On the Use of Metaballs to Visually Map Source Code Structures and Analysis Results onto 3D Space

Juergen Rilling and S. P. Mudur
Department of Computer Science,
Concordia University, Canada
rilling@cs.concordia.ca, mudur@cs.concordia.ca

Abstract

Many reverse-engineering tools have been developed to derive abstract representations from existing source code. Graphic visuals derived from reverse engineered source code have long been recognized for their impact on improving the comprehensibility of the structural and behavioral aspects of software systems and their source code. As programs become more complex and larger, the sheer volume of information to be comprehended by developers becomes daunting. In this paper, we combine dynamic source analysis to selectively identify source code that is relevant at any point and combine it with 3D visualization techniques to reverse engineer and analyze source code, program executions, and program structures. For this research, we focus particularly on the use of metaballs, a 3D modeling technique that has already found extensive use representing complex organic shapes and structural relationships in biology and chemistry, to provide suitable 3D visual representations for software systems.

Keywords: software visualization, program slicing, 3D modeling, metaballs, visual mapping.

1. Introduction

Reverse engineering as part of program comprehension can be described as the process of analyzing subject system components and their interrelationships to create a higher level of abstraction and to understand the program execution and the sequence in which it occurred. The goal of software visualization is to acquire sufficient knowledge about a software system by identifying program artifacts and understanding their relationships. As programs become more complex and larger, the sheer volume of information to be comprehended by the developers becomes daunting. It would be ideal to be able to simultaneously view and understand detailed information about a specific activity in a global context at all times for any size of program. As Ben Shneiderman explains in [25,26], the main goal of every visualization technique is "Overview first, zoom and filter, then details on demand". This means that visualization should first provide an overview of the whole set of data then let the user restrict the set of data on which the visualization is applied, and finally give more details on the part of interest to the user. Software visualization of source code can be further categorized in static views and dynamic views. The static views are based on a static analysis of the source code and its associated information and provide a more generic high-level view of the system and its source code. The dynamic view is from analysis of monitored program execution. Based on their available run-time information, dynamic views can provide a more detailed and insightful view of the system with respect to a particular program execution. Compared to the static views, the dynamic nature of the information requires additional overhead while gathering the required data. As Mayhauser [16] illustrated, dynamic and static views should be regarded as complementary views rather than being mutually exclusive.

Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a program's functionality, programmers will focus on selected functions (outputs) with the goal of identifying which parts of the program significantly influence those particular functions. One approach is to apply program slicing that allows for a reduction of data to be displayed by including only those software entities (files, modules, classes, functions, statements and objects) that are relevant with respect to the computation of a specific program function of interest. Program slicing is a well-known decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a selected program function (output). This is particularly of interest for the analysis and comprehension of large software systems and program executions associated with them.

Visual representations of programs, primarily based on some diagrammatic notation, have been evolved right from the early days of computing [27]. However, for large, complex software systems, the comprehension of such

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 IEEE

diagrammatic depictions is restricted by the resolution limits of the visual medium (2D computer screen) and the limits of user's cognitive and perceptual capacities. One approach to overcome or reduce the limitations of the visual medium is to make use of a third dimension by mapping source code structures and program executions to a 3D space. Mapping these program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code. In this paper, we focus on the use of metaballs, often also referred to as metablobs, soft objects, point clouds or more generally implicit surfaces, a 3D modeling technique that has found extensive use in representing and visualizing complex organic shapes and structural relationships such as the DNA, humans, animals and other molecular surfaces [4,5,32,33,34]. In this research, we extend the application domain of metaballs to include the visualization and comprehension of very large program artifacts. The extent of their applicability in other domains has been such that virtually every significant commercially available 3D modeling software incorporates metaball modeling and rendering in some fashion or the other. Correspondingly, there are a large number of free software sites for packages supporting this technology [1]. However, to the best of our knowledge, ours is the first such attempt to apply the metaball metaphor in software visualization.

The rest of this paper is organized as follows: Section 2 introduces background related to 2D and 3D visualization techniques and program slicing. Section 3 discusses application of metaballs in combination with program slicing for typical software comprehension tasks. Section 4 presents a summary and some possible extensions.

2. Background

Program Comprehension

The increasing size and complexity of software systems introduces new challenges in comprehending the overall program structure, their artifacts and the behavioral relationships among these artifacts. Numerous theories have been formulated and empirical studies conducted to explain and document the problem solving behavior of software engineers engaged in program comprehension [6,7,11,16,22]. The *bottom-up* approach reconstructs a high level of abstraction that can be derived through reverse engineering of source code. The *top-down* approach applies a goal-oriented method by utilizing domain/application specific knowledge to identify parts of the program that are necessary for identifying the relevant source code artifacts. Both top-down and bottom-up comprehension models have been used in an attempt to defined how a software engineer understands software systems. Studies have shown that, in reality, software engineers switch between these different models depending on the problem-solving task [16]. This *opportunistic* approach can be described best as exploiting both top-down and bottom-up.

