EXTRACTING HIGH-LEVEL ARCHITECTURE FROM EXISTING CODE WITH SUMMARY MODELS

Nikolai Mansurov
Klocwork, Inc, 1 Chrysalis Way, Ottawa K2G 6P9,
Ontario, Canada
mansurov@klocwork.com

Djenana Campara
Klocwork, Inc, 1 Chrysalis Way, Ottawa K2G 6P9,
Ontario, Canada
campara@klocwork.com

ABSTRACT

Evolution of existing large telecommunications software currently became an important issue. Efficient methods are needed to componentize existing software - identify existing components and gradually improve their boundaries and interfaces. We describe our approach to componentization, which involves precise yet high-level models of existing software. Central to our approach are certain editing operations that make models more high-level yet preserve precision of models. Raising the level of abstraction of the initial model of software is essential for communicating the cleaned-up architecture to the development team, analyzing, optimizing and managing it.

We introduce the concept of a "*summary model*" that is scalable, maintains precision of component interfaces, and maintains links to the source code. We report using the summary models in our Klocwork Architecture Excavation methodology.

We show that in order to accelerate Architecture Excavation, it is important to build various supporting views. These views need to be also precise and scalable, i.e. work at the same level of abstraction as the emerging components. We expand the scope of summary models and present them as a unified approach to reverse engineering.

software architecture, modularization, evolution, model-driven architecture

1. Introduction and background

Significant research and development efforts are being invested into acceleration of development and deployment of new systems. On the other hand, evolution of existing software receives considerably less attention. Evolution of existing software addresses development of new features as well as changes to the source code that has already been developed (legacy code). Designing a new feature to of an existing system can be more difficult, than designing an equivalent system from scratch, because all forward engineering decisions are constrained with legacy decisions.

In order to keep up with increasing market pressures, software development organizations are required to sooner or later undergo costly upgrade of their software. A typical scenario for a telecommunications project (also most real-time embedded software projects) is that the core system has emerged as a "proof of concept" then experienced rapid and usually unexpected growth as investments were poured into the project. The quality of the underlying infrastructure is routinely sacrificed in favor of brute-force adding more features for the market. The initial structure of the software *erodes* and new features are becoming more difficult to add. Quality issues start mounting. Several mergers and acquisitions can further aggravate the situation. All this creates significant pressures for improving of the infrastructure of the software, which can be described as *componentization* of existing software.

Componentization is an *architectural* activity. It identifies and improves *modularity* of the software. In order to improve modularity of existing software the following steps need to be performed:

- identify components,
- introduce boundaries around the candidate components by aggregating the sub-components,
- represent the new component by a single graphical symbol and then
- investigate the interfaces between components to identify modularity issues,
- edit components by adding more contents to the component or removing extra contents from the component
- as better modules emerge, better architecture mechanisms can be introduced both for enforcing the module boundaries and for inter-module interfaces

Tools for componentization require precise information about relations between components. Currently here exist a large number of the so-called *reverse engineering tools* that extract such information for existing source code. However, existing reverse engineering tools is failing to solve the componentization problem. In our opinion, this happens because traditional tools fail to provide sufficiently high-level views of existing code. In other words, current reverse engineering tools extract individual relations between symbols in the software, but do not deal with module interfaces.

To mitigate the above problem is the motivation for the research and development performed by Klocwork. Our approach to the evolution of existing software is based on the so-called Architecture Excavation methodology [2], where the precise initial model of existing software is systematically transformed to raise its abstraction level in order to capture emerging components and their interfaces.

