Dynamic Analysis For Reverse Engineering and Program Understanding [*]

Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton, AB, T6G 2E8, Canada
stroulia@cs.ualberta.ca

Tarja Systä
Tampere University of Technology
Software Systems Laboratory
P.O.Box 553, 33101 Tampere, Finland
tsysta@cs.tut.fi

ABSTRACT

The main focus of program understanding and reverse engineering research has been on modeling the structure of a program by examining its code. This has been the result of the nature of the systems investigated and the perceived goals of the reverse engineering activities. The types of systems under investigation have changed, however, and the maintenance objectives have evolved. Many legacy systems today are object-oriented and component-based. One of the most prominent maintenance objectives is system migration to distributed environments, most notably the World Wide Web, for interoperation with other systems. This new maintenance objective has a great impact on the types of models expected as products of reverse engineering. As the traditional static software analysis techniques keep their valuable role in program comprehension, additional techniques, especially those focusing on run-time analysis of the subject systems, become equally important. In this paper, we focus on the analysis of the system's dynamic behavior, as it pertains to understanding the system's processes and uses. We give an overview of currently used dynamic reverse engineering techniques and identify some challenges yet to be tackled.

Keywords

Reverse Engineering, Dynamic Analysis

1. INTRODUCTION

Software that is used in a real-world environment must change or become less and less useful in that environment.
Lehman Law of evolution [29]

As the context, in which the software system is deployed, changes, the software has to be maintained and adapted in order to deliver the new functions required of it and to meet the new constraints imposed on it. A necessary pre-requisite for effectively maintaining and evolving a software system is to maintain an "operational" understanding of the system in question, and this is the objective of reverse-engineering research. According to Chikofsky and Cross, the purpose of reverse engineering is to analyze a system in order to identify its current components and their dependencies, and to create abstractions of the system design [4]. More specifically, reverse engineering of a software system takes almost always place in service of a specific purpose such as re-engineering to add a specific feature, maintenance to improve the efficiency of a process, reuse of some of its modules in a new system, or assessment in order to decide on whether or not to purchase it.

For a comprehensive understanding of any software system, several complementary views need to be constructed, capturing information about different aspects of the system in question. The "4+1 Views" model, introduced in [25], for example, identifies four different architectural views: the *logical* view of the system data, the *process* view of the system's threads of control, the *physical* view describing the mapping of the software elements onto hardware, and the *development* view describing the organization of the software modules during development. *Scenarios* of how the system is used in different types of situations are used to integrate, illustrate and validate the above views. However, in spite of our awareness that a single view is rarely sufficient for understanding a software system, the main focus of program-understanding and reverse-engineering research has been on identifying and modeling the structure of a program by examining its code. There are two main reasons for this focus: first, the nature of the systems investigated up to now and second, the perceived goals of the reverse-engineering activities. On one hand, the software systems that have traditionally been the subject of reverse engineering were mostly written in procedural languages and run on a single machine. At the same time, the maintenance goal, in support of which reverse engineering has traditionally been performed, has been the adaptation of a single legacy system, to extend its original functionality or to port it to a more modern platform.

Both the nature of the systems under investigation and the maintenance objectives have evolved. An increasing part of

[*](Produces the permission block, copyright information and page numbering). For use with ACM_PROC_ARTICLE-SP.CLS V2.0. Supported by ACM.

the legacy systems today is designed in the object-oriented style and often their components are distributed in multi-tier architectures. Furthermore, one of the most prominent maintenance objectives today is the migration of legacy systems to distributed environments, most notably the World Wide Web, for interoperation with other systems. This new maintenance objective has a great impact on the types of models expected as products of reverse engineering. Instead of identification of the inter-dependencies between the original data structures and procedures of the system, so as not to be disturbed during the subsequent program adaptation phase, the desired products are specifications of functionally coherent subsystems with interesting high-level behaviors that can be integrated in the interoperating system consortium.

The landscape of reverse-engineering research is now changing in response to the evolution of the overall problem. Software-architecture extraction is extending to include all the different aspects of software mentioned above. In this paper, we focus on the analysis of the system's dynamic behavior, as it pertains to understanding the system's processes and uses.

The rest of this paper is organized as follows. Section 2 reviews a range of reverse-engineering tasks for which dynamic-behavior analysis can play an important role. In Section 3 we discuss various aspects of software visualization supporting program comprehension. Special classes of software systems and the infrastructure necessary for inspecting their behavior is discussed in Section 4. Finally, section 5 outlines some directions for future work in reverse engineering, integrating static and dynamic analysis of software systems.

2. BEHAVIOR ANALYSIS FOR REVERSE-ENGINEERING TASKS

A variety of tasks, ranging from software assessment to system re-development, employ reverse engineering as a supporting activity. Some of these tasks require an overall understanding of the subject software systems while others focus on a specific part or a feature in the system. When big architectural level changes are required, the whole system typically needs to be understood. However, there are also cases in which the analysis is targeted at a specific part of the system. Such cases include, for instance, understanding the interface of an unknown (and possibly distributed) component or application and debugging the source of a specific (and typically false) behavior. In this section, we examine a range of reverse-engineering tasks and we discuss the role that dynamic-behavior analysis can play in these tasks.