When it comes to comprehension of very large programs, humans are limited in the density of textual information they can resolve and comprehend [2,16,17,18,24]. Visualization in the form of reverse engineered 2D diagrams (e.g., collaboration diagrams, call-graphs, etc.) and models (UML class models) are suggested in the literature [References] to provide users with higher abstraction views on the software under investigation. For large software systems it becomes increasingly difficult to comprehend these diagrams for several reasons: (1) the diagram complexity is increased because of the large amount of information to be displayed, (2) the awkward layout techniques provided by the visualization approach,
(3) their non-intuitive navigation, and (4) often their very specialized scope in depicting only certain program artifacts and their relationships.

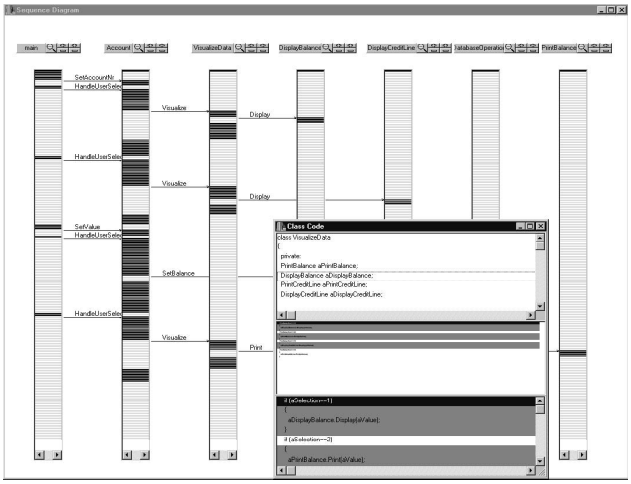


Figure 1. Sequence diagram to visualize program executions

3D versus 2D Visualization

As previously mentioned, software visualization of source code structure and execution behavior could consist of both static views and dynamic views [3,23,30] Compared to static views, dynamic views are based on information from the analysis of recorded or monitored program executions. During the recording of a program execution, a large amount of data may be collected. Although this is not a new problem, the rapid increases in the quantity of information available and a growing need for more highly optimized solutions have both added to the pressure to make good and effective use of this information [19]. This leads to new challenges in visualization, navigation and generally coping with the complexity of the dynamic information.

Three-dimensional visual representations are often suggested and presented as a solution to provide just this required extra space and resulting ease of use in navigation and abstraction level. While the advantages of adding a third dimension are initially quite obvious, these are realizable only if truly distinct and effective use is made of the added dimension. However, most of the current approaches are just transforming established 2D visualizing techniques into a 3D space. 3D software structure visualizations are still centered on creating standard call graphs within a 3D space. For example, the usage of 3D call-graphs does offer a greater working volume for the graphs thus increasing the capacity for readability. However, at the same time, they introduce undesirable effects that significantly affect the gain from the added dimension. Problems that might be introduced by 3D visualization techniques include significant objects being obscured, disorientation, and spatial complexity. To some limited extent, these issues can be resolved by 3D interaction techniques where the viewpoint of the 3D graph is actually within the graph structure; otherwise, the 3D visualization is limited to merely a 2D picture of a 3D structure. Ultimately, for 3D visualizations to be effective, other techniques than just mapping 2D models into the 3D space are required. These techniques have to introduce a more meaningful and abstract program representation that makes full use of the 3D environment and thus the engineer's natural intuition and perceptual skills [20].

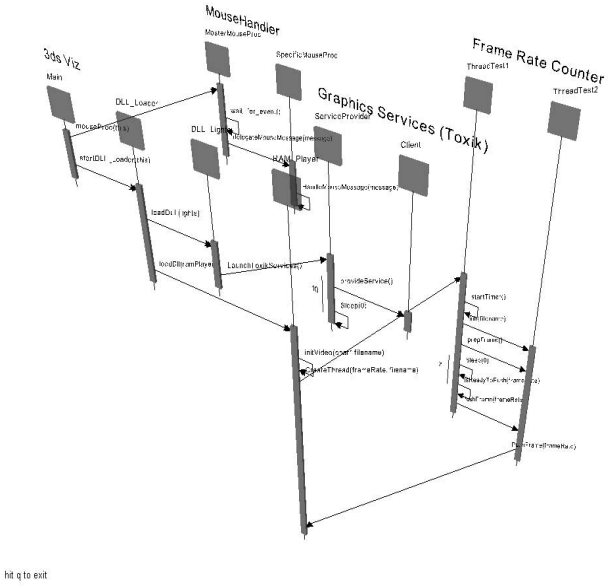


Figure 2. Mapping 2D sequence diagram notation into 3D space.

Program slicing

As mentioned earlier, given the large set of functions/outputs of a large program, in the process of program comprehension, programmers tend to focus on selected functions (outputs) and those parts of a program that are directly related to that function. Program slicing, a program reduction technique, allows one to narrow down the size of the source code of interest by identifying only those parts of the original program that are relevant to the computation of a particular function/output (Figure 3 and Figure 4).