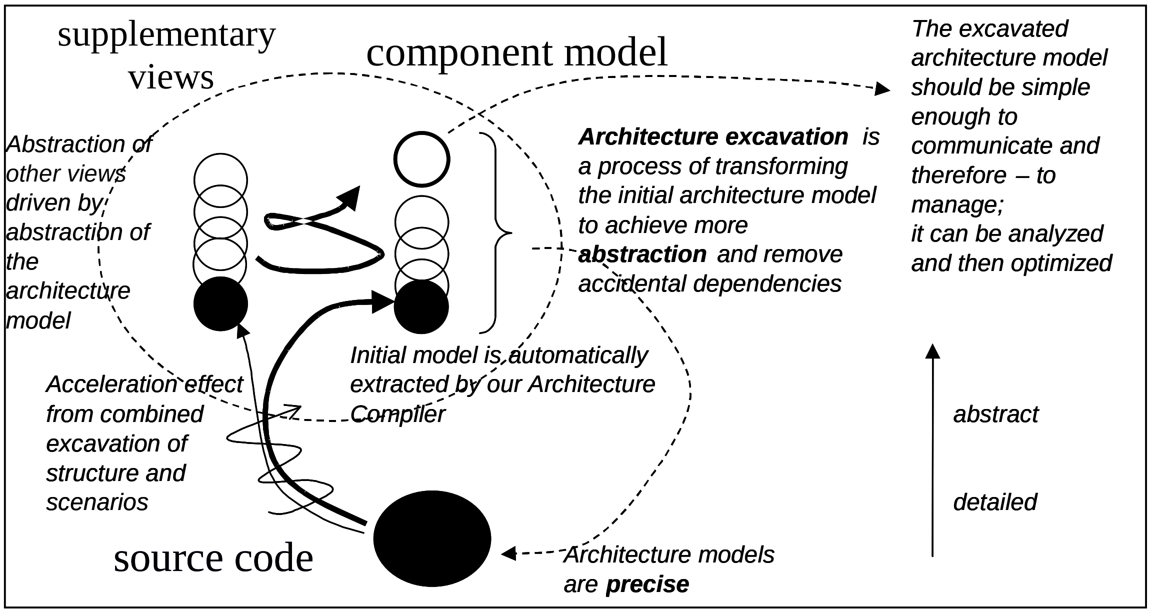
The formal basis of our Architecture Excavation method is the so-called "summary model". Summary models focus on components and their interfaces. Since components can consist of sub-components, summary models support unlimited scalability. Summary models support transformation of components (relocating symbols between components) and preserve precision of the interfaces between components during these transformations. This means, that as components emerge and become more abstract by embracing sub-components, the interfaces between components are still precise. The component model guides the excavation process.

The rest of the paper has the following organization. Section 2 describes the Klocwork Architecture Excavation method. Section 3 provides a formal definition of the "summary models". Section 4 shows how supplementary architecture views can be described using the "summary model" concept. We consider various software architecture views from the maintenance perspective and describe how these views can be used as "summary models". Section 5 compares Architecture Excavation approach with related approaches. In Section 6 we discuss how summary models allow penetrating the "legacy barrier" for transition to OMG's Model-Driven Architecture. Section 7 draws some conclusions.

2. Architecture Excavation

In this section we describe systematic process for creating high-level, architecturally significant component models of existing software - the so-called Klocwork *Architecture Extraction* method. The objective of this methodology is to create a formal architecture model of existing software that is high-level enough to be reasoned about and to be communicated to the development team. Therefore such model can be also used to manage the architecture of the existing software, to analyze problems with the current architecture, and to plan and coordinate the clean-up initiatives. This approach is particularly beneficial in a context where existing design and architectural documentation is absent, imprecise, or obsolete.

An initial structural model is automatically extracted from the source code (see Figure 1). Then the model is manually *transformed* through a series of iterations to



supplementary views
component model
The excavated architecture model should be simple enough to communicate and therefore – to manage; it can be analyzed and then optimized
Abstraction of other views driven by abstraction of the architecture model
Architecture excavation is a process of transforming the initial architecture model to achieve more abstraction and remove accidental dependencies
Initial model is automatically extracted by our Architecture Compiler
Acceleration effect from combined excavation of structure and scenarios
abstract
detailed
source code
Architecture models are precise