2.1 Extracting System Modularizations

A lot of software-engineering research, both past and current, has emphasized modularity as an important quality of software systems. Modular systems are easier to understand and to maintain and a lot of effort has been put into characterizing the nature of alternative modularizations and providing methods to support the development of modular systems. It is not a surprise then that a substantial reverse-engineering effort has been devoted to extracting modularizations of legacy software systems.

A variety of alternative modularizations can be produced, based on different types of relations extracted with code analysis. In fact, many reverse engineering tools provide alternative views to the repository of relations extracted from the code, and enable the user to browse the relations of their choice, and the modularizations they give rise to. The basic technology underlying the modularization construction is clustering of closely related elements, and a range of clustering algorithms have been investigated in this context. The constructed clusterings always require user involvement; the user makes clusterings that she finds useful for understanding the subject system. Approaches and principles for making this a semi-automated process have been developed. For instance, the clustering can be based on structures and encapsulation properties of the language, for instance, packages in Java and some product metrics values. Such a metric-based principle could be "high cohesion and low coupling"[31].

Clustering-based approaches to understanding the modular structure of software systems suffer mainly from the following shortcomings:

- they are brittle: code modifications may result in disproportionate changes in the inferred modularizations;
- they have typically considered only static relations; and
- they are tightly dependent on the programming language, in which the subject software system is written.

There has been a lot of evidence to the brittleness of clustering-based approaches. An experiment reported in [45], for example, showed that random removals of up to 1% of the "extracted facts" database resulted in changes upwards of 2% in the resulting clusterings, in 10% of the cases. A battery of experiments, reported in [1], revealed that changes, especially in the ill-designed parts of a system, may result in steep changes in the modularizations produced by clustering. The brittleness of clustering depends on the clustering criteria used. A clustering just reflects a single point of view (and sometimes a guess) on the modular structure of the software according to the understanding of the reverse engineer. For example, a clustering criteria may be based on domain-independent issues (e.g., graph-theoretic measures such as connectivity) or on domain-dependent information (e.g., application naming conventions) [44]. Therefore, the clustering may be misleading if used for other purposes than that it was originally built for.

Call-graph dependencies have been extensively used as the basis for clustering the elements of a software system. Consider, however, two alternative call dependencies between two pairs of classes; the first call is actually executed once at the beginning of the program, where the second is exercised fairly regularly throughout the program execution. The first call may represent a case in which a configuration or an initialization function is called, possibly by several other functions in the system. Intuitively, one would think that the latter relationship is stronger than the former, especially if the re-engineering objective is to extract a "cluster" of collaborating classes to be reused in a different context. The heart of the problem is that, traditionally, only code-based metrics have been used to infer the strength of the relation between two elements - a prerequisite for defining distance in clustering. Dynamic metrics [32] could also be used. The advantage of such measurements is that, in general, they are good indicators for important external behavioral attributes, such as performance and memory usage, and could be used for the assessment of cluster quality in terms of non-functional requirements, such as maintainability, reliability, reusability, and usability.

Finally, different languages provide a different degree of syntactic support for encapsulation. In pure object-oriented languages, such as Java for instance, methods and variables are encapsulated inside classes, and the classes are further enclosed in packages. This, in turn, facilitates the usage of automated clustering algorithms. The quality of the generated clustering depends on how well the classes are formed (i.e., how related the methods and variables are) and how successful and descriptive their encapsulation in packages is. It does not, for instance, reveal situations in which only loosely cohesive parts are "forced together", that is, the principle of "high cohesion and loose coupling" is not followed. On the contrary, such a clustering may hide parts that violate this principle, while finding them would be most valuable for understanding and correcting the design flaws.

Clusterings are constructed for a certain purpose and from a certain point of view. In addition to static clusterings that aim at characterizing structural modularizations (e.g., subsystem identification), dynamic clustering that are based on behavioral issues can be constructed. Dynamic clusters are often represented as use cases and recurring behavioral patterns [43, 17, 14, 8]. The dynamic clusters can be used, e.g., for feature identification and visualization. It is worth noticing that static and dynamic clusterings do not typically match. On the contrary, they are usually orthogonal: a single use case presumably uses classes belonging to several structural clusters. Therefore, a static clustering does not provide much support in understanding the internal behavior of various features of the software, and may in fact, even obscure it.

The fundamental problem underlying modularizations constructed solely based on analysis of the static code structure is that a legacy system often includes "dead" and "glue" code. In component-based software engineering, software systems are ideally built by combining existing software components with well defined and clear interfaces in a "plug and play" fashion. However, this seldom is the case in reality. Instead, because teh constituent components do not interoperate, a lot of glue code is needed to plug the components together and to make them play. Furthermore, in net-centric software systems, a variety of problems that may occur at run-time, such as unannounced changes in the resources involved, errors and recovery from them for example, must be anticipated by the system's implementation. Finally, maintenance is often carried out using minimized effort. This, in turn, yields to uncontrolled evolution of the software and increased amount of glue code used. By studying the software evolution in various case studies, Lehman et al. have noticed that adding new code typically causes increasing complexity, declining comprehensibility, and increasing resistance to future changes [30]. The examination of a system's run-time behavior can help to focus the scope the overall problem of system understanding; it can focus the examination of the code only to these parts of the system structure exercised by the system use and it can illuminate the context in which data is used. Therefore, reverse engineering techniques that combine static and dynamic analysis provide valuable support for the engineer to understanding the need for glue code and, in general, the functionality and role of different software components [20, 28, 36, 42, 21].

