

Dynamic Component Program Visualization

Ludger Martin*

Anke Giesl*

Johannes Martin⁺

*Darmstadt University of Technology
Department of Computer Science
Wilhelminenstr. 7

64283 Darmstadt, Germany

{lumartin—giesl}@gkec.informatik.tu-darmstadt.de

⁺University of Victoria
Department of Computer Science
PO Box 3055

Victoria, BC V8W 3P6, Canada

jmartin@csc.uvic.ca

Abstract

Dynamic program visualization, i.e. the visualization of the runtime behavior of a program as opposed to the static structure of its source code has been investigated for various kinds of programs, usually on a fairly low level. The focus has been on control flow in procedural programs or on the communication between objects in object-oriented programs.

In this paper, we explore the usefulness of dynamic program visualization for component programs. We discuss how to visualize component-programs and motivate the need for a proper visualization of the communication among these components. We present a three dimensional visualization and show the integration into the HOTAGENT component development environment.

1. Introduction

Component technology is gaining importance for the development of various kinds of software. Johnson [7] explains that building applications using components is very efficient. The reuse aspect of components is obvious. In addition, applications built from components are not only easier to create but they are more reliable and maintainable.

Program *analysis* is a means for understanding and exploring the structure and behavior of a program. It can focus on two different facets of this problem: static analysis uses the source code of a program whereby dynamic analysis investigates the runtime behavior of a program. Various kinds of graphs are used as *visualization* techniques to show results of the analysis. In static analysis, source code artifacts such as data types, functions, variables and their relationships are visualized, whereas in dynamic analysis, scenario, sequence and state diagrams are shown. The information gathered during the initial analysis steps is usually

fairly low level, so the visualizations become rather large and dense. Hence, the graphs are clustered to raise their level of abstraction and obtain a more meaningful visualization.

Design documents such as UML sequence diagrams visualize the desired runtime behavior of a program but they do not show its actual behavior. There is still the need to check whether the implementation really follows the design. Formal verification methods to check this would be best, but they are costly. Program visualization offers a convenient way to discover discrepancies between the design and the actual implementation. By visually presenting the actual program analysis results, it allows a comparison with the desired behavior. However, since program analysis techniques are only infrequently taught to software engineers and analysis tools are often separate from development tools, developers avoid using these tools, and discrepancies go unnoticed. But if program analysis tools were integrated into development tools, developers would be encouraged to use them.

In this paper, we explore program analysis and visualization techniques for component programs. We focus on *dynamic* analysis of component programs and propose meaningful three dimensional visualizations for the analysis results. We present the dynamic analysis tool HOTAGENT VISUALIZE which is implemented as part of the HOTAGENT component development environment. It takes a two-dimensional visual component program that has been constructed with HOTAGENT ASSEMBLY and extends it to the third dimension. The third dimension represents the time and allows the whole measured runtime behavior of the program to be visualized in one rotatable 3D-Figure. By keeping to the structure of the two-dimensional visually constructed program, the developer does not have to rethink the correlations between the components. Through the tight integration with the rest of the environment, we aim to achieve high acceptance by developers that are al-

ready familiar with the HOTAGENT environment. This new approach, i.e. the visualization of a program's execution based on the static visualization given by a visually composed program is a worthwhile extension for every component development environment.

The HOTAGENT [11] development environment is presented in [13]. It is a component framework to construct agents for electronic commerce. The framework provides a special set of components to facilitate the construction of agents. The agents could undertake the task of doing routine work, such as analyzing electronic documents, extracting important information, and composing an answer or a form. The HOTAGENT development environment offers different tools to develop components, to test components, and to compose agents using components.

In Section 2, we discuss the basic concepts of components, analysis, and visualization. In Section 3, the component model used by HOTAGENT is presented. The component program visualization tool HOTAGENT VISUALIZE, which is based on dynamic program analysis, is presented in Section 4. Other parts of the HOTAGENT development environment are described in Section 5. Section 6 presents some related work. The paper concludes with a summary and a discussion of future work in Section 7.

2. Basic Concepts

To discuss dynamic component program visualization, it is necessary to first define the term component. Later on, dynamic analysis of component programs and program visualization can be discussed.

2.1. Components

Szyperski [23] defines: "A *component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

According to Lüer and Rosenblum [9, 10], a component should have a public and a private part. A *component* implemented by an object oriented programming language consists of several classes which are encapsulated by a set of well defined interfaces. The *interface* is the sole public part of the component. All other classes of the component are private. Therefore, no knowledge about their structure and behavior can be assumed for dynamic analysis. The interfaces ensure the *communication* between all the components.

In our component model, every component provides *entrances* and *exits*. Communication between components is possible only through these public entrances and exits. Using the notation of Lüer and Rosenblum [9, 10], the en-

trances could be called *requires ports* and the exits *provides ports*. But in contrast to the model of Lüer and Rosenblum, an entrance can not only receive data but also produce a result depending on the component. An exit is activated if a component has to show any change by providing the appropriate data. To compose components, exits are connected with entrances.