Figure 1 Overview of the Architecture Excavation process

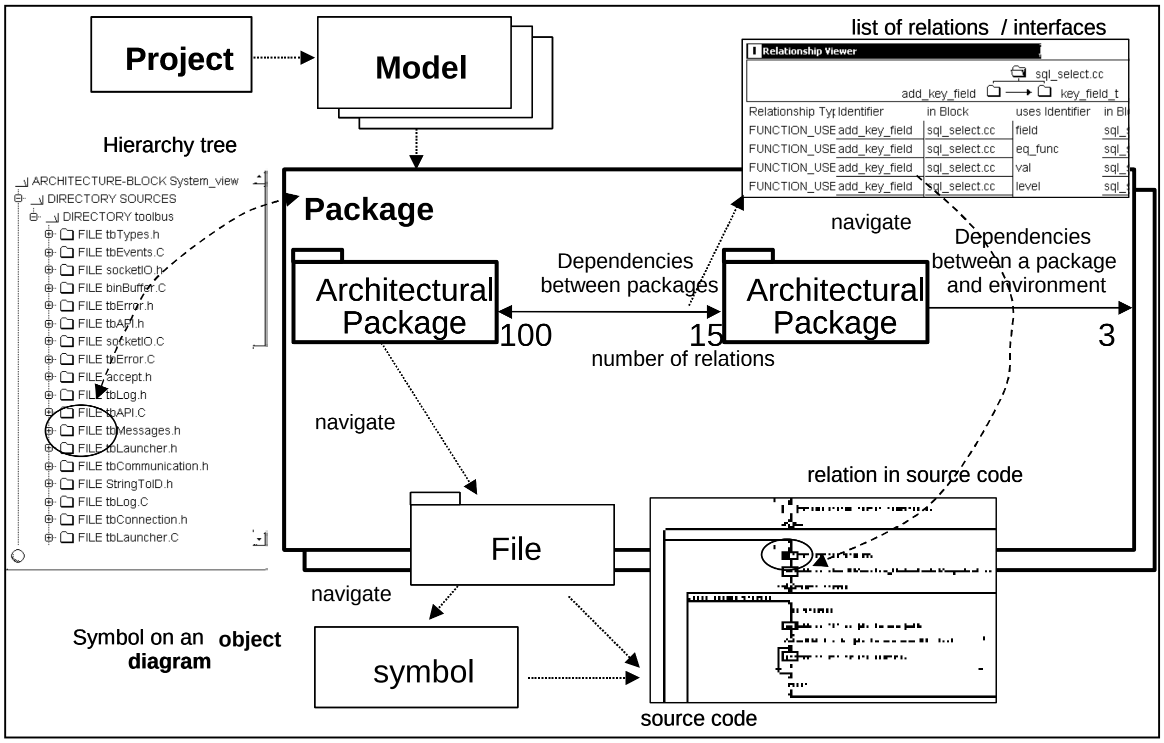
increase the level of abstraction and to remove any accidental dependencies between components. The whole process preserves the precise information that is initially extracted from the source code. Therefore, at each step the Klocwork architecture model is *precise* with respect to the implementation. The resulting model can be used for architecture analysis and management.

The structure of Klocwork architecture models is further illustrated in Figure 2. These models are based on UML package and objects diagrams [3]. The package model (see Figure 2) presents high-level "modules" and dependencies between them. The model is hierarchical, and the hierarchy is also presented as a separate view. At the bottom of the hierarchy of packages are files. Further, the files contain partial object diagrams, consisting of symbols for classes as well as individual procedures, variables and types.

Dependencies between packages (modules of existing software) in Klocwork diagrams are derived from the relationships between individual symbols (elementary language-dependent entities of the source code, like types, procedures, variables, macros, etc.). Whenever one collection of entities (for example, classes, files, directories, etc.) is grouped into a package and another collection – into the second package, relations between individual symbols in different packages contribute to dependencies between the two packages. This means that each package diagram shows precise interfaces between packages. It is possible to investigate such interfaces by navigating through dependency links and viewing the list of all individual items of the interface, or even the complete list of individual relations between symbols in each package.

Each relation is associated with a certain location in the source code. It is possible to navigate from any high-level package diagram directly to the source code level, for example to look for clues as to why a certain relation exists, and what is its responsibility. Klocwork Suite uses a flowchart graphical representation for viewing source code, independently of the programming language used.

The underlying meta-model of Klocwork Component Models is significantly more complex than the meta-model for the corresponding UML package and object diagrams, because of the support for unlimited aggregation as well as the requirement to keep precision and associations to the source code.



Relationship Typ	Identifier	in Block	uses Identifier	in Bl
FUNCTION_USE	add_key_field	sql_select.cc	field	sql_
FUNCTION_USE	add_key_field	sql_select.cc	eq_func	sql_
FUNCTION_USE	add_key_field	sql_select.cc	val	sql_
FUNCTION_USE	add_key_field	sql_select.cc	level	sql_

Figure 2 Overview of Klocwork Component Models

Architecture Excavation is supported by two main operations on Klocwork Component Models: *aggregation* and *trimming*. Aggregation is a visual operation that allows one to select several blocks and group them by creating a new level of the hierarchy. The model automatically recalculates the dependencies between the packages at all levels. However, aggregation alone does not allow raising the level of abstraction of the architecture diagrams, unless it is supported by the trimming operation.

Trimming is a visual operation that allows moving blocks between packages. Usually, we move individual symbols from the low-level object diagrams from one module to another. This means that trimming usually involves a virtual editing of files (only the corresponding part of the model is modified of course). Trimming allows eliminating dependencies between higher-level components that are induced by accidental placements of functionality at the file level. This happens far too often, when the physical architecture of the software is not managed properly.

Architecture Excavation actively transforms the initial architecture model. Therefore, there is no clear boundary between Architecture Excavation and (architecture analysis) and architecture optimization. Usually, excavation deals with minor and accidental problems that prevent the understanding of the “real” architecture and therefore prevent the exposure of the “real” architecture problems due to the overwhelming complexity of the raw data.

