JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2000; **12**:143–170

Research

Maintaining a legacy: towards support at the architectural level

Reinder J. Bril[1,*,†], Loe M. G. Feijs[2], André Glas[3], René L. Krikhaar[4] and M. (Thijs) R. M. Winter[5]

[1]*Philips Research Laboratories Eindhoven (PRLE), Prof. Holstlaan 4 (WL-01), 5656 AA Eindhoven, The Netherlands*
[2]*Eindhoven University of Technology (TUE), P.O. Box 513, 5600 MB Eindhoven, The Netherlands*
[3]*National Aerospace Laboratory (NLR), Voorsterweg 31 (20.1.06), 8316 PR Marknesse, The Netherlands*
[4]*Philips Medical Systems (PMS), P.O. Box 10000 (QR-1139), NL 5680 DA Best, The Netherlands*
[5]*Philips Business Communications (PBC), Anthony Fokkerlaan 5 (KOC-1), 1200 JD Hilversum, The Netherlands*

SUMMARY

An organization that develops large, software intensive systems with a long lifetime will encounter major changes in the market requirements, the software development environment, including its platform, and the target platform. In order to meet the challenges associated with these changes, software development has to undergo major changes as well. Especially when these systems are successful, and hence become an asset, particular care shall be taken to maintain this legacy; large systems with a long lifetime tend to become very complex and difficult to understand. Software architecture plays a vital role in the development of large software systems. For the purpose of maintenance, an up-to-date explicit description of the software architecture of a system supports understanding and comprehension of it, amongst other things. However, many large complex systems do not have an up-to-date documented software architecture. Particularly in cases where these systems have a long lifetime, the (natural) turnover of personnel will make it very likely that many employees contributing to previous generations of the system are no longer available. A need to 'recover' the software architecture of the system may become prevalent, facilitating the understanding of the system, providing ways to improve its maintainability and quality and to control architectural changes. This paper gives an overview of an on-going effort to improve the maintainability and quality of a legacy system, and describes the recent introduction of support at the architectural level for program understanding and complexity control. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: architectural views; legacy systems; program understanding; complexity control; software architecture; reverse architecting; visualization

*Correspondence to: Reinder J. Bril, Philips Research Laboratories Eindhoven (PRLE), Prof. Holstlaan 4 (WL-01), 5656 AA Eindhoven, The Netherlands.
†E-mail: Reinder.Bril@Philips.com

Received 26 June 1999
Revised 24 February 2000

1. INTRODUCTION

Philips is an electronics company that operates worldwide and develops (high volume) consumer systems, e.g. audio and televisions, as well as (low volume) professional systems, such as medical systems and business communications systems. All these systems are becoming more and more software intensive. Whereas hardware development dominated a product's development lead time pre-eminently a decade ago, software development is gradually taking over this role due to a continuous growth of the software contents. This growth of the software has two main causes. Firstly, the ever increasing power and decreasing cost of micro-electronics makes it possible to add significant new functionality without increasing the manufacturing costs. Secondly, some of the desirable new functions can only be realized in software, like an intuitive easy-to-use user interface and the integration of stand-alone products into a system. The growth of the software contents in combination with the long lifetime of a system implies that the stability and extensibility of the software in general and its architecture in particular are determining the business success of the system.

Business communications systems, and especially the private branch exchanges (PBXs; i.e. telephony systems in a business environment), are examples of large complex systems with a long lifetime. These days, the economic lifetime of a delivered PBX is approximately 15 years. Software development in the digital PBX area within Philips spans almost two decades and the maintenance obligations for installed software packages may last over 10 years. The size and complexity of these systems combined with their long lifetime has two main consequences.

Firstly, subsequent generations of systems are developed in an evolutionary way. Hence, development may be viewed as *maintenance* and *all* categories of maintenance (see, for example, [1]) apply for these systems, not only the more common *corrective*, *adaptive* and *perfective* categories (which are explicitly covered by the definition of maintenance in [2]), but also the less common *preventive* maintenance category. For evolutionary development, program understanding is one of the major subjects to be addressed in order to enable these systems to be developed efficiently and within a short lead time, while preserving their quality.

Secondly, the software architecture of these systems is not only important, but its significance is also growing rapidly due to new and changing market requirements, amongst others. The growing importance of the software architecture of systems requires it to be sufficiently documented, clearly communicated, well understood and explicitly controlled. New developers should learn the architecture rapidly so as to become cost effective as fast as possible. They need to understand and comprehend the system's architecture in order to be able to maintain the system. Experienced developers as well, however, feel the need for automated support to browse, analyze and comprehend the system, due to the system's size and complexity.

The quest for program understanding combined with the need to manage the software architecture leads to the need for support at the architectural level.

Whereas software design received a great deal of attention in the early 1970s, software architecture is still an emerging discipline [3,4]. Although special issues of journals have addressed the subject (e.g. [5,6]), books are gradually appearing on the market (e.g. [7]) and research has been reporting successful experiments in extracting the software architecture from a complex software system's implementation for at least ten years [8,9], commercial off-the-shelf (COTS) tools for either forward or backward (i.e. *reverse*) software architecting are still scarcely available, if at all. Given the current state of affairs of software architecture and the theoretical foundations laid and experience gained by Philips

Research [10–17], it was decided to develop a basic set of proprietary architecture comprehension tools for SOPHO,[‡] termed URSA,[§] alleviating the pressing needs of software development. URSA is dedicated to support program understanding and complexity control at the *architectural* level, leaving support at the *programming* level to standard tools that are readily available on the market. URSA is an integral part of the software development environment, available to each individual developer and multiple developers simultaneously, and provides information for every build package, when deemed appropriate.

This paper gives an overview of an on-going effort to improve the maintainability and the quality of a family of PBXs developed and maintained within Philips Business Communications (PBC), and describes the recent introduction of support at the architectural level for program understanding and complexity control. This paper is structured as follows. In Section 2 an overview of the application domain is given, with examples of activities belonging to each of the maintenance categories. In Section 3, URSA is put into perspective. It is shown how the development of URSA is embedded in an overall maintainability and quality 'improvement' program. Although improvements in the software process are typically essential in order to be able to perform other improvements [18], process improvement falls outside the scope of this paper. The support provided by URSA is the topic of Section 4. The experience gained with the development of URSA and its introduction so far are addressed in Section 5. The concluding remarks are given in Section 6.

2. APPLICATION DOMAIN

2.1. Introduction

An organization that develops large, software-intensive systems with a long lifetime will encounter major changes in the software development environment and in the market requirements. In order to meet the challenges associated with these changes, software development has to undergo major changes as well. Examples of environmental and market changes for the range of PBXs developed and maintained by PBC are given in Section 2.2 and Section 2.3, respectively. These examples are related to activities belonging to the various maintenance categories. A characterization of the range of PBXs is given in Section 2.4.

2.2. Changing environment

In the early 1980s, the development of a new range of digital telephony systems for the business market was started at Philips: SOPHO. At that time, it was not only customary but also ineluctable to build your own hardware, embedded real time software (including an application domain specific programming language and a proprietary operating system) and life cycle and management tools supporting large

[‡]SOPHO may be viewed as a family of PBXs developed by Philips Business Communications.
[§]URSA is an acronym for Understanding and Recovery of the Sopho Architecture.

software developments. Times are changing, however, and in order to meet today's challenges, our software development had to undergo major changes, as exemplified below.

- *Software Development Environment (SDE):* Software development spanning almost 2 decades will encounter changes in the (host) development environment (including the platform). Whereas the initial SDE contained a suite of dedicated proprietary tools, the current SDE is to a large extent (and increasingly) based on commercial tools.
- *Operating system:* Development started with a proprietary operating system, supporting a quite sophisticated process model for those days, written in assembly. The process model of today is still basically the same, although it has been enhanced considerably. This model requires a dedicated layer (written in C++), which resides on top of a commercial real-time operating system (RTOS).
- *Programming language:* Development started with a proprietary language derived from (i.e. an extended subset of) CHILL,[¶] a language dedicated to the telecommunications domain defined and recommended by the CCITT[‖] [19]. Use of a proprietary language implied the construction and maintenance of a set of language support tools. Today the application is written entirely in C++, giving all the advantages of a standard language such as commercial off-the-shelf tooling and short learn-in times of (new and hired) staff.

From a software development point of view, the changes in the SDE may be considered as *adaptive* maintenance, necessitated by the changing environment. Similarly, the changes in both the operating system and the programming language may be considered as *preventive* maintenance, improving future maintainability and reliability, and providing a better basis for future enhancements. Though said to be 'rare in the software world' (see, for example [1]), we are convinced that our legacy system is still successful due, amongst other things, to preventive maintenance.

One may also look at these changes from a business point of view rather than a software development point of view. In that case, the environmental changes enabled a strategic move; standard components are used where this is deemed appropriate, allowing a healthy focus on the core business of the company which lies in the telecommunications domain.

2.3. Changing market requirements

Today the economic lifetime of a delivered PBX is approximately 15 years and the *corrective* maintenance obligations (solving reported bugs from the field) for installed packages may last over 10 years. The new and changing market requirements give rise to *perfective* maintenance. Until recently PBXs were benchmarked on the basis of call-related features (like automatic ring-back, call-forwarding-always, etc.), and PBX suppliers competed by providing a wealth of features, amongst others. Notably, the market for PBXs is undergoing radical change. Apart from typical developments in the telecommunications domain, like support for PVN (Private Virtual Network) and cordless

[¶]CHILL is an acronym for CCITT High Level Language.

[‖]CCITT is an acronym for Comité Consultatif International Télégraphique et Téléphonique (i.e. International Telegraph and Telephone Consultative Committee). The CCITT is one of four permanent bodies of the ITU, where ITU is an acronym for International Telecommunication Union.

telephony (e.g. by means of DECT, Digital European Cordless Telephony), one may currently witness the integration of the telecommunications and information technology (IT) domain, as exemplified by CTI (Computer Telephony Integration) and IP-telephony (telephony via internet).