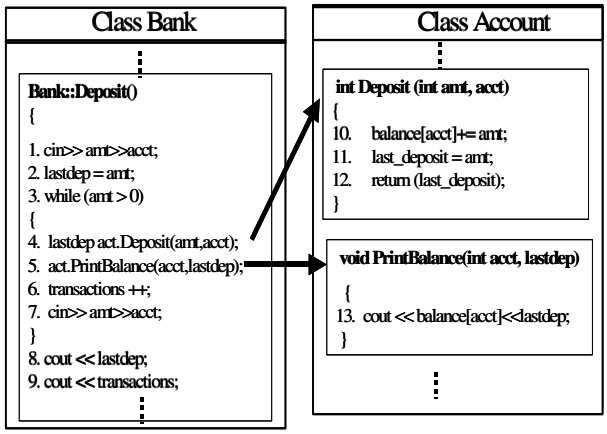


Figure 3. Sample program

Through reverse engineering, it is possible to derive data and control dependencies that can be utilized for algorithmic source code analysis.

The source code analysis provides users with additional insight in the dependencies and relationship among the different program artifacts. Static slicing [31] derives its information through the analysis of the source code. A static program slice consists of all statements in program *P* that may potentially affect the value of variable *v* at some point *p* [8,9]. That is, a static slice preserves a program's behavior with respect to variable *v* for all possible program inputs. Static slicing reduces the size of the original program, but frequently still leads to slices that are rather large and difficult to analyze. Slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance [8,9,21,31].

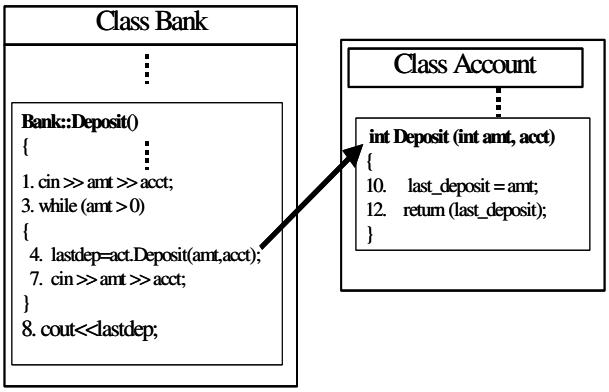


Figure 4. Slice criterion, *lastdep* at position 8

The strength of static slicing lies in the following areas: (a) the computation of a static slice is relatively inexpensive (compared to dynamic slicing) in that only the static analysis of the source code and no program execution is required, and (b) it is useful to gain a general understanding of the program parts that contribute to the computation of a selected function with respect to all possible program executions. However, static slicing also has some major drawbacks: (1) for programs containing conditional statements or dynamic language constructs like polymorphism, pointers, aliases, etc., static slicing has to make conservative assumptions with respect to their run-time contribution and their relevance for the slice computation, (2) based on its static nature, static slicing does not provide any information with respect to the analysis of the program execution, and (3) frequently, static slicing produces larger program slices than dynamic slicing algorithms.

Dynamic slicing can be regarded as a refinement of the static slice by only preserving a program's behavior for a specific program data input rather than all possible program data inputs [13]. Several different techniques for the computation of dynamic slices have been proposed, e.g., [13,21]. For programmers, it is often not feasible to comprehend the programs without observing a particular program execution. Dynamic slicing allows a programmer to focus on a particular program execution (program input), rather than all possible program executions. This is achieved by using data based on an actual program flow, which also leads to accurate handling of dynamic and conditional language constructs. Frequently dynamic program slices are smaller than the ones computed by static slicing algorithms. One of the major drawbacks of dynamic slicing, compared to static slicing, is that it requires the identification of relevant input conditions and a program execution for that particular input. The identification of input conditions and the recording of program executions require an additional system overhead. Figure 5 summarizes the trade-offs between static and dynamic approaches

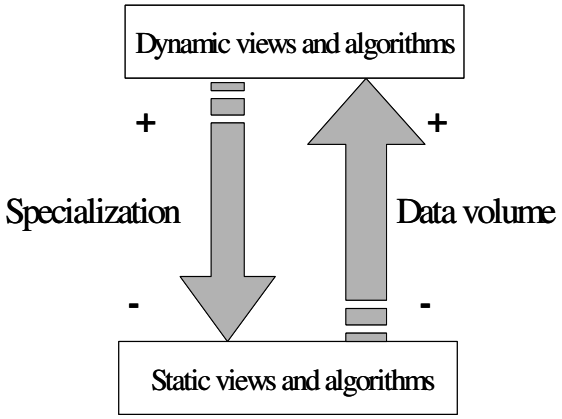


Figure 5. Dynamic versus static approaches

The dynamic approaches provide additional insights with respect to program structures and their executions, but at the same time they also create new challenges: (1) effective methods to visualize these large amounts of information, (2) development of "dynamic" visualization approaches that allow showing of the dynamic changes in the data. Static approaches, on the other hand, provide more generic, less detailed view on the data and its comprehension support.

