation* technique since we belief that it may enable quite reliable *Mapping* between high- and low level components and well as between high level components. It must be stressed, however, that probably no single mapping technique is superior. It is the union of *Mapping* technologies that ultimately gives enough useful clues as to how to support/constrain our *View Integration Framework*. Further, none of the above techniques must always yield correct results. It may very well happen that two different techniques come up with different results. In those cases heuristics (likelihood of error, etc.) may be used to make decisions - or a human being has to be consulted to make a final assessment.

Trace Observation for Reverse Engineering

The Trace Observation technique adds a reengineering element to view integration. It does this by combining the development model with the actual implementation and execution of the software product in order to identify dependencies between model elements. These dependencies can then be used to establish traces (mapping) between model elements which ultimately enables a feasible verification of the conceptual integrity of the software model. The Trace Observer technique is however not only limited to establishing traces between the product implementation and its corresponding system/software model; we

will later also discuss how dependencies between high-level model elements may be derived based on the observations made during system execution/simulation.

Figure 2 illustrates this in the case of a typical life-cycle view. A software development process is usually top-down and goes from the problem description, to analysis (requirements), to design, and finally to implementation (solution). Once a solution is found, the behavior of that solution can be observed using test scenarios (e.g. acceptance test scenarios, module test cases, etc.) that were created during development.

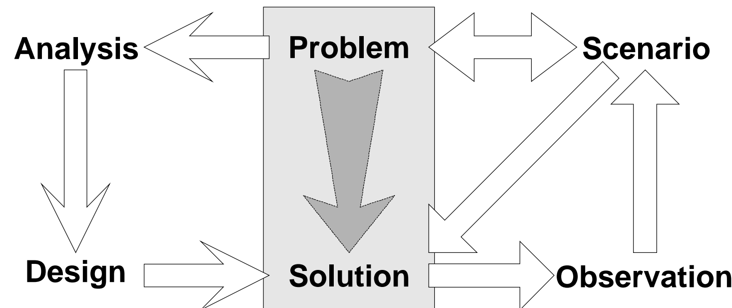

Figure 2. Combining Top-Down and Bottom-Up Engineering

The Trace Observer technique only works once an executable product, prototype, or simulation is available so that scenarios may be tested against it. In doing so, the internal activities of the product (or prototype) can then be observed and recorded. Since those observations correspond directly to scenarios and these scenarios in turn correspond directly to some model elements (requirements, architecture, or design), traces (mapping) between the implementation and those higher-level model element are established.

Although, trace observations are not a new invention, using them to automatically create dependencies between model elements (e.g. between architecture and design components) was explored in detail so far. It is our belief that this approach makes the *Mapping* activity less difficult and it even fits nicely into existing testing activities as will be discussed later. Furthermore, most of the enabling technologies for trace observation already exist. Tools for spying into software systems during execution and simulation are widely available COTS products (commercial-off-the-shelf). These tools are usually able to observe what lines of code are executed at what times, and what implementation functions (methods, etc.) are dependent on what other functions.

The remainder of the paper will therefore discuss how these low-level implementation traces are also useful for supporting our view integration framework.

TraceObserver supporting Mapping Activity

To evaluate the usefulness of the *Trace Observer* approach, we applied it to some real-world projects conducted at the University of Southern California (USC). There student teams developed products for their customer, the USC Library (see [Boehm et al, 1998] for a more detailed description). Those projects used a number of engineering techniques and tools, such as Rational Rose and the Unified Modeling Language (UML) to develop multimedia-related library products.

As part of their development process, the teams negotiated requirements for their respective systems and then continued applying the spiral model [Boehm 1988] with its anchor points [Boehm 1996] to describe the problem and to construct a working system in two semesters' time. In this paper we show one of those projects, the Inter-Library Loan System (ILL) [Abi-Antoun et al. 1998], to demonstrate the workings of the Trace Observer approach. The ILL system automates the borrowing process of books, paper, and other forms of media between Libraries. The system is being used by library patrons who wish to receive documents not available at their local library. ILL allows document requests to be filed via a web browser which in turn submits them as tagged electronic emails to a specially dedicated email