2.4. Characterization of SOPHO

SOPHO may be viewed as a family of PBXs, where the number of (telephone) lines may range from one hundred in a small single system to one million in a fully integrated network (FIN) of systems. Each of the members of the family effectively provides the same functionality, regardless of the size of the system. The possibility to scale the system combined with the wealth of features as typically provided by PBXs [20] gives rise to a high *intrinsic* complexity.

SOPHO is a large software-intensive system; the core of the switch consists of approximately:

- 5000 files, containing 2.5 MLOC written in C++;
- 8000 architectural entities (i.e. subsystems, components, modules and files), organized in an unbalanced tree (representing the decomposition structure of the system) with a depth ranging from 5 to 12;
- 35 000 includes between files; and
- 150–250 static processes and up to 3000 simultaneously executing dynamic processes.

Telephone calls are represented by means of (dynamic) processes in the system. The distinguishing mark of these call processes is that they are all ‘alike’, and mainly consume processing time at the start-up and hang-up of a call. Note that contemporary operating systems are still pushed to their limits when they have to handle such large numbers of calls as separate tasks or processes.

3. URSA IN PERSPECTIVE

3.1. Introduction

The development of URSA is embedded in an overall maintainability and quality improvement program within PBC. In this section, the relevant activities preceding the development of URSA are described in brief, followed by a motivation for the development of URSA, and concluded by a characterization of URSA.

3.2. Product specific course

Program understanding is said to be one of the greatest costs of software maintenance [21,22], taking more than 50% of the effort. Experience within software development for the telecommunications domain within Philips showed a similar figure. In order to reduce the apparent difference between the *perceived* complexity (due to a lack of understanding) and the *intrinsic* complexity of the system, a product specific course was developed. The course covers the software architecture, the related design rules and problem isolation techniques. The software architecture is illustrated by using handmade diagrams and typical scenarios extracted from a running system and is visualized by means of proprietary tools based on interworkings [23]. Explanation of the use of these extraction and

visualization means for program understanding and problem solving is part of the course. All people involved in software development have attended the course (lasting approximately 50 hours) and new staff will follow the course as part of their learn-in period.

3.3. Quality assurance tools

Although a course may minimize the apparent distinction between the perceived and the intrinsic complexity of the system, education alone is clearly insufficient. In particular, it neither prevents the increase of the intrinsic complexity of the system nor guarantees that the complexity stabilizes, let alone that the complexity will be reduced. An appropriate set of metrics may be used to gauge the maintainability and quality of the system [24]. The conversion of the code from the CHILL-derivative to C++ allowed the application of 'standard' quality assurance tools at the *programming* level (such as QAC^{TM}; see [25]). It is not self-evident, however, which sets of metrics and automated support to use at the *architectural* level for our specific application.

In order to be accepted, architectural metrics shall match a developer's subjective judgement of modules. Some specific modules of the system were found to be relatively complex. An examination of the history log of corrective maintenance efforts revealed that these modules were also the most fault-prone modules. A preliminary investigation (inspired by Henry *et al.* [26]) yielded a satisfactory model that identified the most fault-prone modules. This model is based on the length, the fan-in (in terms of received signals and number of procedures used by other modules) and the fan-out (in terms of signals sent and number of procedures used by other modules) of a module. This model matches a developer's subjective judgement of modules and may provide appropriate architectural metrics. Further study in this area is needed, however.

3.4. Software architecture recovery and maintenance experiment

In the early 1990s, it was observed that the documentation of the software architecture of SOPHO was not up-to-date, like many large software systems of a considerable age [27]. Although the description of the software architecture can be recovered (as demonstrated by, for example, Müller *et al.* [8]), recovery without its subsequent maintenance does not provide a structural solution. A small experiment was therefore conducted to explicitly *control* the software architecture during a perfective maintenance activity. The activity concerned the creation of support for CTI by means of CSTA (Computer Supported Telecommunications Applications; see [28,29]), involving the development of a new component of approximately 10 KLOCs and extension of about 10 components. The control of the software architecture was an additional constraint on that activity. The *intended* software architecture (which has a layered structure) was defined during an initial stage of the project, guided the implementation, and was subsequently compared with the *extracted* (or *as built*) software architecture. The differences between the intended and extracted architecture were either resolved (by adapting the intended architecture and/or the implementation) or explicitly documented as optimizations performed at the (detailed) design or implementation level. Note that quality control at the architectural level is introduced by using the intended software architecture as a standard against which the extracted software architecture is placed. Visualization of the software architecture was done by using a collection of small prototype extraction and abstraction tools, mostly shell scripts, and an existing, experimental graph visualizer called 'TEDDY' as a means for presentation (TEDDY is a box-arrow visualizer developed within Philips

Research similar to Rigi, see [8]). The same set of tools was used to *reverse* architect other components, visualizing their (extracted) software architecture. Notably, the visualization of the architecture of the component containing the most fault-prone modules revealed a relatively high deteriorated structure.

The experiment showed that it is possible to recover the software architecture, and to control the software architecture during perfective maintenance, preventing deterioration. The results of the experiment were favorably received within the organization, both from a program understanding as well as a complexity control point of view. The prototypical nature and lacking functionality of the set of tools used, the absence of COTS tools, and the estimated cost of developing a basic set of tools prevented the incorporation of appropriate means in the development environment, however. The experiment was concluded early 1995.

3.5. Motivation of URSA

The visualization of the scenarios extracted from a running system by means of the proprietary tools based on interworkings, as described in Section 3.2, aids program understanding considerably. The gap between the scenarios on the one hand, and the actual code on the other, was felt to be very large, however. Means that bridge this gap were felt to be a desirable improvement. Following this, the organization urged the use of architectural support similar to that offered by the prototype tooling used in the small experiment described in Section 3.4. Due to the size of the system (see Section 2.4), tooling is a necessary pre-requisite for efficient control of its software architecture. The set of tools used for the experiment were prototypes, and lacked desirable functionality. In particular, the tools should support the visualization of arbitrary views of the system (rather than predetermined views). This requires a systematic (rather than a seemingly ad hoc) approach for the extraction and abstraction tools, especially because of the size of the system and the fact that the decomposition structure of the system is an *unbalanced* tree. Furthermore, two main additional functionalities were desired from a visualizer. Firstly, the automatic layout of graphs, because graphical formatting turned out to be very laborious. Secondly, browsing (e.g. expanding boxes and/or arrows), in order to be able to investigate the architecture and find the cause of unexpected and/or undesired dependencies between architectural entities. Moreover, the experiment showed the occasional need for browsing up to entities at the programming level. Finally, it was considered desirable to have additional means for quality control at the architectural level, next to visual inspection.

Due to a lack of COTS tools that support architectural comprehension, and given the theoretical foundations laid and experience gained by Philips Research since the experiment described in Section 3.4 [12–15,17,30], it was decided in early 1997 to develop a basic set of proprietary tools for SOPHO, termed URSA.

The development of URSA may come as a surprise, considering the desired focus on core business, as described in Section 2.2. As in the situation in the early 1980s, the means required are not available at present, and the availability of those means is considered indispensable for the maintenance of our legacy. Moreover, URSA is not developed by PBC in isolation, but as a joint effort between PBC and Philips Research. Hence, although URSA has been developed for SOPHO, its application is not restricted to the telecommunications domain; instantiations of the generic parts of each of the basic tools are also applicable and used to support software developments of consumer and other professional systems within Philips.

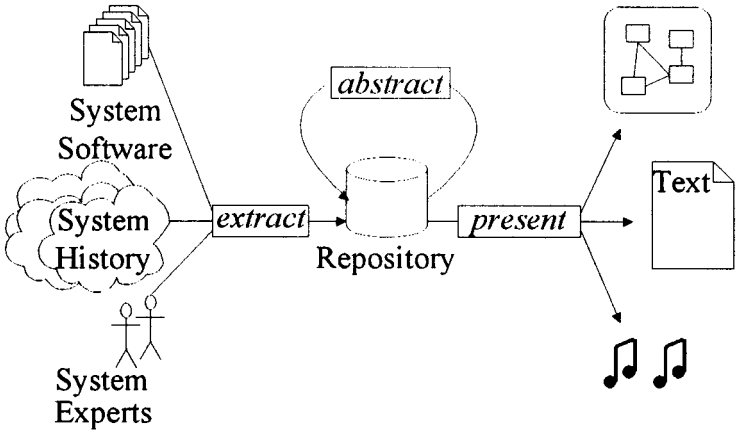

Figure 1. 'Extract–abstract–present' paradigm.

3.6. Characterization of URSA

URSA is based on the 'extract–abstract–present' paradigm (inspired by [8]; see Figure 1), which is basically the same as the 'repository-based-reverse-engineering' approach as recommended in [31] for the reverse-engineering of large legacy systems. The three activities shown in Figure 1 are characterized briefly as [32]:

- *extract*: extracting relevant information from system software, system history and system experts;
- *abstract*: abstracting extracted information to higher (design) level; and
- *present*: presenting abstracted information in a developer-friendly way, taking into account his or her current topic of interest.

In order to be able to describe the tools, the following definition of software architecture** is used: the software architecture of a system comprises software *components*, *connections* (or relationships) between those components, and *constraints* on the connections. A software architecture may be described using different concurrent views (see [33] for an empirical or [34] for a synthesized model), where each view addresses a specific set of concerns. The '4 + 1 view model' described in [34] distinguishes the following views (see also Figure 2, which has been taken from [34]):

- the *logical view*, describing the services the system provides to its end users;
- the *process view*, describing the system's concurrency and synchronization aspects;
- the *development view*, describing the system's static organization;

**There is no generally accepted definition of software architecture. According to [7], this definition lacks the notion of externally visible properties. Nevertheless, the definition serves our purposes for this document.

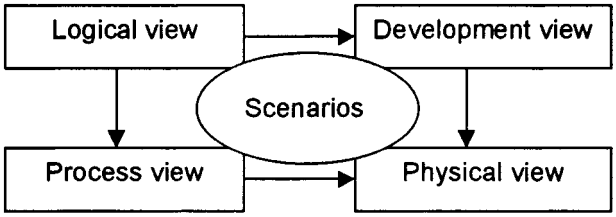

Figure 2. The 4 + 1 view model.

- the *physical view*, describing the mapping of the software onto the hardware, reflecting its distributed aspect; and
- *scenarios*, showing how the other four views work together.