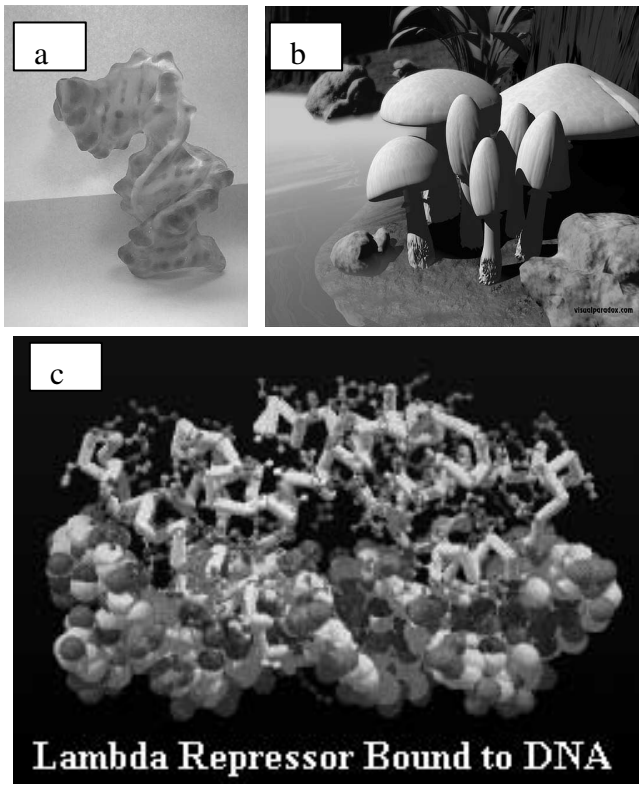
3. The metaball metaphor to visualize source code

One aid to improve the understanding of large programs is to reduce the amount of detail a programmer sees by using a higher level of abstraction to represent a program. Over the last decade, programs became larger and more complex, causing new challenges to the programmer in visualizing these complex and large source code structures. Different techniques and approaches have been developed and validated with users. Providing different levels of abstraction might not be sufficient since users might be still dealing with a large amount of information and data. Not every visualization technique is equally usable in displaying a particular dataset. The visualization technique might lack an appropriate navigation support or may not allow the effective reduction of the amount of information displayed through a choice of distinct views. The disadvantages of most of the commonly used high-level abstractions such as call-graph, UML class models, collaboration diagrams, etc. have already been discussed before.

Metaballs, also known as metablobs, soft objects, point clouds or more generally implicit surfaces, are a 3D object modeling technique which blends and transforms an assembly of particles with associated shapes into a more complex 3D shape, whose use is most suitable for animal and other organic forms. This technique models particles in 3D space, which have energy (strength) and have a well defined, parametrically controlled influence over the surrounding and neighboring particles. A metaball is defined by a so-called three-dimensional variable density field, radiating from a given center point. The value of the field can vary linearly with distance from the center, or in any other way expressible via a mathematical formula. For example, a field can have a negative density distribution, or even an eccentric distribution. A point on a metaball surface is constructed at all points in the field with the same density value, which is given by the modeler or derived from the modeling context.

If two or more metaballs are constructed in close proximity to one another so that they overlap, they coalesce and their fields are added in a process called fusion to produce a composite field, which is then evaluated to produce a composite surface. Metaball fields

can be transformed in a variety of ways to produce organic shapes necessary to represent, for example, the human form. Metaball surfaces are usually rendered as polygons. Metaballs have found extensive use in representing and visualizing complex organic shapes and structural relationships such as DNA, humans, animals, and other molecular surfaces. Extensions include grouping of particles, selective influence over other particles, hiding particles, etc.



a) DNA structure (www.scripps.edu/pub/olson-web)
b) Organic visual (www.visualparadox.com)
c) Molecular Images (The Scripps Research Institute.)

Figure 6: Traditional applications of metaballs

In this paper, we propose to apply the metaball metaphor[1] to visualize software entities and their dynamic influence over other entities. By defining visually intuitive mappings between the entities or parameters in the software slices, and metaball models, we can create a 3D virtual environment in which it is possible to walk around these entities, see what significantly influences the entity of interest, hide insignificant influences or zoom into entity-groups for understanding more detailed interactions. Mapping different entities in multiple views, say object oriented or functional, it becomes possible to use the same 3D metaphor to help understand software from different viewpoints. Mapping entity type to shape gives us the potential to visually differentiate, for example, free functions from member functions in a C++ program. Interacting with a complex metaball model, by moving an entity of interest closer to clusters of other entities and seeing the animated response from these further helps in visualizing the more dynamic aspects of a large software program. In short, the metaball metaphor gives us a constantly moving micro-universe of entities (metaballs), which can be dynamically altered to model program parameters and can be interactively walked through for various reverse engineering purposes, such as design evaluation, maintenance, testing, etc.

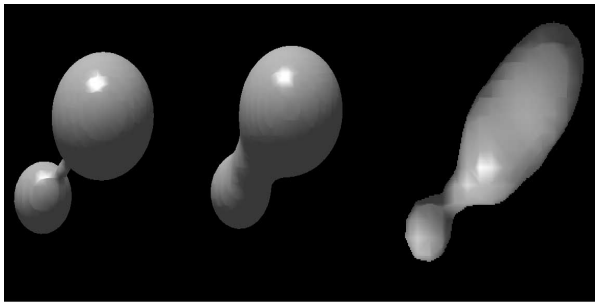
Mapping metaballs to program structures

Difficulties in understanding of OO programs are caused by the relationships that exist between classes and other parts of the program. Furthermore, in the case of very large programs, programmers face difficulties in comprehending the resulting highly complex diagrammatic representations of these relationships. Thus, UML based static and dynamic visualization techniques such as class models, sequence and collaboration diagrams can be applied for smaller software systems to provide an overview of the relationships in a program. However, for large software systems these diagrams will not provide adequate abstraction to visualize all the dependencies.