Architecture Excavation involves a large amount of *software understanding* (and common sense!) in order to select families of symbols to aggregate and symbols to trim.

Supplementary views can facilitate such understanding. Supplementary views allow a temporary reduction of the complexity and therefore "unfreeze" the flow of the Architecture Excavation. A supplementary view allows creating a temporary architecture view of all objects involved in a certain primary view. Usually this provides enough insight into the architectural abstractions involved. A partial architecture view can be *quickly* excavated and then applied to the more complete context.

In this paper we provide a more general framework for Architecture Excavation and supplementary views that can take advantage of the precise and scalable operations on components. Section 3 describes the underlying meta-model of Klocwork Component Models – the concept of "*summary models*". Section 4 provides more detail on how the summary models can be used to integrate multiple architecture views.

3. "Summary models" a *meta-model* of Klocwork Component Models

The fundamental concern of our approach is to produce high-level architecturally significant graphical component models of existing software. Section 3.1. provides a more formal definition of semantic and visualization aspects of "summary models". Section 3.2. describes fundamental editing operations on "summary models". Section 3.3. discusses some properties of "summary models" that make them attractive for dealing with existing software.

3.1. Definition of summary models

The UML meta-model of the Klocwork Component Models is a *package graph* that consists *nodes* and *edges*. We will further differentiate between the so-called *regular* nodes and edges and *summary* nodes and edges. *Regular* nodes and edges are of some domain-specific *types* (defined by the source language). Each regular node and each regular edge of the graph is "*anchored*" to the source code, i.e. contains a reference to a particular *location* in existing source code. This is a fundamental property of our approach that allows other supplementary architecture views to "piggyback" the component graphs by using the anchors as the common denominator between views.

Fundamental operation on a "summary model" is called "*aggregation*": nodes of the graph can be grouped into *summary* nodes. Summary node is connected to other nodes

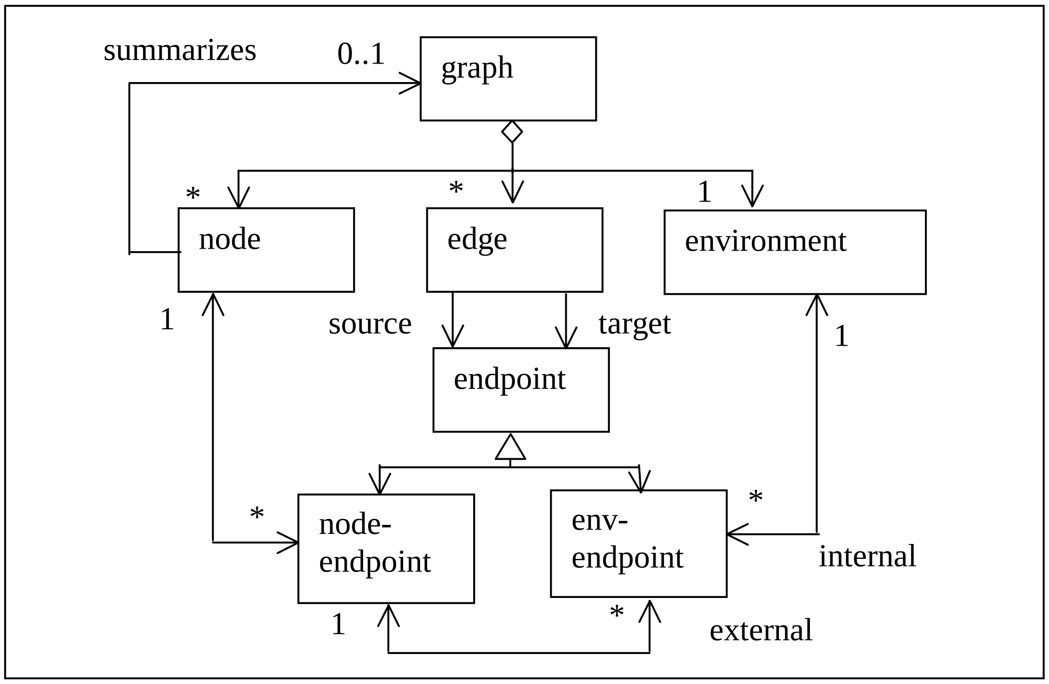


Figure 3. Meta-model of Klocwork Component Models

with summary edges. Summary edges are computed automatically as follows: let `N={n1,…,nk}` be nodes which are summarized into a new node `ns`. Let's assume that nodes `ni,..nj` in `N` are nodes which have edges E=`{ei,…,ej}` to some node `x` not in `N`. Then node `ns` has a single summary edge `es` to the node `x`. Summary edge `es` has an additional attribute called *density*, which is the total number of summarized regular edges. Density of a regular edge is 1.