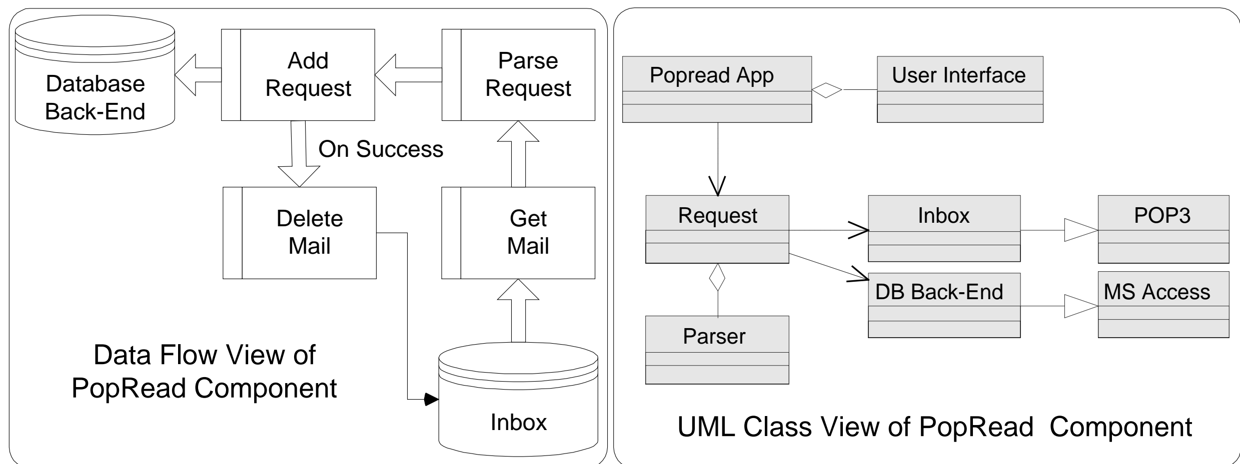

Figure 3. PopRead Component of ILL System [Abi-Antoun et al. 1998]

account (POP3 server). There the email messages get read and processed by another part of the ILL system (the PopRead component) which is supervised by the library administration. The PopRead Component is illustrated in Figure 3. The left hand-side shows the functional decomposition of the system as the ILL development team depicted it in their Software System Requirements Document (SSRD). The right hand-side shows the corresponding object-oriented design. Both views show the PopRead component in a high-level fashion.

PopRead reads emails messages from the inbox (POP3 account), parses them, and if the parsing is successful (valid), stores the identified requests on a back-end database. The design view further shows how the PopRead component was decomposed in an object-oriented fashion using UML class diagrams. The PopRead tool was implemented in C++ and is a Windows application with a typical menu centered user interface for specifying settings (location of Inbox and Database Back-End, passwords, etc.), activating check mail, and showing the About box (copyright).

Associated with the PopRead component, the ILL developers also created a number of usage scenarios associated with the model elements in Figure 3. One such scenario was *checking email and successfully parsing several requests and storing them in the database*. Another scenario was the *unsuccessful parsing of an email message*. Both scenarios can be derived from the data flow diagram in Figure 3. The design view also allowed scenarios to be broken down into test cases for individual components (e.g. testing the parser component).

It was part of the team's development process to create structural views (as the ones in Figure 3) as well as behavioral views (scenarios) and to properly interconnect them. The ILL team used mostly UML sequence diagrams to describe scenarios and UML class diagrams and data flow diagrams to describe structure. Whenever the *Mapping* between those two types of views were not obvious, the Software System Architecture Document (SSAD) was used to specify additional dependencies.

However, the ILL team did a rather poor job in describing how the PopRead components in Figure 3 (both sides) were implemented. Thus, the *Mapping* (traces) from both the data flow diagram and class diagram to source code are missing. Furthermore, the team did not specify how the dataflow diagram and class diagram relate to each other and, thus, traces between both high-level views are missing. Even in this rather simple example, those traces are not intuitive since functional and object-oriented decomposition were mixed.

The following sections will show how the trace observation technique can be used to derive those dependencies in a more automated fashion. As discussed previously, dependencies (mapping) are also the foundation for identifying view mismatches since they reveal what information need to be compared. We will therefore also discuss briefly how this is done.