Note that in the definition of software architecture given above the interpretation of the *components*, *connections* and *constraints* depends on the particular view. For example, for the development view, the components are the modules, the connections are the uses and part-of relations between modules [17], and the constraints are the restrictions imposed on these relations.

Using the terminology of [34], URSA contains the following tools:

- *Jolly Jumper*: a Clickable MSC (Message Sequence Chart) Viewer, visualizing a scenario in terms of actors and their communication, and providing (hyper-)links to the source code. The Jolly Jumper links *scenarios* with the *process view* and *development view* [35];
- *MAB*: a Module Architecture Browser, visualizing the structure of the system (expressed in terms of a part-of and uses relation) by means of tables. The MAB supports the *development view* [36]. The term 'MAB' is inspired by the term 'Module (Interconnection) Architecture' of [33]. The notion *module architecture* may be considered to be part of the *development view* [32]. The distinction between module architecture and development view is not relevant for this document, however; and
- *ArchiSpy*: a module architecture verifier, determining whether or not the software conforms to a set of architectural rules (or constraints) for the *development view*, and reporting breaches. ArchiSpy is basically a quality assurance tool at the architectural level, and is based on a relational calculator [12].

4. SUPPORT PROVIDED BY URSA

4.1. Introduction

In [32] a software architecture improvement approach is described that is termed 'Software Architecture Reconstruction'. This approach provides the foundation for URSA from a process perspective. The approach covers both forward and backward (or *reverse*) engineering in the software development cycle and has been proven to be applicable for various complex systems in different domains within Philips (see also [13]). Parts of the approach have also been evaluated by independents

outside Philips [37]). The approach distinguishes different levels of reconstruction (which have been inspired by the levels in Humphrey [18]). Systems that are barely documented and whose software architecture is implicit are at the *initial* level of reconstruction. By making the software architecture of such systems explicit (by *reverse architecting* the system), the *described* level is reached. Improvements of the architecture are necessary when the gap between the described architecture and the ideal architecture is too large. By *re-architecting* the system, the *redefined* level is reached. By means of *architectural verification*, the software architecture becomes explicitly controlled during the evolution of the system and the *managed* level is reached. The final level, termed *optimized*, may be reached by taking future extensions into account.

URSA provides means that support the *described* and *managed* level.

- *Described level*: The Jolly Jumper and the MAB *visualize* architectural information, making the software architecture of a system explicit using *reverse architecting* information. Visualization is typically part of an architectural analysis activity, either implicit or explicit. Whereas the Jolly Jumper is primarily meant for program understanding, the MAB is used for both program understanding and (qualitative) complexity control.
- *Managed level*: ArchiSpy *checks* compliance of a system to a defined set of architectural rules, supporting the *architectural verification* of a software architecture. ArchiSpy is exclusively used for (quantitative) complexity control. ArchiSpy currently supports an initial set of approximately 15 rules. The intention is for it to support a set of metrics at the architectural level in due course.

URSA provides no means for support at the *redefined* level (i.e. for re-architecting a system). Re-architecting will not be dealt with in this paper either. Note that the software bookshelf [38], which is meant to capture, organize, and manage information about a legacy system, only supports the *described* level for the module architecture view.

Reverse architecting approaches reported upon in the literature are typically applied in the context of the module architecture [8,27,39,40]. The MAB fits in with this tradition, providing basically the same support for similar purposes. The distinguishing mark of the MAB forms its theoretical foundations and the quantitative information it provides along with the connections between components. Although there exist many tools that visualize the dynamic behavior of systems [41], none compare with the Jolly Jumper with respect to its ability to link scenarios with code.

Relation algebra with multi-relations [14,15,32] provides the theoretical foundation for both the MAB and ArchiSpy. The relational approach supports the analysis of software architectures and provides options for visualization, view calculations and software architecture verification (i.e. checking the conformance of an existing architecture to a set of architectural rules). Although the underlying machinery is quite different, our approach in the area of the module architecture is basically the same as those reported upon in [9,42].

The Jolly Jumper and MAB are considered in more detail in Section 4.2 and Section 4.3, respectively.

4.2. Bridging the gap between scenarios and code

4.2.1. Motivation

Most maintenance activities start with a request for enhancing or changing the dynamic behavior of (small parts of) the system. Although our product-specific course gives an introduction to this dynamic

behavior in terms of the relevant software architectural artifacts and some typical scenarios, the size and the complexity of the system are such that for most maintenance requests, on its own, the information provided is far from sufficient for the developers.

Instead of the (global) architectural principles and the typical scenarios, detailed understanding of the existing dynamic behavior including knowledge of the corresponding 'responsible' software is a prerequisite for the successful realization of the requested enhancement or change. The existing proprietary tools, which are based on interworkings [23], visualize scenarios extracted from a running system in MSC (Message Sequence Chart) like diagrams. These diagrams relate the external behavior of the system with internal actions performed by the system. The most important lack of information, however, is still the link to the actual software parts that correspond with the visualized dynamic behavior. In a large system with many interacting features, which is what our legacy system currently is, people lacking the in-depth knowledge of the feature under maintenance cannot easily find this link. To be able to spread the maintenance work and also to assign maintenance work to less experienced developers or developers who are experienced in other parts of the system, extra tooling is required that assists the developer in finding the links between visualized dynamic behavior and code.

Let us take a closer look at the process model in order to understand why the link to the actual software parts that correspond with the visualized dynamic behavior is needed. The diagrams depict all the *processes* involved and their communication behavior related to one or more specific features. The diagrams are based on the ITU standard for MSCs [43]. MSCs are a standard for the description of the communication behavior between concurrent processes. MSCs are an addition to SDL (Specification and Description Language) [44], the ITU standard for the specification and description of the behavior of telecommunications systems. Similar to SDL [45], the proprietary process model for the legacy system also allows for signals to be sent from and received in (separate states in) *procedures*. The sending and receiving may occur in a procedure at an arbitrary level of nested procedure calls. Because the nested procedure calls are not shown in MSCs, the link between the signals exchanged between the processes shown in the diagram and the corresponding code is not always obvious. The same holds for other internal actions performed by the system and visualized in the diagrams.

No tooling is currently available that supports the linking of MSC 'actions' to code fragments. If any link to code were to be supported, the link to SDL designs (either graphical or textual representation) would be the most natural to expect. Although the proprietary process model of the legacy system is similar to the process model of SDL in many respects, no one-to-one mapping between SDL and the proprietary process model is possible (the details fall outside the scope of this paper, however). Therefore, within the context of URSA, specialized tooling has been developed for the above purposes. The extended MSC language CMSC (Clickable MSC) plays a special role in this tooling.

4.2.2. The tooling exemplified

Because the details of the system are proprietary, only a general description of the functionality of the system is given, using a simplified example of a 'basic call' for illustration purposes. Technical details of the tooling fall outside the scope of this paper; see [35], covering the technical details in depth.

Initially, the system starts a number of service processes and a generic call process. The generic call process is responsible for creating a dynamic process for each telephone call. A call process consists of a number of cooperating tightly coupled sub-processes, where extension (i.e. telephone set) specific

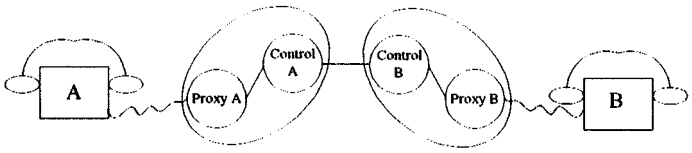

Figure 3. Basic call.

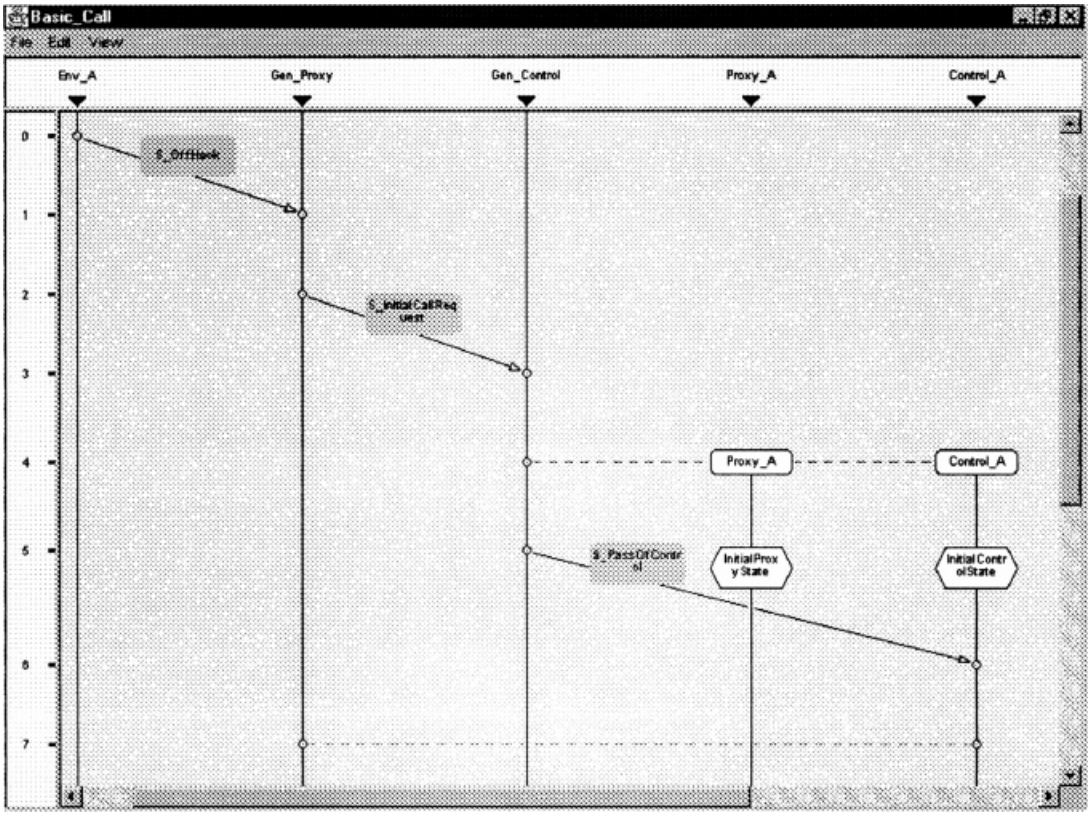

Figure 4. Jolly Jumper.

information is encapsulated in a proxy and generic functionality is dealt with in a general controller; see Figure 3.