2.2 Legacy Interface Migration

With the advent of e-commerce, one of the most important activities in the IT industry today is the web-enablement of legacy systems. For example, many organizations, in order to facilitate their collaborations with their partners or to improve their customer service, want to allow access of their proprietary legacy systems over the Web. The dominant industrial practice for addressing this problem has been "screen scraping", i.e., development of emulators for "translating" the data from(to) the legacy system, which is formatted for ascii terminal screens, into(from) modern WIMP interfaces and Web browsers. This practice, which has been exclusively manual, relies on the understanding the run-time behavior of the legacy user interface.

CelLEST [10, 19, 39, 41] provides a semi-automated method for reverse engineering the legacy user interface, in order to support exactly this type of practice. The CelLEST reverse engineering process starts by collecting traces of the users' interaction with the original legacy interface through specially instrumented emulators. Based on these traces, a state-transition model of the user interface of the legacy system is constructed. This model identifies the unique user-interface screens and the actions that the user can perform to navigate from one screen to another. The legacy interface screens are identified by clustering visually similar screen snapshots in the collected traces, and the syntax of the screen-transition actions is extracted as generalized patterns of specific actions performed in the collected traces. This interface model describes how the application is actually used through its current interface.

The next step of the process, after having constructed a model of the overall system interface, is to construct task-specific models of the system-user information exchange. For this step, traces of users executing the same task, albeit with different problem parameters, are collected. These task-specific traces are analyzed in order to identify the pieces of information provided(received) by the user to(from) the system. The resulting model is subsequently used as the basis for designing a new web-accessible interface that acts a legacy from-end, thus enabling the migration to the Web of specific tasks supported by the legacy system.

The CelLEST method is independent of the programming-language used to develop the application and as a result, it is quite generally applicable. In fact, a similar method has been applied to existing thin-client web-based applications [40], in order to construct adapter front-ends so that many applications can exchange data in a common formalism. Furthermore, it is quite lightweight in terms of the skills it assumes. Finally, since it does not change the legacy system implementation, it is almost without risk as long as the objective is service migration, as opposed to service modification.

2.3 Understanding the role of software artifacts

The end goal in software construction is to build software systems that deliver the behavior desired of them. The requirements of a system's end-user usually focus on specifications of its desired behavior. And as these requirements may change, understanding the role of the elements of the system structure becomes crucial.

Because of the strong dependencies between structural and behavioral aspects of software, static and dynamic analysis should also be coupled. In fact, experiments with users aiming at constructing task-specific explanations to answer software-understanding questions [11] have shown that developers use execution traces in order to identify the libraries involved in a particular behavior they need to modify.

In Shimba prototype reverse engineering environment [43, 42], static information is extracted from Java class files and viewed as a nested graph using Rigi reverse engineering environment [31]. The dynamic information is generated by running the target software under a debugger. The debugged event trace information (including, e.g., method invocations and thrown exceptions) is split (automatically or according to the user's guidance) into a set of sequence diagrams and visualized with a prototype tool called SCED [22].

Both static and dynamic views contain information about the software artifacts (e.g., classes and methods) and their relations (e.g., method calls). This information overlap enables, and also implies the need for, information exchange between the views. Static and dynamic views can thus be used to improve and modify each other and to understand the role of the software artifacts. For instance, the static Rigi views can be used to guide the generation of dynamic information and dynamic SCED views. This is useful when the engineer is interested in a specific part of the software. It is not meaningful then to generate a huge amount of trace information for the whole system. Shimba further allows the user to slice the static Rigi graph with a selected set of SCED sequence diagrams. This technique can be used to analyze which parts of the software are used to implement the captured behavior and how these parts have been constructed. The roles of the high-level static clusters can be understood by using them to construct high-level sequence diagrams that view the interaction among these clusters (e.g., components in a MCV architecture).

Many dynamic reverse engineering tools use variations of Message Sequence Charts (MSCs) [15] to visualize the run-time behavior of the target object-oriented software system [14, 17, 23, 28, 6]. In Shimba, the visualization of the run-time behavior of object-oriented software has been taken one step further: for selected objects and methods taking part in the sequence diagrams, state diagrams can be composed automatically using SCED. Generated state diagrams allow the user to examine the dynamic behavior from a different angle compared to sequence diagrams. While sequence diagrams show the interaction between several objects, a state diagram shows the overall behavior of an object or a method of interest in the system. Optionally, information about the dynamic control flow of selected objects can be extracted and added to the sequence diagrams. Using the state diagram synthesis feature of SCED, the dynamic control flow of a selected object or a method can be visualized as a state diagram.