Particles in the metaball metaphor can be mapped to software structures, with blobs representing an object or a function (distinguished by different shapes for particles) that are created dynamically during a program execution. The potential energy surrounding blobs has traditionally been used to indicate the influence amongst blobs. This can be very intuitively used to visualize the strength of the coupling among program artifacts. For example, the number of function invocations performed among objects could be one of the parameters used to indicate the relationship (coupling) among the blobs (*cf* Figure 6).

The dimensions or size of a blob can be used to indicate a desired measure of the software entity, for example, and number of statements in a function. Blobs will be spatially located in clusters with spatial nearness indicating an identifiable association between the software entities that are mapped. We expect that a programmer preferred spatial configuration of entities is maintained in a persistent manner. This will enable the programmer to retain the visual association with software entities with very little effort.

[1] Other 3D metaphors that have been for software visualization include cone-trees [27], immersive VR [15], human agent [10] and world cities [12]. However each has its own specific comprehension objectives and its own advantages/disadvantages.



left: shows low coupling between two similar type entities
middle: shows high coupling between two similar type entities
right: shows medium coupling between two different type entities

Figure 7: Metaballs in visualizing software interaction

Dynamically changing associations are visually depicted by animating changes in appearance properties of blobs/connections. Collapsing particle clusters enables us to visualize abstraction at different hierarchical levels. Similarly, hiding or dimming blobs that are insignificant or less significant to a particular comprehension task gives us the ability to present only details that matter. Transparency and blob inclusion may be used to depict encapsulation. Table 1 shows the visual mappings that have been defined to use the metaball metaphor for software visualization.

Program Artifact	**Metaball property**
Software Entities	Particles
Entity Types	Blob Shape
Entity measure (eg. no. of statements in function)	Blob dimensions
Entity association	Particle clustering
Entity relationship (eg. coupling)	Energy potential amongst particles.
Hierarchic levels of abstraction	Particle collapsing
Different dynamic behavioral aspects	Blob colors, brightness, shininess, animated change in connections, etc.

Table 1. Visual Mappings

Clearly, the metaball metaphor provides us with a visually rich environment to depict entities in a software system along with visual techniques that enable mapping of software structure and dynamic behavior onto highly intuitive visual renderings.

Mappings for program slices

In what follows, we will discuss further extensions to the previously introduced metaball metaphor in visualizing software structures, based on program slicing techniques. For larger software systems, the metaball visualization technique faces similar problems as more traditional visualization techniques in its ability to scale for large amount of information and in providing guidance during the comprehension and analysis of dynamic dependencies that exist within large software system (*cf* Fig 8).

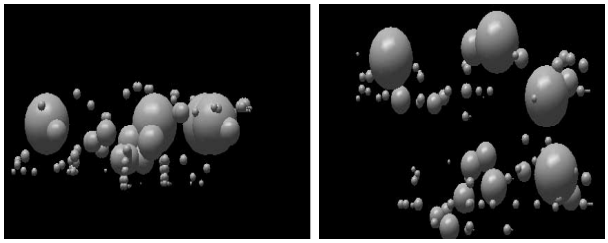


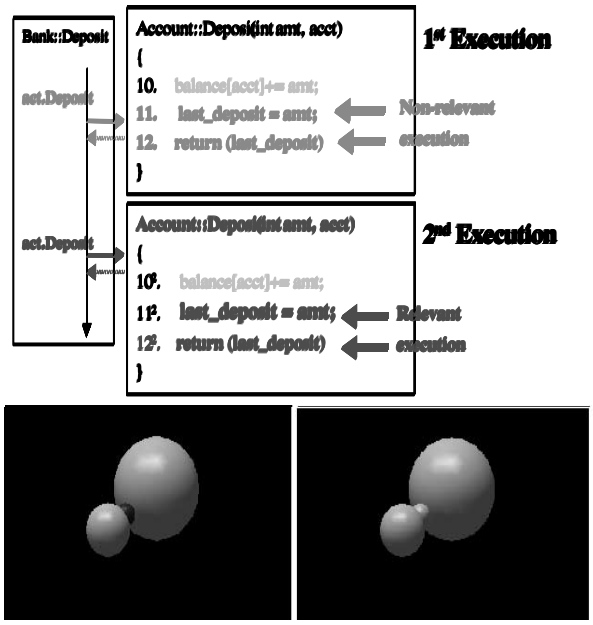
Figure 8. Two different views of a complex metaball environment

Not only can program slicing be used to identify dependencies with respect to a function of interest within the given program and its execution, but also, it allows for a reduction of the amount of information that has to be displayed. This enables a programmer to focus attention on those parts of a program that are relevant with respect to a particular function or feature. An object/function is included in the slice if at least one statement within this program artifact is included in a program slice. Similarly, a call relationship connecting line between two modules M_1 and M_2 (where M_1 calls M_2) is included in the slice if at least one "call M_2" statement inside of module M_1 belongs to the program slice. One approach to display a slice is by highlighting the modules (blobs) and call relationships that belong to the slice in the original metaball diagram, showing the complete program. Another approach is to display a metaball sub-diagram that is constructed from the original metaball diagram by hiding all modules (blobs) and their calling relationships that do not belong to the slice. The metaball metaphor can be further extended to visualize slicing related information through the introduction of different types of shading, texture and lighting techniques. Shading can be used to indicate, for example, the percentage of statements included in the slice. Focused lightning can be used to create a focal point for indicating the current execution position or to highlight objects that gained influence during a step-wise program re-execution.