The dynamic behavior of the system is visualized with the Jolly Jumper by means of a 'CMSC' (Clickable MSC) diagram. Figure 4 shows a small fragment of such a diagram that corresponds with a call set-up. The ruler at the top contains the identifications of the (sub-) processes involved in the

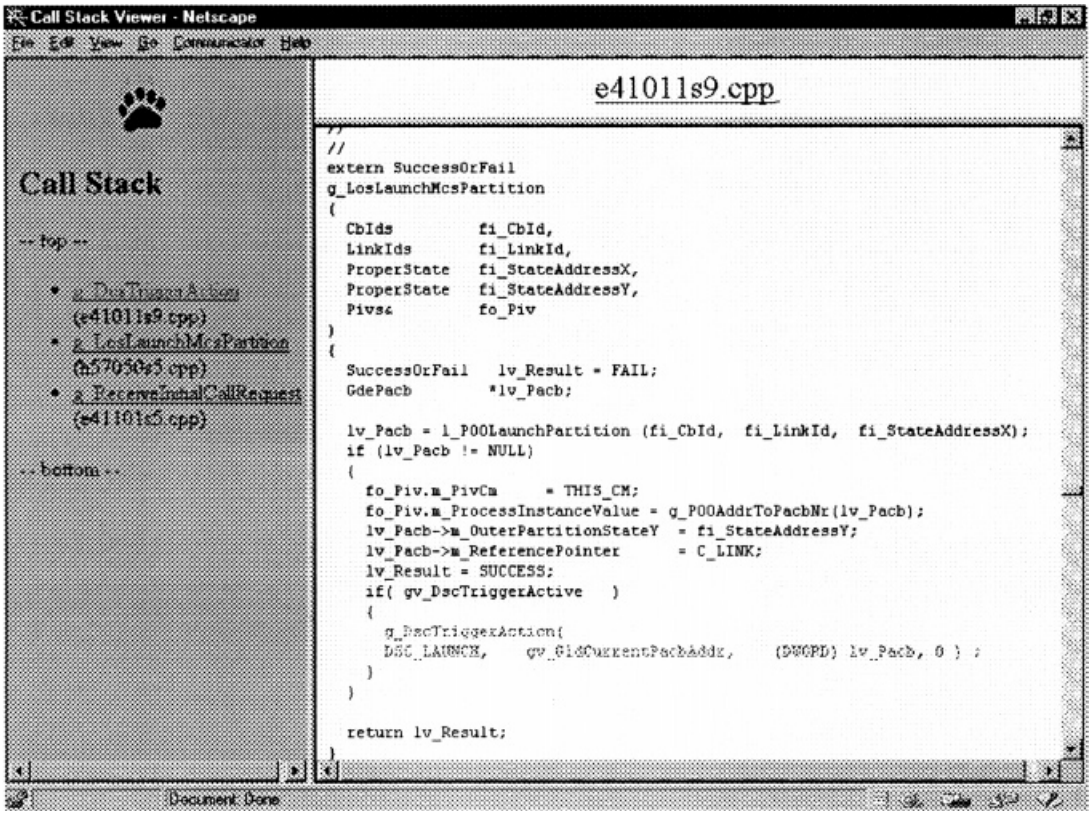

Figure 5. Source code viewer.

scenario. The ruler at the left contains the time stamps at which the various actions occur. In Figure 4, these time stamps have been replaced with successive numbers.

Let us consider the fragment in the scrollable window in more detail. When an extension (say A) goes off-hook, the environment (‘Env_A’) sends an ‘S_OffHook’ signal to the generic call process. The signal is received in the (generic) proxy (‘Gen_Proxy’) of the generic call process. The proxy sends an ‘S_InitialCallRequest’ signal to the (generic) controller (‘Gen_Control’). Upon receipt of the signal, the controller process creates a new sub-process, consisting of a proxy (‘Proxy_A’) for this particular kind of extension and a general controller (‘Control_A’), and sends an ‘S_PassOfControl’ signal to the controller of the newly created sub-process. Note that the dashed line, that corresponds with the creation of the sub-process, represents *synchronization*, a feature that is not available in MSCs. The proxy and controller of the newly created sub-process have initial states ‘InitialProxyState’ and ‘InitialControlState’, respectively. Upon receipt of the ‘S_PassOfControl’ signal, the newly created sub-

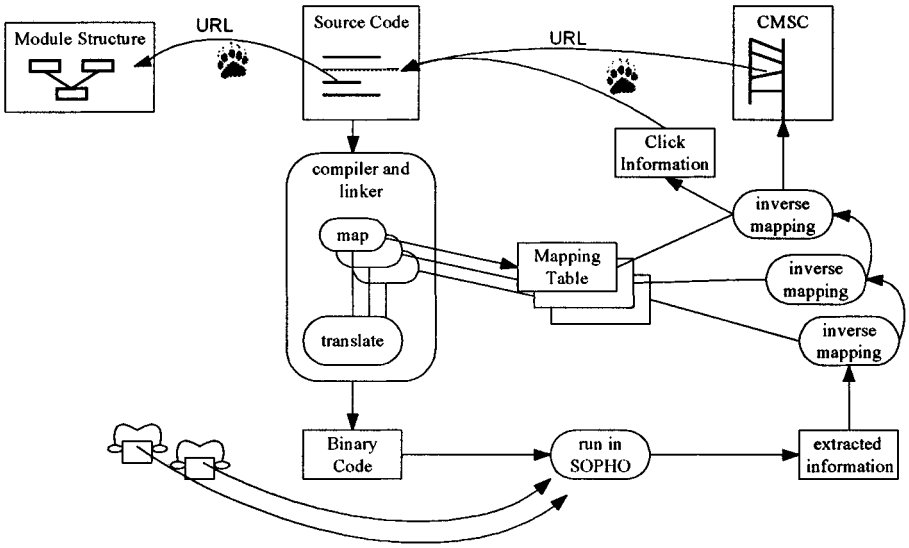

Figure 6. Connection between tools and related artifacts.

process separates itself (and its related proxy) from the generic call process and becomes independent. The dashed line from 'Gen_Proxy' to 'Control_A' corresponds with the separation.

CMSC is a graphical language, in the tradition of MSC [43]. The most distinguishing feature of CMSC is that nodes, edges (vertical lines connecting nodes) and messages (represented by labeled arrows in Figure 4) have an associated URL (Universal Resource Locator). By clicking the URL, 'jumps' to the source code can be made and other relevant information can be visualized (hyper linking) via your own Web browser.

A 'source code viewer' presents the source code upon jumping (see Figure 5, which shows the situation that corresponds to the creation of the sub-process). The source code viewer highlights the corresponding code and presents the stack of nested procedure calls that dynamically embed the code. Each of the procedures in this stack may be clicked, causing the frame containing the source code to be updated. The frame above the source code contains the file name with a hyperlink to the 'MAB', the module architecture browser, which is described in the next section.

The connection between the various tools and the related artifacts is shown in Figure 6. Let us consider the source code as the starting point in this figure. In order to be able to extract information from a running system, extra functionality that provides the means to create a logging of the internal actions performed by the system has been added to (i.e. *instrumented* [46]) the source code. This source code resides in the dedicated layer supporting our proprietary process model on top of the commercial operating system (see Section 2.2). By running a particular scenario on the system involving one or more features, a logging is created. This low-level logging, which contains hexadecimal codes only, is translated in a CMSC using 'mapping tables' generated by the compiler to allow for symbolic (rather

than hexadecimal) identifications of processes, signals, etc. The Jolly Jumper is subsequently used as a presentation tool for the CMSC. Clicking may be viewed as 'closing the diagram' by jumping to the corresponding code presented by the source code viewer. Possible click actions are indicated by a bear's claw in Figure 6 (URSA means bear in Latin).

4.3. Module architecture visualization

4.3.1. Motivation

Unless descriptions of the software architecture of a system are maintained, they will become out-of-date due to maintenance activities during the evolution of the system. Just like many large software systems of a considerable age, the description of the software architecture of SOPHO was not up-to-date. Without a well-documented software architecture, large and complex systems are hard to understand and maintain. Recovery of the software architecture is therefore the first step towards improving the understanding of a system at the architectural level. Once the architecture has been recovered (by means of reverse architecting), it may be (optionally) improved (by means of re-architecting), and is preferably controlled to prevent it from becoming out-of-date again. The experiment described in Section 3.4 showed that control of the software architecture is feasible during *perfective* maintenance activities.

The (module) architecture of SOPHO has a layered structure. Structuring architectures by means of layers has a long tradition. The notion of a layered structure and its advantages with respect to ease of development and maintenance have already been described in [47]. The benefits of systems with a layered architecture in terms of development effort and costs have been verified empirically in [48]. A module architecture comprehension tool shall therefore support the systematic investigation of a module architecture and the identification of connections violating the layered structure.

During re-engineering work of a large (medical) software system [15] the usefulness of quantitative information (or 'weights') on the connections between architectural entities was discovered. Given this information, one can distinguish between important and minor or even accidental connections in a module architecture view. Support for weights is therefore a requirement for a module architecture comprehension tool.

Unlike the (specification and) description of the behavior of telecommunications systems by means of SDL [44] and the additional description of communication behavior between concurrent processes by means of MSCs [43], there are no standards for the description of a module architecture. A module architecture is typically drawn as a nested set of boxes (where the nesting represents the decomposition structure of the system and the boxes the architectural entities) and arrows (representing the connections between those entities). Before addressing the functionality provided by the MAB, the description of a module architecture is addressed briefly and the approach taken to extract the module architecture information for SOPHO is described.