Classes/Methods	check mail and successful parsing	settings only	unsuccessful check	successful check for a book request	startup, about, shutdown	settings and check
Classes/Methods	**47.27**	**9.98**	**30.92**	**47.27**	**7.4**	**40.9**
POP3.CPP	**76.79**	**4.76**	**69.05**	**76.79**	**4.76**	**73.81**
AA_CPOP3	0	0	0	0	100	0
CheckStatus	30.77	0	30.77	30.77	0	30.77
Connect	82.35	0	82.35	82.35	0	82.35
GetConnection	0	0	0	0	0	0
GetCutoffSize	0	100	0	0	0	100
GetMessageA	85.19	0	85.19	85.19	0	85.19
GetMessageCount	93.33	0	93.33	93.33	0	93.33
LogIn	90.91	0	90.91	90.91	0	90.91
LogOut	100	0	100	100	0	100
MarkForDeletion	92.86	0	0	92.86	0	0
ParseOKnn	100	0	100	100	0	100
ReadMultiline	100	0	100	100	0	100
ReadString	100	0	100	100	0	100
SetCutoffSize	0	100	0	0	0	100
WriteString	100	0	100	100	0	100
~AA_CPOP3	0	0	0	0	100	0
aboutDlg.cpp	**0**	**0**	**0**	**0**	**100**	**0**
GetMessageMap	0	0	0	0	100	0
OnInitDialog	0	0	0	0	100	0
OnOK	0	0	0	0	100	0
OnTimer	0	0	0	0	100	0
aboutDlg.h	**0**	**0**	**0**	0	**100**	**0**
AA_CAboutDialog	0	0	0	0	100	0
illdb.cpp	**71.43**	**7.79**	**63.64**	**71.43**	**9.09**	**71.43**
AA_CILLdb	0	0	0	0	100	0
AddEntry	100	0	0	100	0	0
Connect	0	0	0	0	0	0
Connect	70	0	70	70	0	70
Disconnect	100	0	100	100	0	100
GetSource	0	100	0	0	0	100
SetSource	0	100	0	0	0	100
~AA_CILLdb	0	0	0	0	60	0
MakeCleanEntry	100	0	100	100	0	100
illdbset.cpp	**96.34**	**0**	**96.34**	**96.34**	**0**	**96.34**
AA_CILLDBset	100	0	100	100	0	100
DoFieldExchange	100	0	100	100	0	100
GetDefaultConnect	0	0	0	0	0	0
GetDefaultSQL	100	0	100	100	0	100
GetRuntimeClass	100	0	100	100	0	100
mailReader.cpp	**36.62**	**29.57**	**29.58**	**36.62**	**4.23**	**59.15**
AA_mailReader	0	0	0	0	100	0
CloseMailbox	100	0	100	100	0	100
GetCutoffSize	0	100	0	0	0	100
GetMailboxInfo	71.43	0	71.43	71.43	0	71.43
GetMessageA	45.45	0	45.45	45.45	0	45.45
GetPort	0	0	0	0	0	0
GetServer	0	100	0	0	0	100
GetUser	0	100	0	0	0	100
MarkForDeletion	83.33	0	0	83.33	0	0
OpenMailbox	70	0	70	70	0	70
SetCutoffSize	0	100	0	0	0	100
SetPasswort	0	100	0	0	0	100
SetPort	0	0	0	0	0	0
SetServer	0	100	0	0	0	100
SetUser	0	100	0	0	0	100
getPasswort	0	0	0	0	0	0

Classes/Methods	check mail and successful parsing	settings only	unsuccessful check	successful check for a book request	startup, about, shutdown	settings and check
parsing.cpp	**54.58**	**0**	**13.54**	**54.58**	**0**	**13.54**
GetBoundedEntry	90.48	0	23.81	90.48	0	23.81
GetSingleLine	93.33	0	0	93.33	0	0
MonthToInt	32.26	0	32.26	32.26	0	32.26
ParseBookData	83.1	0	0	83.1	0	0
ParseCancelDate	85.29	0	0	85.29	0	0
ParseDissertationData	0	0	0	0	0	0
ParseMail	70.59	0	26.47	70.59	0	26.47
ParseMailDate	87.23	0	87.23	87.23	0	87.23
ParsePhotocopyData	0	0	0	0	0	0
ParseUserInformation	77.65	0	0	77.65	0	0
pop3dlg.cpp	**33.33**	**0**	**33.33**	**33.33**	**23.81**	**33.33**
GetMessageMap	0	0	0	0	100	0
Hide	0	0	0	0	0	0
OnInitDialog	0	0	0	0	100	0
OnStop	0	0	0	0	0	0
SetFlag	100	0	100	100	0	100
SetMessage	100	0	100	100	0	100
pop3dlg.h	**0**	**0**	**0**	**0**	**100**	**0**
AA_CPOP3Dialog	0	0	0	0	100	0
popread.cpp	**23.68**	**6.43**	**23.68**	**23.68**	**14.04**	**30.11**
InitInstance	0	0	0	0	100	0
AA_CMainWin	0	0	0	0	100	0
AddMessage	100	0	100	100	0	100
GetMessageMap	0	0	0	0	100	0
OnAbout	0	0	0	0	100	0
OnCheckRequests	85.71	0	85.71	85.71	0	85.71
OnClose	11.11	0	11.11	11.11	88.89	11.11
OnFont	0	0	0	0	0	0
OnSave	27.78	0	27.78	27.78	0	27.78
OnSetSettings	0	100	0	0	0	100
OnSize	0	0	0	0	100	0
~AA_CMainWin	0	0	0	0	100	0
CheckRequests	36.16	0	36.16	36.16	0	36.16
Unix2Dos	0	0	0	0	0	0
settingsDlg.cpp	**0**	**95.06**	**0**	**0**	**0**	**95.06**
GetCutoffSize	0	100	0	0	0	100
GetDatabaseName	0	100	0	0	0	100
GetMailServer	0	100	0	0	0	100
GetMessageMap	0	100	0	0	0	100
GetPassword	0	100	0	0	0	100
GetUser	0	100	0	0	0	100
OnBrowse	0	100	0	0	0	100
OnInitDialog	0	94.44	0	0	0	94.44
OnOK	0	100	0	0	0	100
SetCutoffSize	0	100	0	0	0	100
SetDatabaseName	0	100	0	0	0	100
SetMailServer	0	100	0	0	0	100
SetPassword	0	0	0	0	0	0
SetUser	0	100	0	0	0	100
settingsDlg.h	**0**	**100**	**0**	**0**	**0**	**100**
AA_CSettingsDialog	0	100	0	0	0	100