Enhancing dynamic views through the notion of relevancy

The notion of relevancy is derived from the computation of a dynamic slice. Relevancy is based on the fact that it is possible for a statement that is part of a slice to be executed several times, however only a subset of these executions might be relevant to the computation of a selected function. In other words, it is possible for one action of call *X* to be relevant to the computation of the selected function but a different action of the same call *X* in the same execution trace is not relevant to the computation of this function. In many situations, a

programmer may be interested in analyzing only program executions (execution of the dynamic slice) that are relevant to the computation of the function of interest. Figure 9 shows this visual for the slice *last_dep* (shown in Figure 3). Only the second execution of the member function *Deposit* will be relevant, hence the second execution overwrites the previous value of the variable *last_dep*. Therefore, the notion of relevancy allows for reduction in the information complexity of the execution trace to be observed.



Coupling in the right picture is more relevant than in the left picture

Figure 9. Relevancy and its visualization

Another program slicing extension is the concept of influencing program artifacts. Influencing program artifacts allow a user to identify those program parts that currently influence a variable/function of interest. For a programmer, it is almost impossible to determine during the analysis process which program artifacts are currently influencing the computation of a particular function. The concept of an influencing program artifact is similar to the concept of the relevant program artifact based on dynamic slicing related information that is normally discarded after a slice computation. The difference between relevant and influencing artifacts is that relevant artifacts identify program executions that are relevant for the computation of a particular function. Influencing artifacts, on the other hand, provide information about which program artifacts are influencing at a current execution position in the computation of a selected function. One possible visualization approach is to highlight the influencing parts and dim the rest.

Additional visualization enhancements

Another approach to reducing the amount of data is to allow the user to select the granularity of a particular view. For a visualization technique to be scalable to represent large amounts of data, the metaphor has to support techniques such as collapsing, hiding, and expanding parts of the diagram, therefore giving the user the ability to select the view granularity and consequently the amount of information that has to be displayed.

Clustering/grouping

Frequently, it is advantageous to reduce the number of visible elements at any time. Limiting the number of visual elements to be displayed both improves the clarity and simultaneously increases performance of layout and rendering [14]. Various "abstraction" and "reduction" techniques have been applied by researchers in order to reduce the visual complexity of graph like structures. One approach is to perform clustering.

Clustering can be described as the process of discovering groupings or classes in data based on chosen semantics. Clustering techniques have been referred to in the literature as *cluster analysis, grouping, clumping, classification, and unsupervised pattern recognition* [14]. The use of the semantic data associated with metaballs to perform clustering could be termed *content–based* clustering. Content–based clustering can yield groupings that are most appropriate for a particular application and can even be combined with structure–based clustering. Content–based clustering requires application–specific data and knowledge. It is important to note that clustering can be used for functions such as filtering and search. In visualization terms, filtering usually refers to the de–emphasis or removal of elements from the view, while searching usually refers to the emphasis of an element or group of elements. Both filtering and search can be accomplished by partitioning elements into two or more groups, and then emphasizing one of the groups.

We apply clustering of metaballs to provide users mainly with an option to summarize and analyze a program structure or a program execution. The clustering techniques are less applicable in visualizing dynamic changes (in particular for large software systems), because of navigation or orientation issues. Grouping, and therefore, changing the layout dynamically will distort a user's ability to correlate the clusters with a particular program structure or content.

Applying the metaball metaphor in program comprehension

One basic requirement to enhance acceptance of visualization techniques is to provide programmers with navigation techniques that provide easy navigation and a clear perspective regarding how the current program part under investigation relates to the overall structure of the

software system. Enhancing the maintainability requires software developers and designers to enhance the quality and the design of existing systems. The metaball metaphor can be applied to provide programmers with an intuitive technique that visually guides programmers during typical maintenance tasks. Maintenance tends to degrade the structure of software, ultimately making maintenance more costly. The longer object oriented systems are in use, the more likely that these systems have to be maintained. They have to be changed to reflect new features (perfective maintenance), fix identified defects (corrective maintenance), and adjusted for a changing environment (adaptive maintenance). In what follows, we describe the potential use of metaballs in connection with program slicing and its application in software maintenance.

Design evaluation

Software systems have to be flexible in order to cope with evolving requirements [6,16,29]. Although good software engineering practice encourages programmers to plan for future modifications, not every future design change can be predicted. User requests for changes are often a consequence of using the system after delivery. Well-designed modules should exhibit a high degree of cohesion and a low degree of coupling, such that each module addresses a specific, well-defined sub-function from a system structure view. As we have already seen, the metaball visualization metaphor allows one to identify and analyze in an intuitive way the coupling among different program artifacts. In combination with program slicing, the metaball metaphor enables the creation of several logical views of the system, representing the slice specific couplings that exist in a particular system. A hotspot approach can be applied to identify units of viability, called hotspots, by highlighting those parts of the diagrams that do not meet a user pre-specified quality criteria. After analyzing a program execution, the system will display hotspots to visualize locations in the current design where the system does not meet the selected design quality (with respect to coupling interaction).