2.4 Debugging and Profiling

There are a lot of performance-related properties of software systems, which are of extreme importance in assessing its overall quality. The are usually not visible by examining the software code, but they become apparent when its dynamic behavior is analyzed. Such properties include memory management, code usage, and efficiency, to name a few.

Efficiency, especially, is crucial for time critical systems. For improving slowly behaving systems, the memory management needs to be understood. Jinsight is a tool for visualizing the dynamic behavior of Java programs [14, 7]. It views information about object population, method invocations, garbage collection, CPU and memory bottlenecks, thread interactions, and deadlocks. [46] introduces another method for visualizing program execution information using high-level models. The visualization focuses on object information and interaction information (e.g., a current call stack and a summary of calls). The object information, in turn, includes information on the lifetime (creation and destruction) of objects.

Overall correctness is also difficult to establish. When a bug is discovered, for example, the first step is to reproduce it in order characterize the situations when it occurs. Goal-driven reverse engineering approaches are especially useful for debugging. Tracking down a bug might be difficult. For example, consider a software system that is irregularly unstable. In this case, it might not be sufficient to know when the failure occurs, or what events happened before the failure; the engineer needs to find out in which order these events occurred before the failure. The visualization techniques used in reverse engineering tools also support software debugging. In Shimba, the exceptional behavior of the subject system is first recorded and then used to slice the static models in order to support the engineer to understand how the parts of the software causing this false behavior are built [43].

3. VISUALIZATION

Most reverse-engineering tools come with visualization capabilities. In addition to constructing models in some internal representation, they usually also provide a visual representation of these models. The underlying reason is that the software developer using the tool has to "trust" the constructed models to be correct before using them for her re-engineering or maintenance task. Visual representations have been found particularly effective in communicating complex information to the users who have to understand the extracted models. Since different tasks require different types of information to be shown to the user, a variety of graphical notations have been developed, covering a wide spectrum of information abstraction and detail.

Reverse-engineering tools visualizing dynamic system behavior often use variations of directed graphs. For example, a directed graph can be used to visualize the run-time object interactions by representing objects as nodes and method calls or variable accesses as arcs between the nodes. When sequential information on the order of the calls is desired, various kinds of scenario diagrams are used. Both of these graphical representations are simple and fairly intuitive and thus suitable to be used for program understanding purposes. However, without notational extensions, they do not scale up. A large amount of run-time information is typically generated, even as a result of a relatively brief usage of the system. Thus, managing and abstracting the extracted information is necessary. This is usually the most challenging problem in dynamic reverse engineering. Behavioral patterns are often used to build abstract views of the dynamic event trace information. High-level views can also be constructed by taking advantage of the static clustering.

The extracted information is not useful unless it can be shown in a readable and descriptive way. Current reverse engineering tools use basically two different approaches to visualize the extract information, possibly originating from different sources: (1) all the information is merged into a single view thus avoiding the problem of keeping the different views synchronized or (2) different views for different purposes are used. Both of these approaches have advantages and disadvantages.

A single view directly illustrates connections, e.g., between static and dynamic information. In addition, the quality of the view can be insured when merging static and dynamic information. This approach is used, for instance, in the Dali tool [20]. On the other hand, building abstractions for merged views can be difficult because static and dynamic abstractions usually differ considerably as discussed in Section 2.1. Furthermore, forming merged views themselves might be complicated. It is easy to add, e.g., information on code usage to a static view but it is much more difficult to add information about concurrency or sequential behavior. Finally, an inevitable problem is that the more information is attached to a single view, the less readable it becomes, thus failing to fulfill one of its main purposes.

In forward engineering, UML [33, 37, 3] has become an industrial standard for the presentation of various design artifacts in object-oriented software development. UML provides different diagram types that can be used to view a system from different perspectives and/or at different levels of abstraction. Hence, the various UML models of the same system are not independent specifications but strongly overlapping, depending on each other in many ways [38]. From a large set of diagrams, the user chooses the ones that best suit for her purposes. Ideally, this should be the case also in reverse engineering. If a large set of diagrams is chosen, the problem of keeping them consistent and connected to each other needs to be attacked, as in forward engineering. However, UML is currently an informal notation and does not support "tight" cross-referencing among the different models of the system. Furthermore, its behavioral models are insufficient to visualize all interesting aspects of the system run-time behavior, such as behavioral patterns. They are quite verbose and result in large diagrams for fairly small interactions. In addition, UML does not support composition of behavioral diagrams, and as a result representation of complex behavior becomes in effect impossible.

The number and type of diagrams to be used depend on the purpose and needs in the same way as in forward engineering. Distinguishing static and dynamic views allows showing information that would be hard, or event impossible, to include in a single merged view. This, in turn, offers extended possibilities to support program slicing and to build abstractions, requiring that there is a connection that enables information exchange between the views.

In principle, a notation should support multiple types of models, loosely corresponding to the complementary views discussed above. A comprehensive notation should enable

- *problem-domain* descriptions, possibly in natural language or visually with domain-specific icons;
- *user-specific* descriptions, such as program features, the basic services on which they are built and how they can be invoked from the system user interface;
- *structural* descriptions of the basic components of the system and their connectors;
- *behavioral* descriptions of how these components collaborate, i.e., synchronize and exchange data, at run time; and
- *physical* descriptions of how the code base is organized and controlled during development and how the processes are distributed on the hardware during execution.