Figure 4. Summary of ILL Trace Observation for a number of scenarios

Mapping from High-Level Components to Implementation

Testing the system or some of its components and observing their traces is a straightforward activity. Figure 4 shows a summary of the observations of running six test scenarios and observing them via the Rational VisualQuantify tool. The 'h' and 'cpp' files correspond to the header and source files of the PopRead C++ code; the names underneath the source files correspond to the methods that were defined in these files. For instance, there is a *POP3* source file containing a *POP3* class with the methods *CheckStatus*, *Connect*, *GetConnection*, etc.

The six scenarios (columns) in Figure 4 are variations of the scenarios discussed previously. The *Unsuccessful Check* scenario reads emails from the POP account, tries to parse them, and finds that those mails do not correspond to the proper format. It can be seen that testing this scenario executes 31% of PopRead's source code lines (note that starting up the program and shutting it down is not included here). As such, this scenario uses the physical (implementation) classes *POP3*, *ILLdb*, *Mailreader*, and *Parsing*. Using this information, it is now straightforward to establish traces from scenarios to the product implementation. If those scenarios were also associated with high-level components (e.g. above scenario corresponds to *GetMail* and *ParseRequest* in Figure 3), then traces from these components to implementation can be established.

Figure 5 illustrates this process. The upper left side contains the dataflow (DFD) diagram from Figure 3. Two of the processes of that DFD diagram (*ParseRequest* and *GetMail*) have a scenario attached that is depicted on the right side of Figure 5. This scenario can now be tested against the product implementation and in doing so it can be observed that a number of classes/methods were executed[1]. We have therefore established a mapping (Trace) from the *ParseRequest* and *GetMail* processes to the physical implementation. This mapping is a derived mapping since it abstracts the transitive relationship we found via the test scenario.