Testing

Program slicing has already been shown to be useful for software testing [8,9,13,31]. The metaball visualization techniques can further provide a very intuitive high-level interface in identifying the test coverage of a particular modules/classes. In particular, for regression testing performed after modifying a program part, the metaball approach combined with program slicing can help to improve the testing process. The reduced program complexity/size of the slice in combination with the metaball visualization allows for further reduction in the time and effort required to test program parts. Additionally, by applying clustering and grouping of the metaballs in the system that are highly coupled with each other, problem areas can already be identified at a very high level of visual abstraction.

Debugging

For the visualization of long program executions, one has to apply different levels of granularity within the metaball visualization. A metaball can represent a variety of programming constructs, like a file, a class/module a loop, etc [30]. It is essential that the user have the option to collapse and expand each node to select an appropriate view. Moreover, the color and/or the shape of the nodes could be used to visually help separate types of files (headers, sources). Metaballs can be used in connection with program slicing to re-execute a program (based on a record of program execution) and to highlight the objects that are influencing the computation of a variable of interest at a current execution position. Therefore, the user can dynamically identify those parts of a program execution that are influencing the computation of a selected variable at a current execution position.

Performance Analysis

In typical performance analysis tools, program executions are analyzed and associated to resource requirements. During performance analysis it would be desirable not only to identify program artifacts that were executed but also their frequencies. For a programmer, it is more important to distinguish between executions that are relevant and those executions that are not relevant. Metaballs can be applied to guide visually the analysis of programs and their executions, providing a summary view (and analysis) of a program execution. By combining the size property and the notion of relevancy, one can not only visually identify places of those executions that have no or a very low relevancy with respect to a particular dynamic slice, but also it is possible to cluster and group these objects based on their relevancy. This information not only provides guidance during performance analysis but also guides the user during the process of locating places with non-optimized source code.

4. Conclusions

It is a well-known fact that a major share of systems development effort goes into the comprehension of large systems and their source code, about which we usually know usually very little. The large and complex programs developed and maintained in current software environments are the ones that can most benefit from the visualization and source code analysis techniques presented in this research. Our paper presents a novel approach of applying the metaball metaphor to visualize source code and source code analysis information. Specifically, in combination with program slicing, this technique provides a rich, powerful and intuitive method

of visual presentations that can considerably enhance and speed program comprehension. Along with program slicing it not only allows for a reduction of the information to be displayed, but also enable us to provide additional source code insights that can be applied for a variety of source code based comprehension tasks (e.g. debugging, testing, performance analysis, etc.).

Given the complexity of software and the different problem solving characteristics of programmers, it is now well recognized that there is unlikely to be any one single visualization metaphor that can be considered most optimal for software visualization. Instead, different metaphors may be better suited to specific program comprehension purposes and for particular types of analyses results. In our opinion, the metaball metaphor is rich and has the potential to be a very good candidate for a number of software reverse engineering tasks. This is in no small part due to it being highly effective in visually capturing the relationships between software entities such as coupling, relevancy, and influence. Relationships that play a vital role in virtually all program comprehension and reverse engineering tasks. While there is a large body of powerful software available for this technology, both commercial and public domain, all of this software is tailored towards application in domains such as molecular modeling, animation, and electronic gaming. Our use of metaballs is similar but not identical to those domains. It is important to develop metaball visualization software specifically tailored to producing the kind of visuals and 3D interactions that have been elaborated upon in earlier sections. With such software, it should be possible to experiment with real large software systems to acquire feedback that could then be used to further refine software visualization methods using the metaball metaphor.

Acknowledgements
We also would like to thank David Cunningham for his support in developing the 3D sequence diagram.