Summary edges are also "*anchored*" to the source code. Let's assume, that each edge `ei,…ej` has attribute *anchor* `li,…lj` (location in the source code). Then the summary edge `es` will have an attribute `l={li,…,lj}` (set of all anchors of the summarized regular edges).

“Aggregation” operation creates the hierarchy relation on the set of component diagrams: the set of grouped nodes `N` are called the *child* nodes on the *summary* (parent) node `ns`. These nodes now constitute a separate diagram, which is associated with the summary node `ns`.

“Aggregation” operation introduces the concept of the “*environment*” of a package diagram, which can be the *endpoint* of certain edges. Let’s consider the diagram with the summary node `ns`. Let’s assume it contains a summary edge `es` to some node `x` on the same diagram. Let’s now consider the diagram for the set of child nodes `N` of the summary node `ns`. This diagram will contain set of edges a from nodes `ni,…,nj` to the “environment” of the diagram.

Another key operation on summary models is the so-called “*trimming*” operation: a child node `ni` is moved out of one summary node `ns1` and into another summary node `ns2` (possibly at a different level of hierarchy). Trimming operation can be considered as “ungrouping” a package and then re-grouping it to reflect the updated position of one of the nodes. Therefore, trimming operation updates the edges in the “summary model” according to the new “aggregation”. Note, that after applying trimming operation, some edges may disappear and some new edges may appear.

In order to display the model, we design a *graphical language*, where nodes and edges are mapped onto certain graphical *symbols* and *lines*. Then we design a suitable *layout*, which positions symbols and lines within a *frame*, possible in some meaningful way, representing essential patterns of the component graph. Often, a different layout can be used to make the model more useful.

Thus, a “summary model” is hierarchical set of high-level graphical diagrams, anchored to source code. Nodes of the summary models can be aggregated and trimmed. Aggregation and trimming operation maintain precision of the package graph in terms of the densities of the edges. Packages are used to represent modules (components) within existing software.

3.2. Operations on “summary models”

Definition 1. **Aggregation.** Aggregation modifies the model by introducing a new graph and a new summary node based on a set of nodes selected in a certain graph. Selected nodes are replaced by a summary node in the original graph and become the nodes of the new child graph. The edges from selected nodes to other nodes and the environment of the original graph become the edges to the environment in the child graph. In the original graph, these edges are replaced by the so-called summary edges to the summary node.

Definition 2. Detalization. Detalization is an inverse operation to aggregation. Based on a selected summary node, this operation removes the child graph and replaces the summary node with the nodes from the child graph. The summary edges are replaced with the edges to the environment of the child graph, which are now connected back to their original endpoints in the parent graph. The composition of summarization of a set of nodes and detalization of the summary node does not change the model.

Definition 3. Trimming. Trimming modifies the semantic domain by moving selected node from one graph (the donor graph) to another (the acceptor graph). Note, that *is_child* relation on graphs in semantic domain defines one or more tree structures. Let’s assume, that there always exists the top graph, which contains summary nodes for all graphs without a summary node. The trimming operation is defined as follows. Identify the least common ancestor (LCA) of the donor and the acceptor graphs. Recursively ungroup the donor and the acceptor graphs into the LCA, while preserving selection sets in each graph. Exclude the trimmed out node from the corresponding selection set in the donor graph and add it into the selection set of the acceptor graph. Recursively group the donor and the acceptor graphs using the preserved selection set with the above modification.

3.3. Properties of “summary models”

“Summary models” preserve the precision of the interfaces between components during aggregation and trimming operations. This means, that as components emerge and become more abstract, the interfaces between components are still precise which allows componentization to proceed.

“Summary models” also preserve links to the source code. This allows seamlessly integrate various supporting architecture views as they all are traceable all the way down to the source code. As the result, supplementary views “piggyback” the structural views in that these views are automatically aggregated as the underlying structural view is aggregated. Therefore, all supplementary views remain at the same level of abstraction as the component models. Supplementary views focus the Architecture Excavation process by allowing the analyst to concentrate on a smaller subset of candidate blocks.