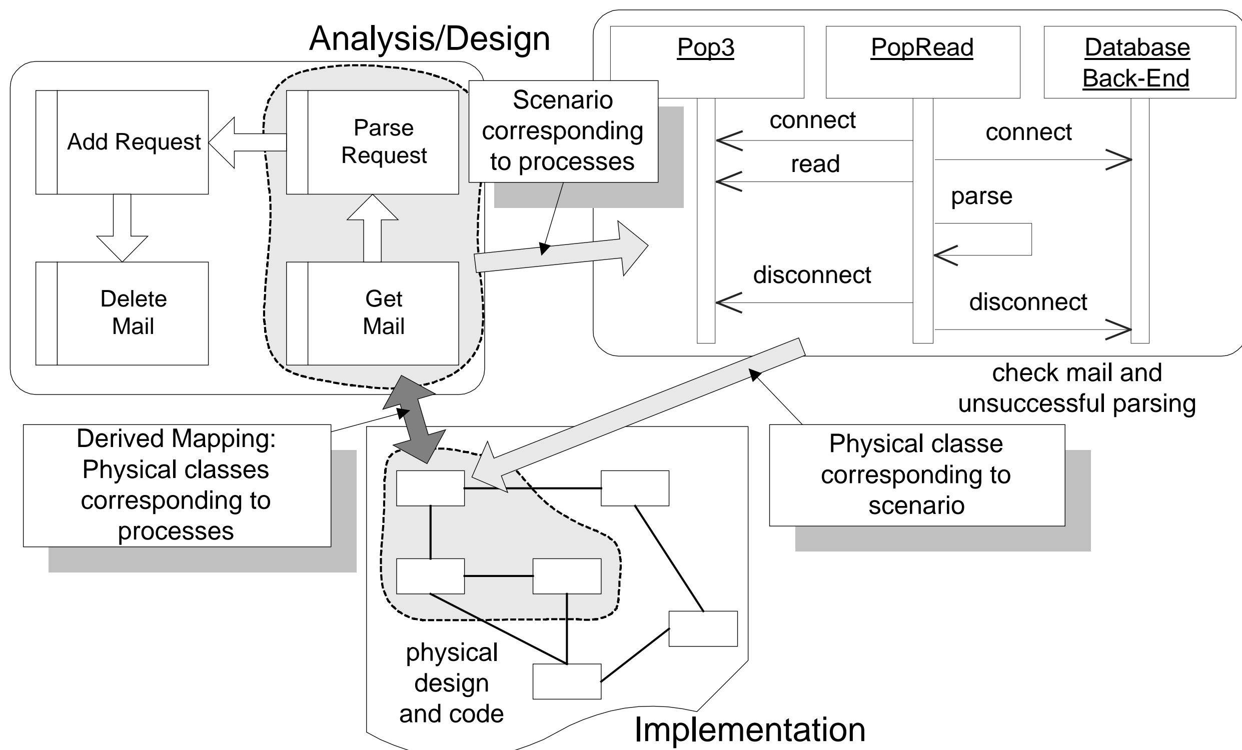

Figure 5. Trace Observation from DFD Processes via Scenario to Physical Design reveals Mapping

[1] The physical model would be too large for us to depict; refer to Figure 4 to see the impact of our scenario

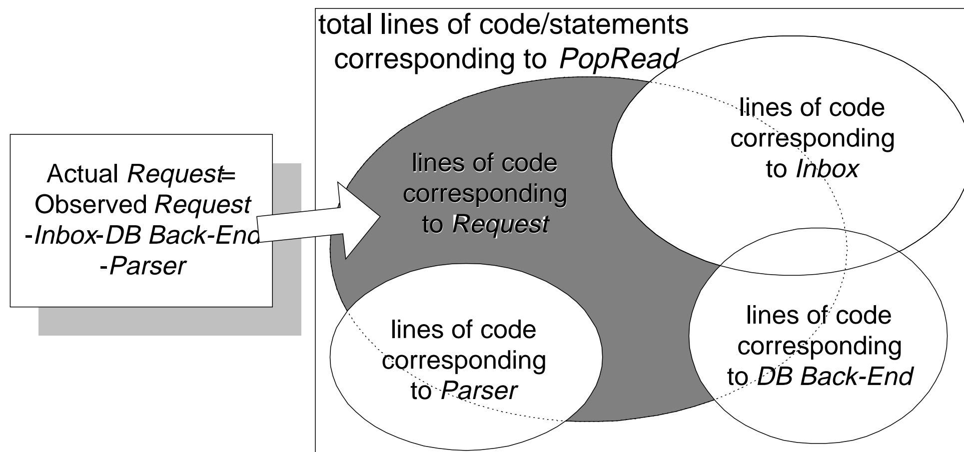

Figure 6. Deriving Actual Lines of Code versus Derived ones.

This trace observation process can be applied to other scenarios, e.g. to scenarios corresponding to the components in the class diagram in Figure 3. In this example, the traces from the high-level class diagram to implementation are more intuitive but still not trivial. For instance, observing trace scenarios for the class *Inbox* reveals dependencies to POP3.cpp and Mailreader.cpp, the class *POP3* corresponds to POP3.cpp, and the class *Parser* corresponds to parsing.cpp.

However, it is less trivial to see what the class *Request* corresponds to. Using a scenario for *Request* we find that it maps to *popRead*, *parsing*, *MailReader*, etc. This result is, however, not fully satisfactory since Trace Observer also identified dependent classes and methods as being part of *Request*. Figure 6 illustrates that relationship. The rectangle stands for all lines of codes (LOC) the PopRead program incorporates. The ellipses delimitate the LOCs corresponding to individual classes. Since *Request* incorporates a number of other classes, we may derive the actual LOC for Request by taking the observed LOC and extracting the LOC used by its dependents. This is doable because the knowledge what other classes are incorporated by *Request* is visible in Figure 3. Thus, we combine forward-engineered knowledge (design) with backward-engineered trace observations to derive more precise mapping. The Trace Observer technique discussed above can be automated to a large extend. The actual testing has to be done manually, however, even re-testing can be automated.

Mapping between High-Level Components

The previous section showed how the Trace Observer method helps in mapping (traces) high-level model elements to implementation. This technique can, however, also assist in establishing mapping between high-level components. Figure 7 highlights this process for the dataflow and class model we previously introduced. This figure illustrates again the mapping between the processes and classes and their corresponding implementations that can observed through their various scenarios. This example also shows that the processes *Get Mail* and *Parse Mail* use the same implementation artifacts (objects and functions) as the classes *Parser*, *Request*, and *Inbox*. Thus, the fact that both high-level model elements make use of the same implementation artifacts indicates a relationship between those high-level elements. We therefore use the footprints a test scenario leaves during system execution to compare model elements on a higher level. This technique not only works for structural views such as dataflow diagrams and class diagrams, but it can also be used to define relationships between behavioral views such as state diagrams or sequence diagrams. The latter is possible because trace observations not only reveal the lines of codes executed (source code is structural) but also who was calling whom and when (time line describes behavior).