4.3.2. Towards the description of a module architecture

Figure 7(a) shows a module architecture view of a (very simple) system S. S consists of two subsystems, A and B, where A consists of a component A_2 (which in turn consists of two modules A_{21} and A_{22}) and two modules A_1 and A_3, and B consists of two modules B_1 and B_2. This decomposition

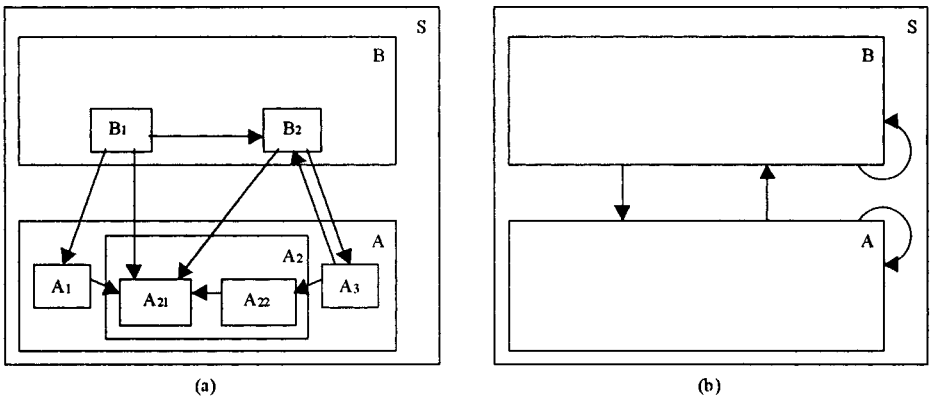

Figure 7. Module architecture views of S.

structure of S is visualized in Figure 7(a) using a 'boxes-in-boxes' representation [49]. Note that the decomposition structure of S is an unbalanced tree. The arrows in Figure 7(a) visualize the uses of the finest grain architectural entities, e.g. B_1 uses A_1.

Using a relational approach, a module architecture view of the system S as shown in Figure 7(a) can be captured by:

(i) a set of architectural *entities* E, representing the Universe of Discourse:

$$\mathsf{E} = \{S, A, A_1, A_2, A_{21}, A_{22}, A_3, B, B_1, B_2\};$$

(ii) a *part-of* relation P on E × E, representing the decomposition structure of S:

$$\mathsf{P} = \{\langle A, S\rangle, \langle A_1, A\rangle, \langle A_2, A\rangle, \langle A_{21}, A_2\rangle, \langle A_{22}, A_2\rangle, \langle A_3, A\rangle, \langle B, S\rangle, \langle B_1, B\rangle, \langle B_2, B\rangle\};$$

(iii) a *uses* relation U on E × E, representing the uses of modules:

$$\mathsf{U} = \{\langle A_1, A_{21}\rangle, \langle A_{22}, A_{21}\rangle, \langle A_3, A_{22}\rangle, \langle A_3, B_2\rangle, \langle B_1, A_1\rangle, \langle B_1, A_{21}\rangle, \langle B_2, A_{21}\rangle, \langle B_2, A_3\rangle, \langle B_1, B_2\rangle\};\ \text{and}$$

(iv) a *layout* relation L, representing the placement of entities in a view. The layout relation is not considered in more detail for this figure.

The part-of relation allows for the use of terms for relatives, such as parent (e.g. A is a parent of A_1, A_2, and A_3), children, siblings and ancestors.

By *hiding* the decomposition structure of both A and B, Figure 7(b) is derived from Figure 7(a). The uses relations from the constituents of B to the constituents of A in Figure 7(a) (e.g. from B_2 to A_3) give rise to a uses relation from B to A in Figure 7(b). Similarly, the uses relation from A_3 (a constituent of A) to B_2 (a constituent of B) in Figure 7(a) gives rise to uses relation from A to B in Figure 7(b). The intra subsystem uses relations in Figure 7(a) (e.g. from B_1 to B_2) give rise to reflexive

uses relations in Figure 7(b) (e.g. from B to itself). The notion of hiding is formalized in [50] using a liberal kind of lifting termed 'oblique lifting'. Oblique lifting has as important advantage (compared to lifting as described in [14]) that it can be used for unbalanced systems. The notion of hiding and its application to visualizing system structure has already been described informally in [9]. Hiding is termed *condensation* in that paper. An informal description of hiding may also be found in [40].

4.3.3. Module architecture extraction

The decomposition structure of SOPHO is an unbalanced tree, with a depth ranging from 5 to 12. The decomposition structure of SOPHO has been stored in the configuration management system, an advantage that can clearly be attributed to the original proprietary configuration management tool. Although the decomposition structure is amenable to improvements, the availability of a well-defined structure simplifies the recovery of the architecture considerably; i.e. the rediscovery of the architecture of legacy systems is stated to be a critical research area [7], and research experience shows that rediscovery of the decomposition structure of a system requires a human's judgement [27].

The part-of relation of SOPHO is derived from the information stored in the configuration management system. The uses relation of SOPHO is extracted from its source code. The set of architectural entities may be derived by calculating the carrying set of the part-of relation (i.e. the union of its domain and range).

4.3.4. Functionality of the MAB

4.3.4.1. Tabular representation. Means for graphical representations are clearly preferred for the visualization of the module architecture. A graphical representation does however cause a major problem in terms of the layout. A fully automatic graph layout will typically yield graphs that will appear unnatural to a system developer, simply because the algorithms lack the information concerning the 'logical' relation between entities. So, although automatic graph layout may provide an initial approximation, manual post-processing is still necessary. Considering the huge amounts of different graphs (in the order of magnitude of millions given the number of architectural entities for SOPHO), where each graph requires its own layout, manual adjustments of these graphs is not feasible within a reasonable period of time.

For a tabular representation, the number of layouts is considerably less (in the order of magnitude of thousands). Next, table layout is a one-dimensional problem, whereas graph layout is a two-dimensional problem. Finally, the overall structure of a table does not change when a few rows and columns are not at their most self-evident position.

4.3.4.2. Weight of a uses relation. In many situations it is helpful to have quantitative information on the uses relations between architectural entities [15]. Considering Figure 7(b), there are various *weights* (or multiplicities) which could be associated with the uses relation from subsystem B to subsystem A; see [32]:

- *existence-oriented* weight: a value of 1, denoting that there *exists* a uses relation between finest grain entities;

- *fan-out-oriented* weight: a value of 2, which is the number of *using* (finest grain) entities (i.e. B_1, B_2);
- *fan-in-oriented* weight: a value of 3, which is the number of *used* (finest grain) entities (i.e. A_1, A_{21}, A_3);
- *size-oriented* weight: a value of 4, which is the number of *uses* from (finest grain) entities of B to those of A (i.e. $\langle B_1, A_1\rangle$, $\langle B_1, A_{21}\rangle$, $\langle B_2, A_{21}\rangle$, $\langle B_2, A_3\rangle$).

The various weights of a uses relation may be calculated for unbalanced systems by means of oblique lifting (see [50] for a formalization based on Feijs *et al.* [15] and Krikhaar [32]).

The use of (the combination of) weights is briefly illustrated by means of an example of a scenario; the replacement of a component (e.g. a proprietary RTOS by a COTS version). Let's assume that files are the smallest grain architectural entities. The size-oriented weight (associated with the uses relation from the applications to the RTOS) indicates how many include statements must be replaced. The fan-out-oriented weight indicates how many files (from the applications) are affected. The fan-in-oriented weight indicates how many files (from the proprietary RTOS) contribute to its used interface, and, hence, which functionality is required from the new version.

4.3.4.3. Presentation and navigation. The basic form in which the MAB displays information is a table, displaying uses relations from one *using* entity to one *used* entity. There is a row for each child of the using entity and a column for each child of the used entity. The cells of the table contain the weight of the uses relation between the two corresponding children, where the specific kind of weight displayed is selectable by a user. Next to the names of the using and used entities, the part-of relations of these entities are displayed, i.e. all ancestors of these two entities up to an artificial *top* ($\top$)[††]; see the table at the left-top of Figure 8 for an example of a table where both the using and used entity are the system S, and the cells contain a size-oriented weight (e.g. subsystem B uses subsystem A four times).

Next to the names of the using and uses entities, their siblings can be displayed. As displaying all siblings of an entity may clutter a table, the number of visible sibling entries is adjustable by a user.

The layout of a table is determined by the order of its rows and columns. For this, each coarse-grain entity has an order relation over all its children. Whenever such an entry acts as a using or used entity in a table, the rows and columns are ordered according to this relation. The user may provide a layout relation. The MAB will generate a default layout relation in cases where no such relation is provided.

Note that the information shown by the MAB is confined to the uses relations from the children of *one* using entity to the children of *one* used entity. Hence the MAB prevents the creation of cluttered module architecture views, which may easily arise when expanding multiple architectural entities. The largest table shown by the MAB for a particular system is the one having the parent with the highest number of children of that system as both using and used entity.

In order to be able to draw conclusions based on a module architecture view, one needs to know what is shown and, more importantly, what is not shown (or hidden). In other words, this means that a *completeness* criterion for the information presented in a view is desired. Such a criterion gives rise to a quality requirement for the MAB. For the MAB, this criterion may be characterized as follows: given a part-of and uses relation, the MAB will display:

[††]The *top* is required in order to show the intra uses relations of a system.

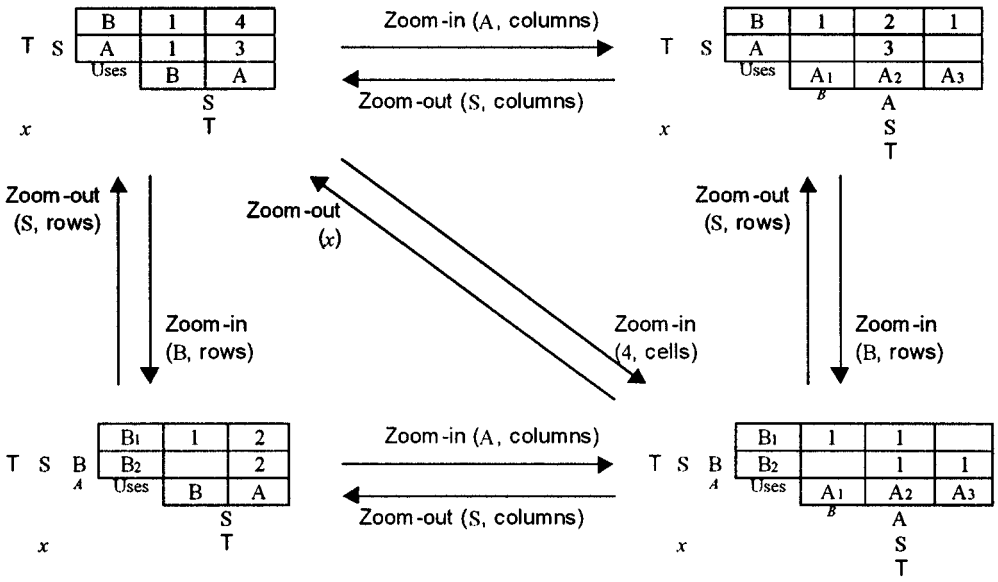

Figure 8. Zoom-in and zoom-out.