In addition to providing such complementary views, the notation should enable the user to consistently review and modify them. For example, if the user reviews and changes the system structure, she should be able to see these changes reflected in the behavioral description, or at least the need for corresponding changes should be flagged. Furthermore, within a single view, more or less detailed descriptions should be supported. For example, the structure of an object-oriented system could be viewed at the class level or at the component level, and its run-time behavior could be described in terms of an execution instance, such a trace for example, or at an aggregate level, summarizing multiple execution instances. Such cross-referencing and translation support would help users to understand the connections among different views, to formulate complex queries about the underlying system, and to construct comprehensive explanations about its design.

Unfortunately, current reverse-engineering environments and the notations they adopt offer limited (if any) support for the tasks mentioned above. Many adopt single view notations. Others adopt informal notations, such as UML for example, which although comprehensive, does not have a sufficiently strong formal meta-model yet to synchronize its multiple views. In fact, the degree of formality of the different notations employed in reverse-engineering tools ranges from "fuzzy", e.g., UML, to formal, e.g., SDL [16] and MSC [15]. Others, yet, adopt independent notations for their multiple views with no well-defined interpretations of the dependencies of the different models. In these cases too, formalization seems necessary.

This need for increased formalization in support for flexible view cross-referencing might be in conflict with the need for comprehensible visualization. Given that one of the objectives of reverse engineering is to improve the developer's understanding of "how the system works", other, less formal and possibly more cognitively motivated mechanisms might be employed. One possibility is to define the semantic relations among the views in a more relaxed way, yet precise enough to allow the information exchange desired. Another possibility could be to employ metaphors. Metaphors rely on the understander's deep knowledge of a similar base domain, and enable her to construct knowledge of the new and less familiar domain by analogy to the entities and relationships in the base domain. In the context of view construction, a metaphor is a conceptual notion (e.g., a feature) that is independent from the underlying subject system and may thus have any kind of implementation. Therefore, with metaphors the engineer can use intuitive knowledge of one domain to understand the other domains [34].

4. BEHAVIORAL INFORMATION COLLECTION INFRASTRUCTURE

By now, the role of behavior understanding in the context of reverse engineering and program understanding should be clear enough. The question then becomes: what is the infrastructure needed to enable the collection of behavioral information during the execution of the system? The answer to this question depends on the nature of the system under investigation and its implementation.

4.1 Centralized Systems

During the run-time of an object-oriented program, information to be collected contains the following:

- events on object construction and destruction,
- events for method entry and exit
- static type information, such as class structure and member function declarations, and
- dynamic type information to resolve classes of the objects at run-time.

In addition to the information on member functions, values of variables are collected in some systems [23]. This is useful for debugging purposes, for example.

Source code instrumentation is, perhaps, the most common way to extract run-time information of the subject program. Instrumentation tools use parsers to find the places where the instrumentation code is to be added. If the language to be analyzed is simple enough (e.g., languages that use LL(k) or LR(k) grammar), the program can be efficiently parsed. Complicated languages, such as the hybrid language C++, are, however, more challenging to be readily parsed.

Traditional instrumentation techniques may need to add pieces of code in many places to detect both method entries and exits. In object-oriented programs, dynamic binding introduces another challenge for detecting which methods have been actually called. On the other hand, object-oriented languages allow tracking the usage of methods by simply adding a local variable assigned with a new instance of a specific class as its value in the beginning of the method [35]. The actual instrumentation code can then be written in the constructor and the destructor of this class; when this method is called during execution, the constructor is called and whenever it is exited the destructor is called.

Alternative ways to generate run-time information is using debugger-based solutions [27, 28, 43]. One advantage of using debuggers is that the source code remains untouched. As a downside, debuggers typically slow down the program execution, which could be crucial, especially if concurrency (e.g., it uses several threads) is an issue in the subject program. Moreover, run-time virtual machine instrumentation is yet another approach to generate dynamic information of interest [14]. As usage of debuggers, virtual machine instrumentation leaves the source code of the subject system unchanged.

Irrespective of the information extraction technique used, monitoring run-time information can become too complex. For example, for complex systems, full instrumentation usually results in too much data. In general there are two approaches to solving this problem: (1) selective instrumentation and (2) filtering of the instrumentation data before and/or after analysis and visualization. Meta-level object protocols, such as Aspect Oriented Programming (AOP) [18] provide an interesting possibility for selectively instrumenting parts of the software. Such approaches could be used, e.g., in feature extraction.

Both these approaches to limit the information space assume some understanding of the static structure of the system under analysis. In both cases the instrumentor has to select the specific system components that are of interest. In the case of selective instrumentation, only these components will be instrumented and their interactions will be explored. In the information filtering approach, search-based methods are used to identify the components that may have an impact on the components of interest.