To extend above example, we may further ask how the dataflow diagram maps to the class diagram in a more precise manner. The dataflow diagram in Figure 3 shows basically two scenarios: 1) read mail and unsuccessfully parse it; 2) read mail, successfully parse it, store it and delete mail. Figure 4 already shows all information on how to relate both scenarios where *Check mail and successful parsing*

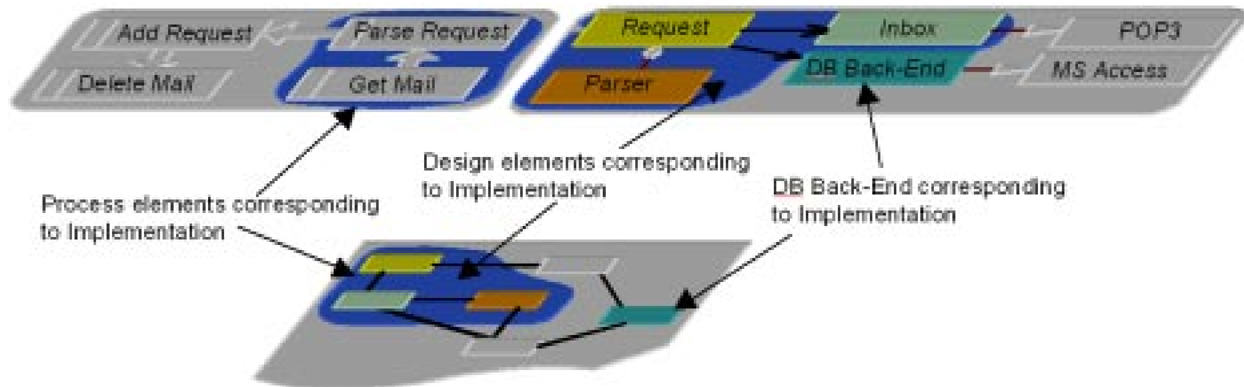

Figure 7. Using Implementation Footprints to Correlate High-Level Model Elements

(uses all four processes) corresponds to the former and *unsuccessful check* (uses processes *Add Request* and *Delete Mail* only) corresponds to the latter. If we now subtract the impact (footprints) of the former from the latter in Figure 4, we will see what lines of code correspond to the processes of *Add Request* and *Delete Mail*. In this example we find that *markForDeletion* (in *POP3* and *mailReader*), *addEntry* (in *ILLdb*), as well as *ParseBookData*, *ParseCancelData*, and *ParseUserInformation* (in *parsing*) remain. This result also conforms to our intuition with the exception of the parsing functions which probably belong to *Parse Request* and not to any of the latter. The fault here is that the scenarios do not exactly correspond to the processes. Thus, some manual intervention may be required to fix that. Nevertheless, even without intervention, the mappings derived through trace observation are fairly precise. If we use trace observations to infer high-level dependencies we may encounter to following cases:

1. Both scenarios use the same lines of code: this is rather unlikely, however, if it happens it may indicate the equivalence of both scenarios
2. Both scenarios use disjoint lines of code: strongly indicates that there is no relationship between both scenarios
3. One scenario uses a subset of the lines of code another scenario uses: indicates that former scenario is part of the latter
4. Both scenarios use some overlapping code but they each also use individual code: without additional information, no immediate conclusion possible. If this happens, one option is to break up those scenarios and to test and relate their pieces.

Another issue is also the level of granularity in which scenarios should be compared. For instance, should we compare them in a very fine granularity such as lines of code or class methods, or a more course granularity such as classes or packages? The higher-level the granularity, the less precise will be the derived relationships. E.g. what does it mean if both scenarios use the same package? On the other hand, lines of code are very detailed and the results of tracing scenarios on that level may vary from time to time (e.g. executing the same scenario multiple times in a row does not imply that the same lines of codes are executed). Thus, the granularity allows to trade off precision with complexity. For our purposes, we found it most useful, to compare scenarios on the level of class methods.

Besides trace information, this technique may also reveal vital additional test information such as what part of the code was tested more/less, or what part of the code was not covered through any test scenarios.