2.2. Dynamic Analysis

Dynamic analysis is used to describe the runtime behavior of a program. During the execution of a program, information on the communication between functions, objects, or components is gathered. In many cases, dynamic analysis is also used for testing software.

In contrast to static analysis, where only the source code of a program is analyzed, dynamic analysis starts during the execution of a program which means that at least a running program is needed, even though it does not necessarily have to be correct. While static analysis can also be performed on source code that is not even compilable, dynamic analysis needs a compiled program.

Since dynamic analysis is performed while the program runs, it is important that the analysis does not slow down the program significantly to avoid a change of the program's behavior. Reiss [19] explains that an inexpensive data collection technique is needed to ensure that the analysis does not have an impact on the program being analyzed.

Souder *et al.* [20] propose to include a filter mechanism in the analysis tool. During the analysis, it is possible that a lot of data accrues. All of that data has to be processed but only a subset of it may be of interest, e.g., if only a part of a system needs to be analyzed. To limit the amount of information generated during the dynamic analysis of a program, a tool should allow the specification of rules to configure exclude and include filters.

Since we understand a component as a block with entrances and exits [14], it is practicable to record the data exchanged between exits and entrances of the components. This sequence reflects the communication between the components during the execution of the program. Using this set of data points, it is also possible to find a failure by comparing it with a desired sequence.

2.3. Program Visualization

After analyzing a program, it is necessary to process the data and to find a proper visualization.

Myers [17] categorizes program visualization. The visualization scope defines the kind of data to visualize, e.g., program code, data, or algorithms. The second category is the kind of visualization, either static or dynamic. Visual

development environments already provide static visualization.

Visualization must not be confused with visual programming. Visual programming means to construct a program visually whereas program visualization concerns the presentation of a program, no matter how it was constructed. Visualization is helpful for understanding a program. Software engineers who are not familiar with a given program can get a better impression of its functionality. Especially in object-oriented and component-oriented programming, it is hard to understand the correlations within a large program and the program's runtime behavior while just using the source code. Usually UML is used to visualize programs, but as mentioned earlier, often the diagrams are not correlated with the actual coding or the real runtime behavior. It is desirable to present the actual relationships in a program in a graphical way. As it is important that the visualization is clear, the information needs to be presented in an abstract way and the presentation must not be overloaded with details. Souder *et al.* [20] add that different views are necessary. Each view can present different information in a clear way, and so multiple views can present a good overview of a program.

Program visualization is not a new technique. Wiggins [24] investigates several simple and more complex techniques, starting from flowcharts, Nassi-Scheidemann diagrams, movies, to pretty-printing. There are also tools which can automatically create a visualization using these techniques. In some cases, animation can be helpful for a better understanding of concepts.

According to Reiss [19], past visualization tools posed problems in that they were difficult to use or unable to handle large volumes of data. To solve these problems, a visualization tool needs to be able to handle a large amount of data or provide proper filtering techniques to reduce the data. The presented visualizations must not be difficult to handle, but at the same time present sufficient data.

To visualize the runtime behavior of component programs, we record the communication between the components during runtime, filter the information if desired and build a three-dimensional communication graph afterwards. Two dimensions are needed to represent the connections between the components visually. The third dimension represents the time, strictly speaking the runtime. This allows a comparison between the measured runtime behavior and the static program in a very clear way. After a short description of our component model, we present this idea and its implementation in more detail in chapter 4.

3. HotAgent Component Model

The HOTAGENT development environment is not based on one of the well-known component models like EJB (En-

terprise Java Beans), CCM (CORBA Component Model), or COM+ (Common Object Model). Instead, a small component model is used. The advantage of a small model is that it is easier to use with a lower overhead, which allows a user to focus on component usage. Inter-component communication only relies on a *single common interface* that each component has to implement. It is also based on an event mechanism which is common to all component models mentioned above. It is not hard to transfer the small model to a more complex one.

The component model is implemented in VisualWorks Smalltalk. A component relies only on one common interface to communicate with other components. The interface is the sole public part of all components. All other classes that are a part of each component are private. This approach enables all components to communicate with each other independently of their tasks.

Lüer and Rosenblum [9, 10] demand self documentation of components. The HOTAGENT component model offers three kinds of documentation. The first is a short unique *descriptive name* of the component, e.g., `eMail`, which is stored in a global name space. To have the opportunity to sort the components into different categories, every component has one *category name*, e.g., `data management`. This is also used to locate components in an easy way. If a component is chosen, it is necessary to check that it is the right one for the task at hand. To do this, a component has a *descriptive text* to give an impression of its features.

If an instance of a component is created, it is important to assign a unique *instance name*. This name is only known in the composed agent and can be used to identify the component during runtime.