In object-oriented systems, due to polymorphism supported by object-oriented languages, components can be loaded at run-time and the architecture of the system thus evolves during execution. Dynamic reverse engineering techniques are especially valuable for analyzing such systems. Since components need to be adaptable to dynamic changes, component-based systems cannot be fully understood without recovering the component dynamism.

The above methods enable a different level of run-time user control, in terms of starting and stopping the data recording. For example, when interested in how a particular feature is implemented, accesses of a particular set of data are of interest; these accesses may not be of interest when another feature is under investigation.

Multiple scenarios of a component's actual behavior can be collapsed into state machines. Such a state machine depicts an overall behavior of the component of interest. In general there are two approaches to discovering a component's state machine from its dynamic behavior: (1) analysis of the component itself [42, 43] and (2) identification of its clients and their usage of the component [10, 19, 39, 41]. The difference is akin to "server" vs. "client" -side instrumentation. In the first case, the calls of methods of interest (e.g., interface methods of the component) and possibly methods with which they have dependencies are detected. In the latter case, in turn, the component itself is treated as a black box but its behavior is analyzed based on how its clients use its services. While the former allows more refined information to be detected (i.e., method calls and their consequences), the latter provides a light-weight approach that is independent of the actual implementation of the component.

4.2 Concurrent Systems

Concurrent systems present special challenges to the collection of their run-time behavior profile. To understand the overall system behavior, one has to understand the way the various processes interact, i.e., how events in the context of one process affect events in the context of the others. These interactions manifest themselves only at run time and cannot be examined statically on the code, since there often is no commonly accessible code base. The first problem then is the instrumentation of the execution environment in order to collect traces of the processes' events. There exist possible alternative instrumentation techniques, depending on the implementation platform of the system in question.

If the various processes run on the same machine, instrumentation of the underlying operating system is an option. In [2, 9] for example, the interactions between several Windows applications are recorded via hooks that intercept DLL communications between the individual applications and the operating system. This mechanism is in fact used, not simply for recording the interaction of these applications, but actually for establishing their integration as off-the-shelf components. Even when the interacting applications are distributed, they may run within the context of a distributed framework, such as CORBA. In such cases, the component that plays the role of the object request broker (ORB) can be instrumented. In such environments, a client process obtains access to the server process through a request to the ORB; thus, instrumentation at this level provides information about two-way client-server relations. In addition, in cases where all subsequent message exchanges between the communicating processes are mediated by the ORB, a precise record of the collaboration can be constructed [12].

If the system is truly distributed, and the various processes run on different machines with no common run-time infrastructure support, the only possible option is to instrument each individual process independently and to try to infer an overall model of their interdependencies by integrating the independent traces. To that end, a consistent "logical" clock [26] has to be implemented so that the individual processes can be "aligned" in time and a consistent global chronology of events can be constructed. [13] describes a method for constructing a Layered Queuing Network performance model for message passing concurrent systems using a graph-rewriting method for analyzing the collecting traces. A corresponding tool is used to visualize the extracted information.

Another interesting class of distributed systems is client-server systems with thin clients acting mostly as user interfaces for the services provided by the server. Such systems include legacy systems running on mainframe hosts with user-interface clients running on terminal emulators and web-based applications with browser-accessible thin clients. There are two alternative instrumentation options for such systems: client-side or server-side instrumentation. Client-side instrumentation is, in principle, preferable because it provides traces of coherent behavior, on the basis of which models of coherent user tasks and user-behavior profile and be extracted. This is, for example, the technology underlying the CelLEST project that aims at constructing models of the system uses for migrating these uses to modern platforms.

Similar technologies are also used in web-based applications, where client-side proxies are used to simplify the user's task, invoking services on behalf of the user. However, client-side instrumentation may not be possible, when the clients are not "controlled" by the service provider, which raises issues of client privacy. This is the case for web sites and web-based applications for examples, where the server may attempt to record the behavior of individual clients (e.g., using cookies) but the clients may prevent this from happening by rejecting them. In this case, server-side instrumentation becomes the sole alternative and the problem of modeling client behavior is exacerbated by the problem of simply recognizing which of the events received by the server belong to which client.

5. FUTURE RESEARCH PROBLEMS

As the complexity of the software systems being developed increases, the complexity of the systems that need to be understood also increases. And as software researchers and practitioners have been working on designing and reasoning about high-level structural concepts from multiple points of view, reverse-engineering research and practice focuses increasingly on extracting a more comprehensive understanding of software, at a higher-level of abstraction. The diversity and complexity of the systems to be analyzed creates a demand to construct models supporting the engineer in comprehending different aspects of the system. In that effort, analyzing the system's dynamic behavior will play an increasingly important role. There are several open problems that need to be addressed to better exploit the "information potential" of behavioral analysis.

The multiple models represent alternative views to the subject system. For enabling slicing and abstractions mechanisms cross the models, the semantic relations among them should be well defined. It would be useful (at least in some cases) to reflect modifications in one view directly in the other views. Moreover, for program comprehension purposes, the reverse engineering environment should allow the user to easily navigate between static and dynamic views as well as between low and high level views. For instance, the user might want to select a component in one view and explore its role in the other views. The current UML tools, for instance, exploit a rather superficial layer of the UML semantics. Taking advantage of the existing logical dependencies and semantic contents, much stronger support can be achieved, allowing the construction of a coherent set of operations to manipulate UML models [24]. Such a "UML model calculus" would also be useful for reverse engineering purposes.