Summary models can be restructured, to explore potential solutions to maintenance problems. In our experience, "summary models" allow significant acceleration of maintenance activities. Sources of acceleration include the following: (1) “what-if” analysis and restructuring using the summary model (dramatically reduces the number of edit-compile-test cycles); (2) using scenarios to focus analysis and understanding of software; (3) using process view enables understanding of inter-process communication and synchronization; (4) dealing with graphical representation of code instead of linear textual representation of code; (5) cross-referencing using database instead of browsing through multiple files; (6) impact analysis;

Some manual intervention in creating "summary models" of existing systems is inevitable, since many important entities are not directly represented in the source code, therefore their selection has to be performed by experienced designers. Building summary models manually ensures the resulting models have high level of abstraction.

either a particular statement in the source code, a procedure call (connecting the procedure call statement and the start statement of the called procedure), a return from a procedure (connecting the return statement with the call statement in the calling procedure), or an inter-process communication event (connecting two operating system calls). The type of an event is captured by the *semantic tag*. Each event is *anchored* to a certain *location* in the source code of the system. Each event contains information about the actual *source statement*, the name of the so-called *key entity*, involved in the event and the so-called *identifier of the entity* (which is a unique representation of this entity in the project repository of the Klocwork inSight tool. These identifiers are used to map scenarios onto architecture diagrams [5,6,2]. The key entity is selected by the user during event capturing. Scenario corresponds to a certain *thread* through the system.

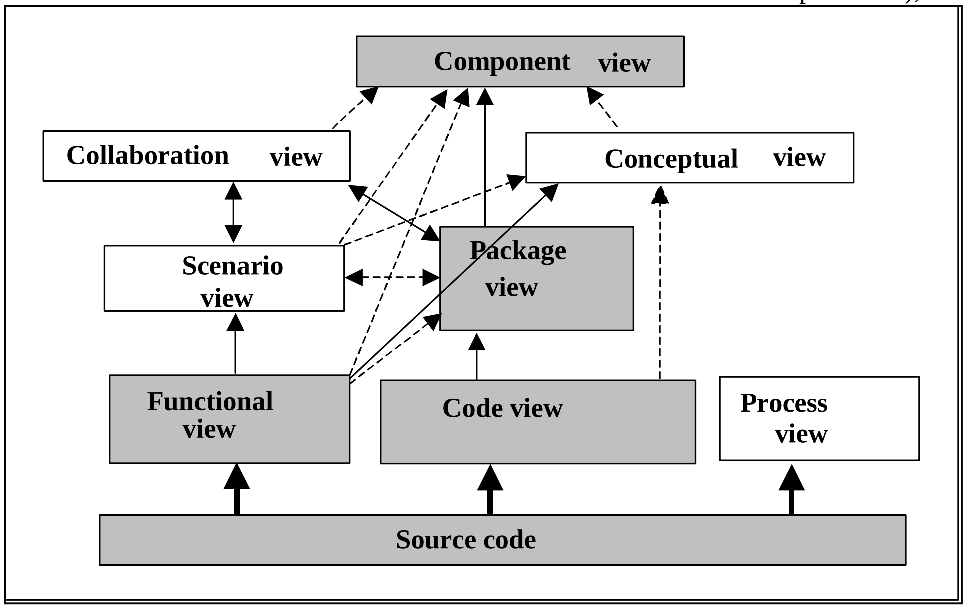


Figure 4. Architecture views from the Architecture Excavation perspective

4. "Summary models" as a unified foundation for supplementary architecture views

The concept of "summary models" can be applied to other well-known *architecture views* of the software, already adopted in forward engineering methodologies. In order to describe a particular view as a "summary model" one needs to describe the meaning of nodes and arcs in the semantic graph and the meaning of the aggregation operation.

The idea of separating software architecture into different views is not unique. Although there is not yet a general agreement about which views are the most useful, the reason behind multiple views is always the same: Separating different aspects into separate views helps people manage complexity [4]. *Code view* describes how the source code, binaries and libraries are organized in the development environment. For example, source code resides in files, directories and libraries. *Module view* describes functional decomposition and layers, which are orthogonal structures. Typical terms are subsystems, modules, layers, imports and exports. *Conceptual view* describes the system in terms of its major design elements and relationships between them. Typical elements are components and connectors. *Process view* describes the system's dynamic structure. Typical elements are tasks, processes, threads, RPC and events.

Scenario Views show architecturally significant sequences of events in the system [5]. An *event* can be

Source threads are used as the basic representation of scenarios. Klocwork Flowchart view is particularly useful for navigating through the source code and tagging scenario events [6]. The information, contained in the source thread can be used to animate this thread in the Klocwork inSight tool, to map it onto the architecture of the system, and to transform it into a number of other representations [6,2].