Combining Forward and Backward Engineering

Observing traces and creating mapping on top of them is a backward (reverse) engineering process. Even without an existing model (e.g. design), Trace Observer may still be used to reverse engineer design and architecture views. However, as above examples have shown, the backward

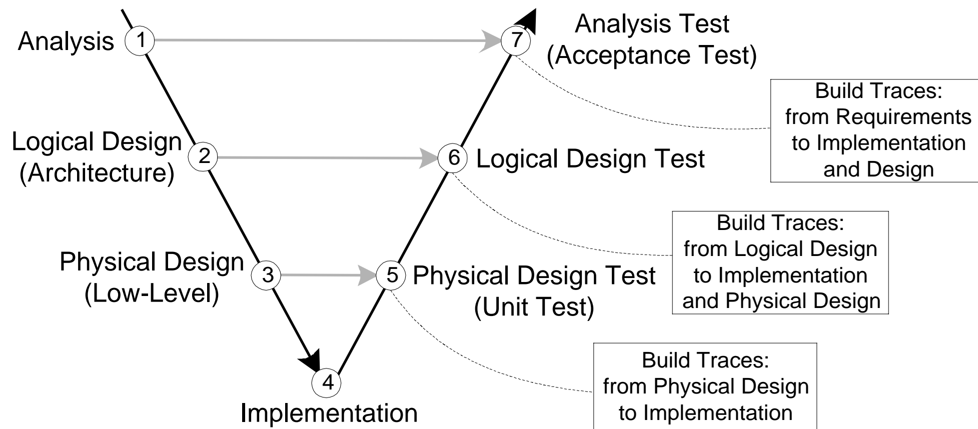

Figure 8. From Requirements to Implementation using TraceObserver

engineering approach can be effectively combined with forward-engineered views such as it was the case with the dataflow diagram and class diagram (e.g. *Request* class in Figure 6).

Nevertheless, on first glance it may seem that trace observation is not very useful during the initial software development cycle. Clearly, we need an executable product in order to observe test scenarios. As such it appears as if our Trace Observer technique can only be used for either incremental software development (product is build in increments and trace observations of previous increments may be carried over to next increment) or for system maintenance. However, this conclusion is not entirely correct. There are two ways on how to make the trace observation technique useful during the entire software development process.

First, the actual existence of the full software product is not required. We also apply this technique on prototypes, partially implemented products, or simulators. Although, the traces from high-level elements to, say, prototype would become invalid after some time, the mappings derived between high-level model elements with the help of the prototype (as it was discussed above) would still be valid afterwards.

Second, even if no prototype or simulator is used, it is still possible to make proper use of the Trace Observer during the current development cycle. Consider, for instance, Figure 8 that shows the development process in form of a V-model. Obviously, as long as no implementation is available no observations can be made, however, a development project is not finished after the first code has been written. Usually, software products are implemented in a bottom up fashion, where low-level components are implemented and tested first. These low-level components are then combined to higher-level components and tested until the system model is fully implemented - trace observation can be done in parallel.

After the low-level components have been implemented they need to be tested. If the corresponding design was done properly, then test scenarios for those components are available. These scenarios can now be used to test those low-level components. Although, initially no traces may be available, more and more trace information is created while the V&V process unfolds. After the low-level components have been tested, they can be combined to build higher-level (design and architectural) components. Again, those components may be tested in a similar fashion as above with the added value of some trace information now being available through the previous stage. This process may be continued until the system product is fully built and tested. At the end, when the complete system is about to be tested (e.g. acceptance test), all trace information of its major sub-components as well as their respective sub-components would already be available. Thus, the top-down development process combined with a mainly bottom-up implementation process enables the Trace Observer technique to be useful during the entire software development life-cycle.

Combing forward and backward engineering also combines the strengths of both of them. One of the major flaws of pure reverse engineering is that high-level views can never be fully reconstructed since

the implementation does not capture vital architecture or design information. Using forward-engineered architecture and design information therefore complements the lack of reverse engineering information.

TraceObserver supporting Transformation and Differentiation

Although, the strongest benefit of the Trace Observer technique is in establishing mapping between model elements, it is nevertheless also very useful in other areas such as the view transformation activity of our integration framework. To give an example, the calling dependencies we observe in the source code must be consistent with higher-level design elements.

For example, the implementation of *CheckRequests*, which corresponds to the *Request* class in the design (Figure 3), calls other implementation classes such as *CPOP3Dialog*, *CMainWin*, *mailReader*, *CILLDBset*, and *parsing*. Figure 9 shows an excerpt of the calling dependency observed while executing the *check mail and unsuccessful parsing* scenario. *CPOP3Dialog* and *CMainWin* correspond to *User Interface*, *mailReader* corresponds to *Inbox*, *CPOP3* corresponds to *POP3*, *CILLDBset* corresponds to *DB Back-End*, and *parsing* corresponds to *Parser*. Therefore, the implementation level calling graph implies a design level dependency from *Request* to *Inbox*, *DB Back-End*, *User Interface*, and *Parser*; as well as a high-level dependency from *Inbox* to *POP3*. The latter conforms to the design in Figure 3, however, the former dependency does not. Figure 3 does not show a high-level dependency from *Request* to *User Interface*. Thus, we may have identified a mismatch between the implementation and design.