- the names of a selected pair of using and used entities;
- all children of both the using and used entity;
- the uses relation from every child of the using entity to every child of the used entity; and
- any ancestor of the using and used entity (including the artificial top T).

The MAB provides, amongst others, the following means to navigate through the tables:

- *zoom-in* and *zoom-out* on rows, columns and cells of the table;
- *exchange* rows and columns of a table;
- list the *compound* of a uses relation (which is the uses relations between the finest grain entities that give rise to this uses relation); and
- *display* the source code.

Consider Figure 8 as an example. The table at the left-top of this figure has the system S as both using and used entity. It is possible to *zoom-in* by 'clicking' on subsystem A or B (in either the row or the column) or on any of the values in the cells of the table (which implies a zoom-in on both its row and column).

Similarly, it is possible to *zoom-out* by clicking on the top T (in the row at the left or the column at the bottom) or on x (which implies a zoom-out of both the rows and columns). Zoom-in and zoom-out are just ways to traverse *vertically* through the decomposition structure. In addition to vertical traversal, it is possible to traverse horizontally through the decomposition structure by clicking on the names of

siblings (when made visible according to a user preference). In Figure 8, the siblings are shown in italics (e.g. *B* is a sibling of A in the column at the bottom of the table at the right-top of that figure).

Given an appropriate requirement for completeness, a graph will present *all* uses relations between entities. The asymmetric nature of a table (in those cases where the using and used entities are different) causes only those uses relations to be shown from the entities in the rows to the entities in the columns. It is therefore possible to *exchange* the rows and columns of a table, yielding the uses relations in the other direction.

The *compound* of a uses relation from an entity E_1 to E_2 is the list of all uses relations from the finest grain entities contained by E_1 to those of E_2. E.g. the compound of the uses relation from B to A yields $\langle B_1, A_1\rangle$, $\langle B_1, A_{21}\rangle$, $\langle B_2, A_{21}\rangle$, and $\langle B_2, A_3\rangle$.

By clicking on a finest grain entity (e.g. B_1 in Figure 8), a window pops up *displaying* its source code.

4.3.4.4. Manipulation. Whereas 'navigation' is related to the 'present' stage of the 'extract-abstract-present' paradigm, 'manipulation' refers to either performing a filter operation or changing the data during the 'abstract'-stage. The MAB provides, amongst others, the means to:

- *strip* empty rows and columns;
- *iconize* rows and columns that are of no interest; and
- manually *change the layout* (i.e. the order of the elements in a row and/or column).

The need for stripping and iconizing is not obvious for small-scale examples, but we found that it becomes pressing when investigating large complex legacy systems, where parents may have many ($\gg 10$) children. Tables that look like sparse matrices become considerably more appealing as a result of the stripping of empty rows and columns. Iconizing all children of all but one yet unidentified parent may simulate lacking abstractions (i.e. lacking intermediate entities within a decomposition structure). Note that care shall be taken to properly represent stripped and iconized rows and columns and to formulate the completeness criterion in order to deal with stripping and iconizing.

4.3.5. Identification of connections violating a layered structure

In [48], it has been verified empirically that systems with layered architectures have benefits in terms of development effort and cost. It will be shown below that the tabular representation of the MAB allows for the verification whether or not a system is (still) layered by means of a simple visual inspection and supports the systematic investigation of breaching uses relations.

In the table at the left-top of Figure 8 (which has S as both using and used entity) the cells on the diagonal from left-top to bottom-right represent the 'intra-subsystem' uses relations. Whenever S was meant to be a layered system, the bottom-left triangle (degenerating to a single cell) of the table should be empty (i.e. A should not use B). The breaching uses relation may be found by clicking that cell.

Figure 9 shows module architecture views of a strictly layered system S', in a graphical and tabular representation (using an existence-oriented weight). The fact that S' is a strictly layered system follows immediately from the table: the bottom-left triangle is empty and the top-right triangle contains only values adjacent to the diagonal.

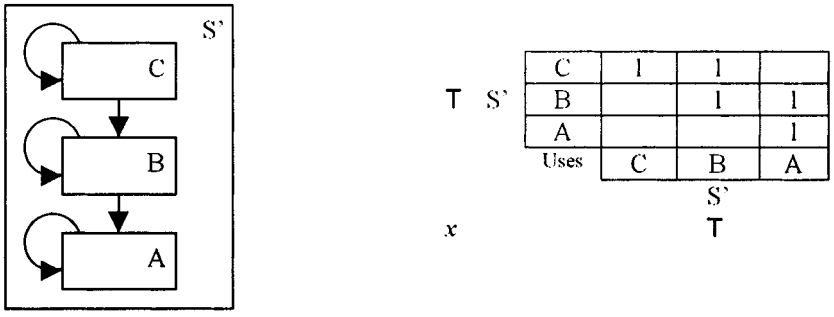

Figure 9. Strictly layered system S'.

4.3.6. User interface of the MAB

Figure 10 shows a view of the system S that corresponds with the bottom-right table of Figure 8 using the MAB. Note that 'World' is used rather than the 'T' for the top.

4.3.7. Design of the MAB

The MAB is client-server based, using state-of-the-art web technologies, such as JavaScript[#] and CGI-scripts (CGI is an abbreviation of Common Gateway Interface). The part-of, uses and layout relations of the various versions of the system are located at the server. A dedicated server process is created for each version of the system upon the first request, and that process will terminate itself when it remains idle for a pre-determined time. At start-up, a server process reads the part-of, uses and layout relations (amongst others), and builds up an internal representation containing all relevant information (including the various weights for *all* uses relations). Although this implies some lead time at start-up, it simplified the design considerably, and provided a short response time upon subsequent requests. Multiple clients inquiring about information relating to a single version of the system communicate with a single server process. Presentation (and navigation) is done at the client side of the MAB. Manipulation is also done at the client side and only involves the server side when the changed layout has to be made persistent. Note that because of the size of SOPHO (3 K coarse-grain architectural entities, giving rise to 9 M tables with an average size of 10 KB, requiring 90 GB disk space for a single version of the system), it is clearly out of the question for these tables to be generated statically for multiple versions.

Although the MAB has primarily been designed to visualize the uses and part-of relations between architectural entities, it is occasionally used for entities at the programming level. One should be very careful with the memory occupation when crossing the architectural/programming boundary, however, because the number of programming entities is typically one to two orders of magnitude larger than the number of architectural entities.

[#] JavaScript and Java are trademarks of Sun Microsystems, Inc.

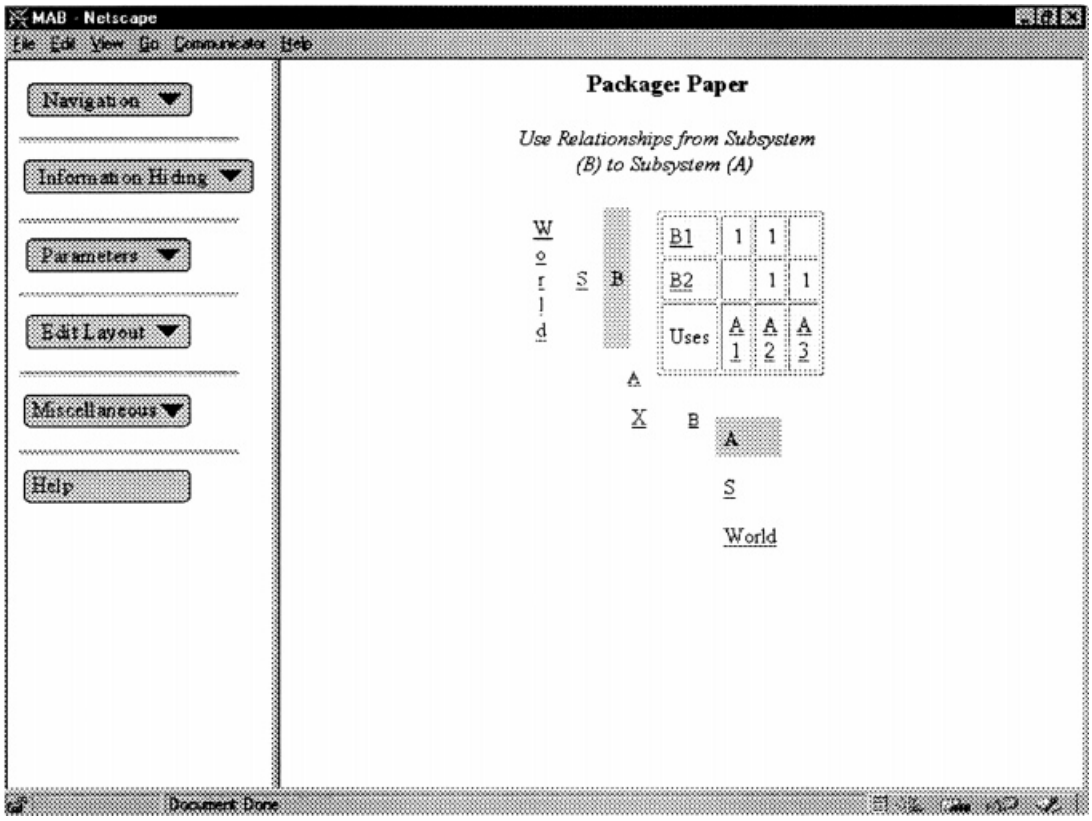

Figure 10. Module Architecture Browser (MAB).