Collaboration Views project scenarios onto component models. Such projection of scenarios can be performed by the Klocwork Suite. Unique identifiers of entities are received from the Klocwork inSight as the user anchors certain locations into a source thread. These identifiers are stored with the thread. The Klocwork inSight is capable of projecting an individual entity onto any architecture diagram using unique identifier associated with that entity [5]. The Klocwork inSight is also capable of highlighting relations on structure diagrams. Klocwork Suite is capable of building two distinct flavors of collaboration views.

Collaboration diagrams provide a simple integration of a structural view and a scenario by highlighting the edges that are involved in this scenario (edges can be further numbered to indicate sequencing). Use Case Maps are more intuitive [2]. Use Case Maps display the so-called time thread through the structural view.

Functional Views show relations between the entities at the programming language level. Functional views are essential for understanding existing software. In fact, most reverse engineering tools only deal with functional views [7]. In our Architecture Excavation approach, functional views are used when building other views (see Figure 4).

When these views are recovered from an existing software system and then abstracted using the operations of aggregation and trimming.

Figure 4 illustrates the architectural views from the Architecture Excavation perspective. Several views can be automatically extracted from the source code (represented as thick lines). These are the functional views (language-dependent entities and relations), code view (files and relations between them) and Process view (processes, threads and inter-process communication relations).

Other views require manual interaction in order to capture architecturally significant information. Code view is directly abstracted into the package view and then – into component view. This is the main transformation, supported by the Architecture Excavation method (represented as shaded boxes at Figure 4).

Scenario views are based on the functional views. As can be seen from Figure 4, Scenario Views assist in creating other views (including package, component and conceptual views). This is represented by dashed lines at Figure 4. Scenario Views "piggyback" package views (represented by bi-directional lines at Figure 4).

Collaboration views both Scenario Views and Package Views, since a collaboration view visualizes a projection of a scenario onto a package or component diagram.

5. Comparison to related work

The concept of summarized structural views (or landscapes) was independently discovered by Holt [8], Krikhaar and Feijs [9], Murhpy and Notkin [10] and our Klocwork team [5].

There exist several tools, which build structural views of existing software systems [11,8,12]. In [12] such tools are described as "software architecture visualization tools", however they are limited to code and module views. The majority of tools only support browsing, such Software Bookshelf [11]. They display structure views and allow users to explore them, but have only primitive query mechanisms [11]. Other tools allow users to query and build views, such as ManSART [13]. However, these tools operate only on architecture level facts, and do not extend to the entity level.

Several groups have attempted to automatically extract scenario views of existing systems [14,15]. Their efforts where concentrated on dynamic extraction of scenarios, by running a suitably instrumented system in the run-time environment. In contrast, the Klocwork Suite tool relies entirely on static extraction, bypassing the need for the run-time environment [16].

6. Klocwork Component Models and transition to OMG's Model Driven Architecture (MDA)

Klocwork Component Models can have other applications that go beyond modularization of existing software. Maintenance of existing software is only one aspect of a much broader issue of the *evolution* of software. The infrastructure for modern enterprise computing is changing rapidly [18]. This means that the software applications need keep pace with these changes, which in turn requires even larger restructuring that mere modularization. It is becoming obvious that the source code itself is not the right artifact for maintenance and evolution since it mixes platform-independent and platform specific concerns.

OMG's Model Driven Architecture is the modern approach to mitigate these problems [18]. MDA is an approach for engineering *future-proof* software. The key concept of MDA is the transition from maintaining the code to *modeling*. MDA also advocates separation of Platform-Independent Models (PIM) and Platform-Specific Models (PSM). The implementation code for selected platform is derived from the PIM through PSM with the use of automated code generation. This is a very promising approach.

However, existing software introduces the so-called "legacy barrier" for transition to MDA [16], since it required excavation of PIMs of existing software.

We believe that summary models that support unlimited aggregation, trimming and links to the source code allow creating architecturally significant models of existing software that can be gradually refined into PIMs for existing modules, and thus will allow integration of existing modules into the MDA–oriented software development. Summary models correspond well to the models used during forward engineering, therefore they allow penetrating the so-called "legacy barrier" for better adoption of advanced software engineering methodologies in industry.

7. Conclusions

Evolution of existing software is rapidly becoming an important issue. The science and art of *Software Archeology* is rapidly becoming a reality of big software houses, especially as new infrastructure technologies are introduced [16,18].

We have presented our Architecture Excavation approach to maintenance and evolution of existing software, based on the concept of "summary models". Summary models are both *scalable* (they support unlimited aggregation) and *precise* (with the respect to the existing software that they represent). The elements of summary models represent "real" entities and relations from the source code of the system (as opposed to “desired” elements in the models of software built during the forward engineering process). Therefore, summary models allow managing software development from the architecture perspective.