The application of reverse engineering techniques is not limited to understanding old legacy systems. They can and should be applied to support forward engineering as well. In software development, reverse engineering the current static structure of the software helps the engineer to ensure that the architectural guidelines are followed, to get an overall picture of the software, to document the implementation steps, and so on. Reverse engineering the run-time behavior during the software development phase is essential for profiling, debugging, understanding and ensuring the current behavior of the software system and its components, etc. Applying reverse engineering techniques during the software development phase also supports documentation, hence avoiding ending up in the similar situation with our current systems, as we now face with legacy COBOL and C code. Ideally, reverse engineering tools should be able to produce standard OOAD models (i.e., UML models) from the subject software. Since such models are familiar to the user, this would unburden her from learning yet another diagram notation. Moreover, if the models used in forward and reverse engineering are the same, the tools would be able to give more support for re-engineering, round-trip-engineering, maintenance, and reuse. The fact that current reverse engineering tools typically use their own notation (in most cases, directed graphs) is one of the major obstacles in their integration with the mainstream tools. At least, there should be tools for converting the used models to UML models understandable and usable for the software developers.

Finally, as the need for application integration increases, the problem of specifying and adapting interfaces becomes more crucial. WebServices, the new stack of standards for integration of the web, offers a new specification language and inevitably creates a challenge to produce extractors of WebServices specifications from systems developed in different languages and platforms. In general, the granularity of software building blocks has increased from functions and variables to software components and application frameworks. Understanding the usage and protocols of the interfaces offered by the components becomes increasingly important, since the components themselves are often used as black boxes.

6. ACKNOWLEDGMENTS

This paper reviews many issues discussed during two working sessions, organized in the context of the two most recent meetings of the premier conferences focusing on program understanding and reverse engineering, i.e., the 23rd International Workshop on Program Comprehension, held in May 2001 in Toronto Canada and the 8th Working Conference on Reverse Engineering, held in October 2001 in Stuttgart Germany. The authors wish to thank the participants of the two discussion sessions, held at IWPC 2001 and WCRE 2001, for their insightful comments. We would especially like to thank Claire Knight for her detailed comments on the WCRE 2001 discussion session.

7. REFERENCES

[1] N. Anquetil, T. Lethbridge, Experiments with Clustering as a Software Remodularization Method, In *Proc of the 6th Working Conference on Reverse Engineering*, 1999, Atlanta, Georgia, pp. 235–125.

[2] R. Balzer, N. Goldman, Mediating Connectors, In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems* , Austin, Texas, 1999, pp.73–77.

[3] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[4] E. J. Chikofsky, J. H. Cross II., Reverse engineering and design recovery: A taxonomy. *IEEE Software*, **7**, 1, 1990, pp. 13–17.

[5] R. Clayton, S. Rugaber, L. Wills, On the Knowledge Required to Understand a Program, In *Proc. of the 5th Working Conference on Reverse Engineering*, 1998, pp. 69–78.

[6] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman, Execution patterns in object-oriented visualization, In *Proc. of the Conference on Object-Oriented Technologies and Systems*, 1998.

[7] W. De Pauw, G. Sevitsky, Visualizing Reference Patterns for Solving Memory Leaks in Java, *Concurrency: Practice and Experience 2000*, 12, pp. 1431–1454.

[8] G. Di Lucca, A. Fasolino, and U. Carlini, Recovering Use Case Models from Object-Oriented Code: A Thread-based Approach, In *Proc of the 7th Working Conference on Reverse Engineering* , Brisbane, Queensland, Australia, 2000, pp. 108–117.

[9] A. Egyed, R. Balzer, Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability, In *Proc. of the 16th IEEE International Conference on Automated Software Engineering* , San Diego, USA, 2001.

[10] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, B. Matichuk: Modeling the System-User Dialog Using Interaction Traces. In the *Proc. of the 8th Working Conference on Reverse Engineering* , Stuttgart, Germany, 2001, pp. 208–217.

[11] A. Erdem, W. Johnson, Task Orientation and Tailoring of Interactive Software Explanations, In *Proc. of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, 1999, pp. 145–156.w

[12] J. M. Favre, GSEE: a Generic Software Exploration Environment, *Proc. of the 9th International Workshop on Program Comprehension*, Toronto, Canada, May 2001, pp. 233-244.

[13] C. Hrischuk, C.M. Woodside, J. Rolia and R. Iversen, Trace-based load characterization for generating software performance models, *IEEE Transactions on Software Engineering*, vol. 25, no. 1, January 1999. 122-135

[14] IBM Research, *Jinsight, visualizing the execution of java programs*, http://www.research.ibm.com/jinsight/, 2000.

[15] Z.120 ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1999.

[16] Z.100 ITU-T Recommendation Z.100: Specification and Description Language (SDL), ITU-T, Geneva, 1999.