5. EXPERIENCE

5.1. Introduction

URSA is based on matured principles, has a proper theoretical foundation and has been developed using state-of-the-art technology. As a consequence, the provision of support at the architectural level is technically feasible and turned out to be straightforward (but certainly not trivial). In our experience, organizational issues are considerably more demanding than technical issues when developing and subsequently introducing a toolset like URSA.

The development of URSA is described briefly in Section 5.2. The introduction of URSA is described in Section 5.3 in some more detail. The following subjects are covered: the involvement of the end-users, the first experiences with the usage of the program understanding and complexity control support to which URSA is dedicated, and the necessity of change in the development process.

5.2. Development of URSA

URSA has been developed using state-of-the-art technology (i.e. it has a client-server architecture and is based upon modern Web technologies, such as JavaScript™, CGI-scripts, and Java™ where deemed appropriate), similar to the approach reported upon in Finnigan *et al.* [38].

The development of the MAB went through a succession of three prototype versions before the final version was made from scratch. Each of the prototypes contributed to the final version by experimenting with either the user interface or technology. This clear separation between prototyping on the one hand and regular development on the other resulted in a well-engineered tool.

Unlike the MAB, the Jolly Jumper was developed in an incremental fashion, where experience with the user interface and the technology was gained during the development of the intermediate versions. Although the result is considered acceptable, the Jolly Jumper did not reach the same level of quality as the MAB.

The development of URSA was carried out in a multi-disciplinary team; as a joint effort between PBC and Philips Research. PBC kept watch over the applicability for the specific application, the usability by the intended end users, the proper engineering, and the possibility to embed the set of tools in both the development and the product generation (i.e. build) process. Philips Research kept watch over a proper theoretical foundation, the use of state-of-the-art technology and the applicability of the tools comprising URSA in multiple development environments within Philips (rather than within the development environment of SOPHO only). In our view, the toolset would not have reached its current level of maturity without this co-development.

5.3. Introduction of URSA

5.3.1. Involvement of end-users

Changes are hard to carry through, even when these changes are for the better (see, for example [18]). Knowledgeable representatives of URSA's target group (the 'early adopters') were therefore involved right from its early conception and their requirements and feedback on intermediate versions was treated as those of regular customers, and concluded with an acceptance test. Whereas the feedback on the intermediate versions mainly focused on functionality, user-friendliness turned out to be the main concern during the acceptance test. In particular, the number of manual steps to be performed for the creation of a scenario should be minimized, and the generation of architectural information for packages presented by the MAB and ArchiSpy should be automated. Furthermore, all documentation (i.e. user manuals) should also be available on-line, preferably by means of Web-technology (just like URSA itself). Moreover, URSA should not only be available on a system on a developer's desk, but also on the test systems located next to (and connected with) the target systems (allowing, for example, immediate visualization by means of the Jolly Jumper of scenarios generated on a target system). For performance reasons, the latter requirement demanded for an improvement of the communication bandwidth between the test systems and target systems and an upgrade of some of the test systems. Finally, two serious omissions were identified for the Jolly Jumper. Firstly, it did not support dynamic scaling (i.e. means for zooming). As a consequence, scenarios involving many actors were felt to be hard to grasp. Secondly, Jolly Jumper did not support printing. These two omissions came as unexpected surprises during the acceptance test, and required non-trivial updates of the Jolly Jumper. URSA became available to all developers by the end of 1998.

5.3.2. Program understanding

The program understanding support of URSA is meant for *all* software developers. By providing an interactive user-interface and a link between the scenarios on the one hand and the actual code on the other hand, Jolly Jumper is a (considerable) improvement with respect to the existing tool support. The application of the Jolly Jumper requires only minor changes in the way of working, implying a low threshold and easing acceptance. The product-specific course is based on the Jolly Jumper and its use is explained within the course. The Jolly Jumper is used for program understanding during both corrective maintenance and perfective maintenance activities.

5.3.3. Complexity control

The complexity control support of URSA is in the first instance aimed primarily at software architects and designers; the average developer currently has less affiliation with this support of URSA. The MAB has been used for a number of perfective maintenance activities, and has proven to serve its purposes. The tabular representation of the MAB is an efficient and effective way to investigate the module architecture, and is sufficient to control the module architecture (in a way as described in Section 3.4). Typically, these maintenance activities involve many tables (in the order of magnitude of tens) rather than a few. The visualization of the structure even encouraged designers to improve the module architecture of existing components upon extensions as part of perfective maintenance activities.

The output of the MAB turns out to be less suitable for (static) documentation. As an example, it takes multiple tables to visualize mutual dependencies between multiple architectural entities especially at several levels of the decomposition structure (like the dependencies between subsystems A and B as shown in Figure 7(a)). As a consequence, graphical representations of selected views are typically created for documentation purposes.

The MAB has also been used to study SOPHO its current module architecture. Many unexpected dependencies were investigated in detail, and various undesirable dependencies identified. Re-architecting has only been performed in the context of perfective maintenance activities, however.

For the time being, it is not common practice that the consequences of maintenance activities on the module architecture are mapped out from the beginning and explicitly controlled during all stages of these activities. As a consequence, the current initial set of basic checks performed by ArchiSpy regularly reveals breaches as *reflection* on the result of a maintenance activity.

5.3.4. Impact on development process

The observation that ArchiSpy regularly reveals breaches as reflection on the result of a maintenance activity is a clear indication that, though necessary, the introduction of the tool support alone is insufficient. Changes in the development process are also necessary and essential for the control of the complexity. The visualization of the extracted module architecture of the system by means of the MAB is considered a first step towards such a change. The identification of breaches to the defined set of architectural rules represents a second step. Note that the breaches represent a state of affairs, which should preferably improve and at least not deteriorate (i.e. be secured). The active participation of those avowing the underlying principles is the next step towards complexity control. In order to give all involved in software development a common basic understanding of the role of architecture within PBC in general and within software development in particular, an architecture awareness course was

developed. The need to control architectural changes and use of the MAB is also explained within the course. All people involved in software development have attended the course (lasting approximately 1 day).

The impact on and the lead-time of introduction in the development process should not be underestimated, in particular when maintaining large, complex software systems that involve multiple project teams at different stages in their development life cycle. Introduction does not happen 'overnight', especially when software architecture is involved; after all, software architecture is still considered an art [51] rather than a matured engineering discipline.

The space of time from the start of the experiment described in Section 3.4 and the completion of the dissemination of the approach through the awareness course is about 5 years. Time will tell whether or not the envisioned changes in the development process are sustainable.

6. CONCLUSIONS

In this paper, the on-going efforts to improve the quality and maintainability of a large software system with a long lifetime have been described. It has been shown that changes in the market requirements, and the software development and target platform during this lifetime gave rise to activities in *all* categories of maintenance, including the less common preventive maintenance category.

The price to be paid for the success of any legacy system is its steadily growing complexity and the related difficulty to understand the system. In the literature [3,4,52], it is stressed that software architecture plays a vital role in the development (and hence maintenance) of large software systems. Large systems with a long lifetime typically do not have an up-to-date well-documented software architecture, however. Although research has reported successful experiments to extract the software architecture from a complex system's implementation for a decade [8,9], commercial off-the-shelf tools for either forward or backward (i.e. *reverse*) software architecting for large complex (legacy) systems are still scarcely available, if at all.

The quest for program understanding combined with the need to manage the software architecture led to the development of URSA. URSA is a set of tools that support program understanding and complexity control at the *architectural* level, leaving support at the *programming* level to standard tools that are readily available on the market.

In this paper, the support provided by URSA is described in detail. URSA bridges the gap between scenarios on the one hand and the process and development views of software architecture [34] on the other hand (by means of the Jolly Jumper). Next, it provides a means to visualize the module architecture (which is part of the development view) of a system (through the MAB) and to report non-conformances of the system to a set of architectural rules (through ArchiSpy). We found no evidence of another single toolset supporting scenarios, architectural visualization and verification, nor of the usage of a tabular representation for the presentation of the module architecture as provided by the MAB.

URSA is based on matured principles (requiring a minimal amount of innovation), and built using state-of-the-art technologies. The development of URSA was carried out in a multi-disciplinary-team as a joint effort between Philips Research and Philips Business Communications. The development of URSA has been embedded in an overall quality and maintainability improvement program of a PBX. Although URSA has been developed for a PBX, its application is not restricted to the telecommunications domain, however; instantiations of the generic parts of each of the tools are also used to support software development of other systems within Philips.

The approach taken and first experiences with the introduction of URSA within Philips Business Communications have been touched upon briefly. In our experience, organizational issues are considerably more demanding than technical issues when developing and subsequently introducing a toolset like URSA.

ACKNOWLEDGEMENTS

The authors would like to thank Léon Huijsdens, Roel P. de Jong, M. (Rien) J. Jordaan, Cees M. Klik, Ron L. C. Koymans, Jeroen P. Medema, Rob C. van Ommering, and Gerard J. J. M. van de Ven for the support, cooperation, help and discussions on the subject of this paper. We thank Angelo E. M. Hulshout, Jürgen K. Müller, and André Postma and the three anonymous referees of JSM for their valuable feedback on a previous version of this paper.