Instead of focusing on relations between individual symbols at the code level, “summary models” visualize interfaces between high-level, aggregated packages, which represent modules or components in existing software.

Summary models can be restructured, to explore potential solutions to maintenance problems. In our experience, "summary models" allow significant acceleration of maintenance activities. Sources of acceleration include the following: (1) “what-if” analysis and restructuring using the summary model (dramatically reduces the number of edit-compile-test cycles); (2) using scenarios to focus analysis and understanding of software; (3) using process view enables understanding of inter-process communication and synchronization; (4) dealing with graphical representation of code instead of linear textual representation of code; (5) cross-referencing using database instead of browsing through multiple files; (6) impact analysis;

Some manual intervention in creating "summary models" of existing systems is inevitable, since many important entities are not directly represented in the source code, therefore their selection has to be performed by experienced designers. Building summary models manually ensures the resulting models have high level of abstraction. However, our experience demonstrates, that it is quite possible to keep the process of architecture excavation cost-effective, incremental and scalable.

The validation of this approach comes from the case studies, pilot projects and the actual feedback from the user community [17]. The average size of an industrial project is 500 KLOC. The largest project known to us has over 12 thousand source files, approx. 5 MLOC The initial top-level structure of the application is usually too large and cumbersome, and often has a shocking effect even on the architects of the organization. Usually, for a reasonably well-structured system, the initial excavation takes 2-3 weeks. For unmanaged and highly eroded systems one often needs to break down the available packaging and regroup individual files. Often, Klocwork tools are used to manage a complete software product line, involving multiple applications that share the common software assets.

We believe that summary models that support unlimited aggregation, trimming and links to the source code allow creating architecturally significant models of existing software that can be used for communicating purposes, managing code, analyzing the current status of the software, planning clean-ups and enforcing architecture integrity.

References

1. IDC, *Application Design and Construction tools market forecast and analysis, 2000-2004*, May 2000
2. D. Amyot, G. Mussbacher, N. Mansurov, “Understanding existing software with Use Case Maps Scenarios”, in *Proc. SDL and MSC’ 02 Workshop*, 2002, Aberystwyth, UK
3. OMG, *Unified Modeling Language Specification (UML)*, version 1.4, 2001.
4. C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999
5. N. Rajala, D. Campara, N. Mansurov, inSight reverse engineering CASE tool, in *Proc. of the ICSE’99*, Los Angeles, USA, 1998
6. N. Mansurov, D. Campara, Using Message Sequence Charts to Accelerate Maintenance of Existing Systems, in *Proc. 10th SDL Forum*, Copenhagen, Denmark, June 2001, R. Reed, J. Reed (Eds), Springer Verlag, pp.19-37
7. H.A. Müller, J.-H. Jahnke, D.B. Smith, M.A. Storey, and K. Wong. “Reverse Engineering: a roadmap”. In A. Finkelstein (ed.) *Future of Software Engineering. 22nd Intl. Conf. On Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 47-60
8. R. Holt, Software Architecture Abstraction and Aggregation as Algebraic Manipulation, in Proc. *CASCON*, 1999
9. L. Feijs, R. Krikhaar, and R. van Ommering, A Relational Approach to Software Architecture Analysis, *Software Practice and Experience*, 28 (4):371-400, April, 1998
10. G. Murphy, D. Notkin, K. Sullivan, Software Reflexion Models: Bridging the Gap between Design and Implementation, *IEEE Trans. Software Eng.*, vol 27, 4, April 2001, pp 364-380
11. R. Kazman, S.J. Carriere, View Extraction and View Fusion in Architectural Understanding, in *Proc. of the 5th Int. Conf. On Software Reuse*, 1998
12. R. Holt, Browsing and Searching Software Architecture, in *Proc. Int. Conf. On Software Maintenance*, 1999, Oxford, England
13. A Yeh, D. Harris, M. Chase, Manipulated Recovered Software Architecture Views, in *Proc. Int. Conf. On Software Engineering*, Boston, 1997
14. D. Jerding, S. Rugaber, Using Visualization for Architectural Localization and Extraction, in *Proc. 4th Working Conf. On Reverse Engineering*, 1997, Amsterdam
15. R. Kazman, G. Abowd, L. Bass, and P. Clemens, Scenario Based Analysis of software architecture, *IEEE Software*, pp 47-55, November, 1996
16. N. Mansurov, R. Probert, Improving Time-to-Market using SDL tools and techniques, *Computer Networks*, 35 (2001), pp. 667-691
17. Web page of Klocwork, Inc. http://www.klocwork.com
18. http://www.omg.org/mda