[17] D. Jerding and S. Rugaber, Using Visualization for Architectural Localization and Extraction, In *Proc. of the 4th Working Conference on Reverse Engineering*, 1997, pp. 56-65.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold: Getting started with ASPECTJ, *Communications of the ACM*, 44(10), pp. 59 – 65, 2001.

[19] R. V. Kapoor, E. Stroulia: Mathaino: Simultaneous Legacy Interface Migration to Multiple Platforms, In *Proc. of the 9th International Conference on Human-Computer Interaction*, Vol 1, New Orleans, LA, USA, 2001, pp. 51–55.

[20] R. Kazman, J. Carriere, Playing Detective: Reconstructing Software Architectures from Available Evidence, *Automated Software Engineering*, **6**,2, 1999, pp. 107–138.

[21] J. Korn, Y.-F. Chen, E. Koutsofios, Chava: Reverse Engineering and Tracking of Java Applets, In *Proc. of the 6th Working Conference on Reverse Engineering*, 1999, pp. 314–325.

[22] K. Koskimies, T. Männistö, T. Systä, J. Tuomi, Automated Support for Modeling OO Software, *IEEE Software*, **15**, 1, 1998, pp. 87–94.

[23] K. Koskimies and H. Mössenböck, Scene: Using scenario diagrams and active text for illustrating object-oriented programs, In *Proc. of the 18th International Conference on Software Engineering*, 1996, pp. 366–375.

[24] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies, Towards Tool Assisted UML Development Environments, In *Proc. of the 7th Symposium on Programming Languages and Software Tools*, Szeged, Hungary, 2001, pp. 1-15.

[25] P. Kruchten, The 4+1 View Model of Architecture. *IEEE Software*, November 1995, **12**, 6, pp. 42–50.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), pp. 558–565, 1978.

[27] D. Lange D and Y. Nakamura, Interactive Visualization of Design Patterns Can Help in Framework Understanding, In *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* , ACM Press, 1995, pp. 342–357.

[28] D. Lange, Y. Nakamura, Object-Oriented Program Tracing and Visualization, *IEEE Computer*, **30**, 5, 1997, pp. 63–70.

[29] M. Lehman, On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, **1**, 3, 1980, pp. 213–221.

[30] M. Lehman, D. Perry, J. Ramil, W. Turski and P. Wernick, Metrics and Laws of Software Evolution: The Nineties View, In *Proc. Metrics 97*, 1997, pp. 20–32.

[31] H. Müller, M. Orgun, S. Tilley, J. Uhl, A Reverse-engineering Approach to Subsystem Structure Identification, *Software Maintenance: Research and Practice*, **5**, 1993, pp. 181–204.

[32] J. C. Munson, T. M. Khoshgoftaar, Measuring Dynamic Program Complexity, *IEEE Software*, November 1992, pp. 48–55.

[33] OMG, *The Unified Modeling Language v1.4*, http://www.omg.org/uml, 2001.

[34] D. Ploix, Building Program Metaphors, In *Proc. of PPIG'96 PostGraduate Students Workshop*, Matlock, UK, September 1996.

[35] T. Raitalaakso, Dynamic Visualization of C++ Programs with UML Sequence Diagrams, MSc Thesis, Tampere University of Technology, 2000, http://practise.cs.tut.fi/pub/index.html

[36] T. Richner, S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, In *Proc. of the International Conference on Software Maintenance* , 1999, pp. 13–22.

[37] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[38] P. Selonen, K. Koskimies, M. Sakkinen, How to Make Apples from Oranges in UML In *Proc. of the Hawaii International Conference on System Sciences*, IEEE Computer Society, 2001.

[39] E. Stroulia, R. Kapoor: Reverse Engineering Interaction Plans for Legacy Interface Migration, *2002 Computer Aided User-Interface Design*, 2002, to appear.

[40] E. Stroulia, J. Thomson, and Q. Situ: Constructing XML-speaking wrappers for WEB Applications: Towards an Interoperating WEB, In *Proc. of the 7th Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, 2000, pp. 59–68.

[41] E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson and B. Matichuk: Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach, In *Proc. of the 6th Working Conference on Reverse Engineering*, Atlanta, GA, 1999, pp. 292–302.

[42] T. Systä, K. Koskimies, H. Müller, Shimba - An Environment for Reverse Engineering Java Software Systems, *Software Practice & Experience*, **31**, 4, 2001, pp. 371–394.

[43] T. Systä, Static and Dynamic Reverse Engineering Techniques for Java Software Systems, PhD Thesis, University of Tampere, Dept. of Computer and Information Sciences, Report A-2000-4, 2000.

[44] S. Tilley, K. Wong, M.-A.D. Storey, and H. Müller, Programmable Reverse Engineering, *International Journal of Software Engineering and Knowledge Engineering*, 1994, pp. 501–520.

[45] V. Tzerpos, R. Holt, MoJo: A Distance Metric for Software Clusterings, In *Proc. Of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, USA, 1999, pp. 187-193.

[46] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. Isaak, Visualizing Dynamic Software System Information through High-level Models, In *Proc. of the 1998 ACM Conference on Object-Oriented Programming, Systems, Languages, and Application*, ACM Press, 1998, pp. 271–283.