References

1. 3D ARK, "3D Related **Software** List ", *http://www.3dark.com/resources/products/softwarelist.htm*
2. Ball T., Eick Stephen G., "Software Visualization in the Large". *IEEE Computer* 29(4): 33-43 (1996).
3. Baker, Marla J. and Eick, Stephen G., "Space-Filling Software Visualization". In *Journal of Visual Languages and Computing*, vol. 6, 1995, pp.119-133.
4. Blinn, J. F., *"*A Generalisation of Algebraic Surface Drawing*", ACM Trans. Graphics,* Vol. 1, No 3, July 1982, pp 135-256
5. Bloomenthal, J., "Polygonization of Implicit Surfaces"*, Computer Aided Geometric Design*, Vol. 5, No 4, November 1988, pp 341-355
6. Demeyer S., Stéphane Ducasse and Michele Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization", In *Proceedings of WCRE'99*, IEEE, pp. 175-187, 1999.
7. Favre J.M., "G*SEE*: a Generic Software Exploration Environment", *9th International Workshop on Program Comprehension (IWPC'2001),* Toronto, Canada, May 2001, pp. 233-244
8. Harman M., Hierons R. M., Danicic S., Laurence M., Howroyd J. and Fox C, 2001, "Pre/Post Conditioned Slicing", *IEEE International Conference on Software Maintenance (ICSM'2001)*, Florence, Italy
9. Harman M. and Danicic S., "A New Algorithm for Slicing Unstructured Programs", *Journal of Software Maintenance*, 10(6):415-441, Nov/December 1998.
10. Hopkins, J. and Fishwick, P. A., "A Three-Dimensional Human Agent Metaphor for Modeling and Simulation", *Proc. IEEE*, 89(2), 2001, pp 131-147.
11. Knight C., Munro M., "Visualising Software - A Key Research Area", *Proc. of the Int. Conference on Software Maintenance*; ICSM'99, IEEE Press, 1999.
12. Knight C., Munro M., "Visualising the non-existing", *IASTED International Conference: Computer Graphics and Imaging*, Hawaii, USA. 2001.
13. Korel, B., "Computation of dynamic slices for unstructured programs", *IEEE Transactions on Software Engineering*, 23(1), pp. 17-34, 1997.
14. Kreuseler, M. and Schuman, H., "Information visualization using a new Focus + Context Technique in combination with dynamic clustering of information space". *Proc. of the ACM Workshop on New Paradigms in Information Visualization and Manipulation,* Kansas city, 1999, pp. 1-5.
15. Maletic, J.I., Leigh, J., Marcus, A., Dunlap, G., "Visualizing Object-Oriented Software in Virtual Reality", *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, Toronto, Canada, May 12-13, 2001, pp. 26-35.
16. Mayrhauser A., A. M. Vans, "Program Understanding Behavior During Adaptation of Large Scale Software", *Proceedings of the 6th Intl. Workshop on Program Comprehension*., IWPC '98, pp. 164-172, Ischia, Italy, June 1998.
17. Michaud J., Storey M.-A.D. and Muller H.A., "Programs, Integrating Information Sources for Visualizing Java", *Proc.s of the Inter. Conference of Software Maintenance (ICSM'2002)*, Italy, 2001.
18. Nielsen, Jakob, "2D is Better Than 3D", AlertBox, http://useit.com/alerbox/981115.html, 1998.
19. Pirolli, Peter and Card, Stuart K. and Van Der Wege, Mija M., "Visual information foraging in a focus + context visualization". In *Proceeding of the ACM Conference on Human Factor in Computing Systems (CHI-01)*, Seattle, 2001, pp. 506-513.

20. Price B., Baecker R., and Small I., “A principled taxonomy of software visualization”, *Journal of Visual Languages and Computing*, 1994, pp 211-266.

21. Rilling J., *“*Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge*”, 8th IEEE Working Conference on Reverse Engineering (WCRE 2001*), Stuttgart, Germany, October 2001, pp. 157-165.

22. Rilling J., Seffah A.*, “*Enhancing the Usability and Learnability of Software Visualization Techniques through Task Wizards and Software Agents*”, Proc. of Intern. Conference on Imaging Science, Systems, and Technology (CISST'2001)*, Las Vegas, June 2001.

23. Robertson G. G., Mackinlay J. D., and Card S. K., “Cone trees:animated 3D visualizations of hierarchical information”, *Proceedings of CHI’91 Conference on Human Factors in Computing Systems,* pages 189–194, 1991.

24. Sanlaville R., Favre J.M., Y. Ledru, "Helping Various Stakeholders to Understand a Very Large Software Product", *European Conference on Component-Based Software Engineering,* Sept. 2001.

25. Shneiderman, Ben, “Tree Visualization with Tree-Maps: A 2-D Space-Filling Approach”. In *ACM Transaction of Computer-Human Interaction*, vol. 11, no. 1, 1992, pp. 92-99.

26. Shneiderman, Ben, “Designing the User Interface, *Addison-Wesley, 3rd edition*, 1997.

27. Stasko, John, Domingue, John, Brown, Marc H. and Price, Blaine A. (editors), “Software Visualization: Programming as a Multimedia Experience”, *MIT Press*, Cambridge, MA, 1998.

28. Storey M.-A., Fracchia F. and Müller H.., “Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration, *Journal of Software Systems*, special issue on Program Comprehension, v 44, pp.171-185, 1999.

29. Van Deursen A., Kuipers T.,” Building Documentation Generators”, In Proceedings International Conference on Software Maintenance (ICSM'99), IEEE Computer Society, 1999, 40-49.

30. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., “Visualizing Dynamic Software System Information through High-level Models”, *Proceedings of OOPSLA'98, pp. 271-283, SIGPLAN Notices 33(10)*, October 1998.

31. Weiser M., “Program slicing”*, IEEE Transactions on Software Eng., SE-10, No. 4*, 1982, pp. 352-357.

32. Wyvill, G. and McPheeters, C. and Wyvill, B., "Data Structure for Soft Objects"*, The Visual Computer*, Vol. 2, No 4, August 1986, pp 227-234.

33. Wyvill, G. and McPheeters, C. and Wyvill, B., "Animating Soft Objects"*, The Visual Computer*, Vol. 2, No 4, August 1986, pp 235-242.

34. Wyvill, B. and Wyvill, G., "Field Functions for Implicit Surfaces"*, Visual Computer*, Vol. 5, 1989, pp 75-82.