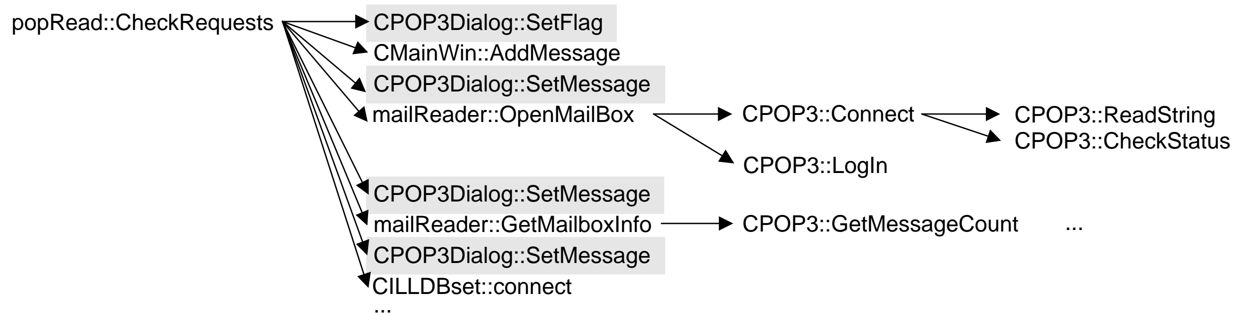

Figure 9. Calling Dependency of Implementation Methods/Classes

Besides, structural transformation and differentiation, the Trace Observer technique may also be used for behavioral analysis. Again, it is the combination of forward and backward engineering that allows effective view integration. It is however out of the scope of this paper to elaborate these issues in more detail.

Conclusion

The lack of view integration is not a new discovery. Many researchers talk about the need of keeping model(s) consistent ([Rechtin 1991][Sage-Lynch 1998][Garlan et al 1997][Finkelstein et al. 1994][Wang et al. 1997] to list just a few). Currently process models provide the only guidelines on what activities one can do to improve the conceptual integrity of architectures. However, these process models are hardly automatable. We have therefore created a view integration framework and it, with its corresponding activities of *Mapping*, *Transformation*, and *Differentiation*, enables a more automated assistance in identifying and even resolving mismatches between views.

The main focus of this paper was the *Mapping* activity of our framework since it is a particularly hard problem. This paper therefore introduced the Trace Observer technique and described how it can be applied to (semi) automatically cross-reference model elements among views. The Trace Observer technique supports the following tasks:

- Mapping between low-level (implementation) and high-level components

- Mapping between various high-level components via dependency of low-level components
- Transformation of low-level behavioral observations to higher-level structural information
- Transformation of low-level structural/behavioral diagrams to corresponding high-level structural/behavioral diagrams

The major advantage of this approach is that some tool support is already available via COTS packages. However, more tool support is need to provide automated mapping support. Although trace observations alone are very powerful in establishing inter-view correspondence, other view mapping techniques are still needed to supplement its shortcomings. It is the combination of techniques that will ultimately show the most promising results. The Trace Observer technique adds, however, a powerful ingredient to this mixture.

We have furthermore shown that it is possible to (at least partially) automate the task of mapping and we also demonstrated one example on how mismatches may be identified via trace observations. Since computers are much more efficient in mapping and comparing views and since consistency checking is probably still the largest development activity done manually, this implies that substantial manual labor can be saved.

References

Abi-Antoun, M., Ho, J., Kwan, J. (1998) "Inter-Library Loan Management System: Revised Life-Cycle Architecture," Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA.

Boehm, B.W. (1988) "A Spiral Model of Software Development and Enhancement," *Computer*, May, v. 21 no. 5, pp. 61-72.

Boehm, B.W. (1996) "Anchoring the Software Process," *IEEE Software*, July, v.13 no.4, 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.

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

Egyed, A. (1999a) "Integrating Architectural Views in UML," Qualifying Report, Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-99-514.

Egyed, A. (1999b) "Automating Architectural View Integration in UML," Technical Report USCCSE-99-511, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 (submitted to ESEC/FSE'99).

Egyed, A. (1999c) "Using Patterns to Integrate UML Views," Technical Report USCCSE-99-515, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 (submitted OOPSLA'99).

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1994) "Inconsistency Handling in Multi-Perspective Specifications," IEEE Transactions on Software Engineering, August, pp. 569-578.IEEE (1998) "Recommended Practice for Architectural Description," Draft Std. P1471, IEEE.

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

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

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

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.