REFERENCES

1. Pressman RS. *Software Engineering—A Practitioner's Approach* (2nd edn). McGraw-Hill Book Co.: Singapore, 1987.
2. IEEE. *IEEE Software Engineering Standards Collection*. Institute of Electrical and Electronics Engineers: New York NY, 1993.
3. Perry DE, Wolf AL. Foundations for the Study of Software Architecture. *ACM, Software Engineering Notes* 1992; **17**(4):40–52.
4. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall Inc.: Upper Saddle River, NJ, 1996.
5. *IEEE Software*. 1995; **12**(6). (Special issue on Software Architecture).
6. IEEE. *Transactions on Software Engineering* 1995; **21**(4). (Special issue on Software Architecture).
7. Bass L, Clemens P, Kazman R. *Software Architecture in Practice*. Addison-Wesley Longman Inc.: Reading, MA, 1998.
8. Müller HA, Klashinsky K. Rigi—A system for programming-in-the-large. *Proceedings 10th International Conference on Software Engineering*, Raffles City, Singapore, April 1988. ACM, 1988; 80–86.
9. Schwanke RW, Altucher RZ, Platoff MA. Discovering, visualizing, and controlling software structure. *ACM Software Engineering Notes* 1989; **14**(3):147–150.
10. van Ommering RC. TEDDY user's manual. *Technical Report 12NC 4322 2730176 1*, Department for Information and Software Technology, Philips Research, Eindhoven, the Netherlands, 1993.
11. Roosen M. Design visualization definition and concepts. *IST Report RWB-508-re-94040*, Department for Information and Software Technology, Philips Research, Eindhoven, the Netherlands, 1994.
12. Feijs LMG, van Ommering RC. The theory of relations and its applications to software structuring. *IST Report RWB-508-re-95011*, Department for Information and Software Technology, Philips Research, Eindhoven, the Netherlands, 1995.
13. Krikhaar RL. Reverse architecting approach for complex systems. *Proceedings International Conference on Software Maintenance'97*, Bari, Italy, 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 4–10.
14. Feijs L, Krikhaar R, van Ommering RC. A relational approach to support software architecture analysis. *Software–Practice and Experience* 1998; **28**(4):371–400.
15. Feijs L, Krikhaar RL. Relation algebra with multi-relations. *International Journal of Computer Mathematics* 1998; **70**:57–74.
16. Feijs L, de Jong R. 3D visualization of software architectures. *Communications of the ACM* 1998; **41**(12):73–78.
17. Feijs L, van Ommering RC. Relation partition algebra—mathematical aspects of uses and part-of relations. *Science of Computer Programming* 1999; **33**:163–212.
18. Humphrey WS. *Managing the Software Process*. Addison-Wesley Publishing Company: Reading, MA, 1989.
19. *CCITT High Level Language (CHILL)—Recommendation Z.200*. Red Book, Volume VI—Fascicle VI.12. ITU: Geneva, Switzerland, 1985.
20. Keck DO, Kühn PJ. The feature and service interaction problem in telecommunications systems—a survey. *IEEE Transactions on Software Engineering* 1998; **24**(10):779–796.
21. Corbi TA. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 1989; **28**(2):294–306.
22. Ning JQ, Engberts A, Kozacynski WV. Automated support for legacy code understanding. *Communications of the ACM* 1994; **37**(5):50–57. (Special issue on Reverse Engineering.)
23. Mauw S, Winter T. A prototype toolset for interworkings. *Philips Telecommunication Review* 1993; **51**(3):41–45.
24. Pearse T, Oman P. Maintainability measurements on industrial source code maintenance activities. *Proceedings International Conference on Software Maintenance'95*, Opio, France, 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 295–303.

25. *QAC^{TM} 3.1 User Guide and Reference Manual*. Programming Research Ltd.: Hersham, UK, 1993.
26. Henry S, Kafura D. The evaluation of software systems' structure using quantitative software metrics. *Software—Practice and Experience* 1984; **14**(6):561–573.
27. Bowman IT, Holt RC, Brewster NV. Linux as a case study: Its extracted software architecture. *Proceedings 21st International Conference on Software Engineering*, Los Angeles CA, May 1999. ACM, 1999; 555–563.
28. Standard ECMA-179. *Services for Computer Supported Telecommunications Applications (CSTA)—Phase I*. ECMA: Geneva, Switzerland, June 1992.
29. Standard ECMA-190. *Protocol for Computer Supported Telecommunications Applications (CSTA)—Phase I*. ECMA: Geneva, Switzerland, June 1992.
30. Feijs LMG. Architecture visualization and analysis: motivation and example. *IST Report RWB-510-re-95042*. Philips Research: Eindhoven, the Netherlands, 1995.
31. Chen Y-FR, Fowler GS, Koutsofios E, Wallach RS. Ciao: A graphical navigator for software and document repositories. *Proceedings International Conference on Software Maintenance'95*, Opio (Nice), France, 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 66–75.
32. Krikhaar RL. Software architecture reconstruction. *Doctoral dissertation*, Faculty of Mathematics, Informatics, Physics and Astronomy, University of Amsterdam (UvA), 1999.
33. Soni D, Nord R, Hofmeister C. Software architecture in industrial applications. *Proceedings 17th International Conference on Software Engineering*, Seattle, 1995. ACM, 1999; 196–210.
34. Kruchten P. The 4 + 1 view model of architecture. *IEEE Software* 1995; **12**(6):42–50.
35. Krikhaar RL, de Jong RP, Medema JP, Feijs LMG. Architecture comprehension tools for a PBX system. *Proceedings 3rd European Conference on Software Maintenance and Reengineering (CSMR)*, 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 31–39.
36. Glas A. A module architecture browser—visualization of architectural information in support of reverse engineering. *Final Report of the Postgraduate Program Software Technology*, Stan Ackermans Institute, Department of Software Technology, Eindhoven University of Technology, 1998.
37. von Mayrhauser A, Wang J, Li Q. Experience with a reverse architecture approach to increase understanding. *Proceedings International Conference on Software Maintenance*, Oxford, England, 1999. IEEE Computer Society Press: Los Alamitos, CA; 131–138.
38. Finnigan PJ, Holt RC, Kalas I, Kerr S, Kontogiannis K, Müller HA, Mylopoulos J, Perelgut SG, Stanley M, Wong K. The software bookshelf. *IBM Systems Journal* 1997; **36**(4):564–593.
39. Carmichael I, Tzeropos V, Holt RC. Design maintenance: Unexpected architectural interactions. *International Conference on Software Maintenance'95*, Opio (Nice), France, 1995. IEEE Computer Society Press: Los Alamitos, CA; 1995; 134–139.
40. Holt RC. Structural manipulations of software architecture using Tarski relation algebra. *Proceedings 5th Working Conference on Reverse Engineering (WCRE'98)*, Honolulu, Hawaii, 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 210–219.
41. Eick SG, Ward A. An interactive visualization for message sequence charts. *4th Workshop on Program Comprehension (WPC'96)*, Berlin, March 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 2–8.
42. Murphy GC, Notkin D, Sullivan K. Software reflection models: Bridging the gap between source and high-level models. *ACM Software Engineering Notes* 1995; **20**(4):18–28.
43. *ITU Recommendation Z.120—Message Sequence Chart (MSC)*. ITU: Geneva, Switzerland, 1996.
44. *ITU Recommendation Z.100: Functional Specification and Description Language SDL Blue Book*, Volume X, X.1–X.5. ITU: Geneva, Switzerland, 1989.
45. Belina F, Hogrefe D, Sarma A. *SDL with Applications from Protocol Specification*. Prentice Hall International (UK) Ltd.: Hertfordshire, UK, 1991.
46. Kazman R, Carrière SJ. View extraction and view fusion in architectural understanding. *Proceedings 5th International Conference on Software Reuse (ICRS'98)*, Victoria, BC, June 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 200–299.
47. Dijkstra EW. The structure of the multi-programming system. *Communications of the ACM* 1968; **11**(5):341–346.
48. Zweben S, Edwards S, Weide B, Hollingsworth J. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering* 1995; **21**(3):200–208.
49. Harel D. On visual formalisms. *Communications of the ACM* 1988; **31**(5):514–530.
50. Bril RJ, Feijs LMG, Glas A, Krikhaar RL, Winter T. Hiding expressed using relation algebra with multi-relations—oblique lifting and lowering for unbalanced systems. *4th European Conference on Software Maintenance and Reengineering (CSMR)*, February 29–March 3, Zurich, Switzerland, 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 33–43.
51. Rechtin E, Maier MW. *The Art of Systems Architecting*. CRC Press: Boca Raton, FL, 1997.
52. Clemens P, Northrop L. Software architecture: An executive overview. *Technical Report CMU/SEI-96-TR-003*, Software Engineering Institute, Pittsburgh, PA, 1996.

AUTHORS' BIOGRAPHIES

Reinder J. Bril was a software architect at Philips Business Communications (PBC) and had a leading role in the work described in this article. He has recently joined Philips Research Laboratories Eindhoven (PRLE) and is currently working as a senior scientist in the area of Quality of Service (QoS) for consumer devices (such as digital TV-sets and set-top boxes), with a focus on dynamic resource management. He received a B.Sc. and a M.Sc. (both with honours) from the Department of Electrical Engineering of the University of Twente, the Netherlands. E-mail: Reinder.Bril@philips.com

Prof.dr.ir. Loe M.G. Feijs obtained his Master's degree in Electrical Engineering from the Eindhoven University of Technology (TUE) in 1979 and his Dr. degree in 1990. For many years, he worked on formal specification techniques at Philips Research. He has contributed to the design of COLD, worked on various industrial applications of formal specifications, and is the (co-) author of three books on formal specification. Since 1994, he has been part-time professor at the TUE, chair: industrial applications of formal methods. From 1998 onwards Feijs has been the scientific director of the EESI. His present research interests include message sequence charts, software component technology and applications of Internet technology to embedded systems.

André Glas graduated in computer science at the University of Twente. After that, he went to the University of Eindhoven to follow a two year post-graduate programme, also in computer science. It was during his final project for this post-graduate programme, that he worked on the project described in this paper. From 1998 onwards, he has been a software engineer at the National Aerospace Laboratory of the Netherlands. E-mail: glas@nlr.nl.

René Krikhaar is a software architect for Magnetic Resonance Systems at Philips Medical Systems in Best, the Netherlands. His (research) interests include software engineering, software architecture modeling and software architecture verification. Krikhaar earned a Ph.D. in computer science from the University of Amsterdam, the Netherlands. His thesis concerns software architecture reconstruction based on relation partition algebra. This theory has been applied on several industrial systems in different domains. Contact him at rene.krikhaar@philips.com.

M. (Thijs) R.M. Winter worked as a software architect at Philips Business Communications. Now he works at Philips Software Centre in Bangalore India. He received a M.Sc. in mathematics from the University of Utrecht and he received a post graduate certification in Information and Communication technology from the Technical University Eindhoven. After that he worked at the Philips Research Laboratories were he was active in the area of formal design methods and protocol specification and verification. At Philips Business Communications he participated in the development of new ISPBX features. His contact address is thijs.winter@philips.com.