The interface of a component ensures the communication between the components. Every component provides so called entrances and exits. An *entrance* can receive data and produce a result, depending on the component functionality. An *exit* is activated if a component has to notify its environment about a change in its internal state. It supplies all necessary data and can process a possible result. Since Smalltalk is an untyped programming language, the data can be of any type. It is only possible to communicate using these public entrances and exits. To compose components, exits can be connected with entrances. Similar to the component, every entrance and exit has a *name* and a *descriptive text* to allow self documentation. It is important to describe the data provided by an exit and needed for an entrance.

4. HotAgent Visualize

HOTAGENT VISUALIZE is part of the HOTAGENT system. It visualizes the measured runtime behavior of a component program that has been visually constructed with the

HOTAGENT ASSEMBLY tool. The visually constructed program is two dimensional. Components are represented by icons. The communication channels between them, i.e. the connections between exits and entrances, are shown as colored arrows with short descriptions. HOTAGENT VISUALIZE uses this figure to extend it to the third dimension which represents the runtime. The components become cuboids and the recorded communication is shown by placing descending arrows between them. The height of a communication arrow represents the time the communication occurred. As the cuboids can hide each other in the picture a navigation allows a good view of all details.

In subsection 4.2 the HOTAGENT ASSEMBLY tool is described in more detail. A description of the dynamic analysis process used to check how the connections represented by arrows are used during runtime is given in the second subsection 4.3. HOTAGENT VISUALIZE, which finally constructs the 3D figure showing the measured communication, is presented in the last subsection 4.4. But first we introduce the example we use to demonstrate the HOTAGENT VISUALIZE tools.

4.1. Example Agent

The Medical Advisory Service agent [2] is an agent which receives patients' symptom descriptions and analyzes them for matches with symptoms of known medical diseases. If the agent finds a disease it answers with a proposal for a drug.

Patients are intended to communicate with the agent via email. For simplification of the prototype program we built a user interface that simulates the email handling. The user types the email-text in a dialog box and the "sending" of the "email" is done by pushing a button. The dialog box is also used to show the answer-email of the agent.

Figure 1 and 2 display the user interface of the running program of the Medical Advisory Service agent. Figure 1 shows a patient's question and Figure 2 the answer received from the agent.

4.2. Visual Programming

The HOTAGENT ASSEMBLY editor, presented in [12], can be used for application composition. Agents can be composed visually using predefined components.

The Medical Advisory Service agent including the user interface [2] has been visually constructed with this editor. The built visual program is shown in Figure 3. It consists of eight components. Two user interface components build the visible part of the agent: the dialog box and the push-button. The other six components are invisible. The control component is the central steering mechanism of the agent. The text analyze component investigates the email and

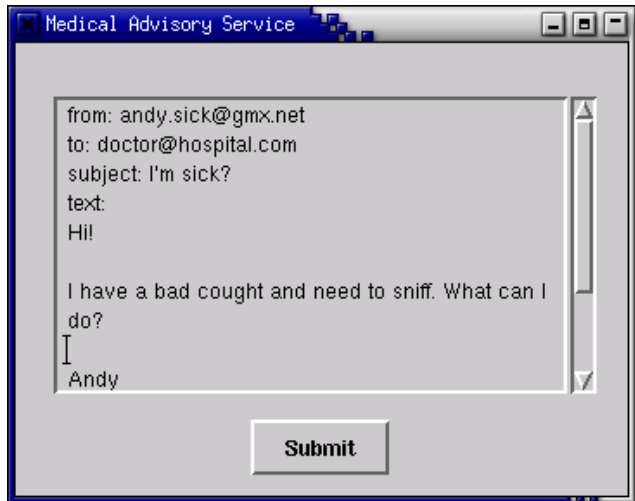


Figure 1. Question to the Medical Agent

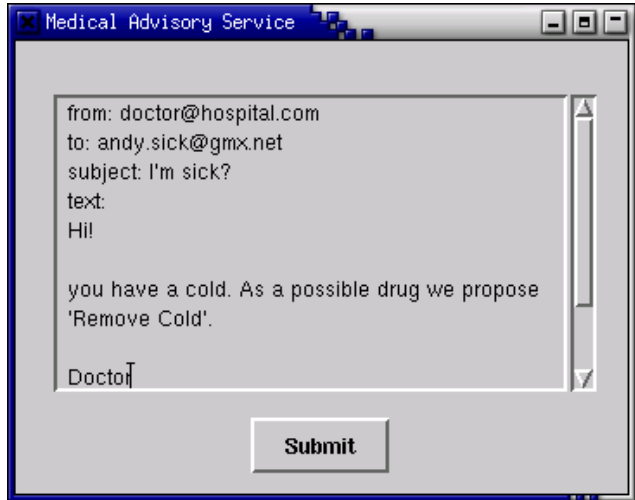


Figure 2. Medical Agent's answer

searches for known symptoms. The symptom database (ISDB) is queried to find a disease and the drug database (MIDB) is queried for a potential drug to cure the patient. The remaining two components serve for email-handling.

The first thing to do when designing an application is to choose a position for a new component on a workspace. Components can be classified into two different types: (i) visible components representing an element of a graphical user interface (GUI) and (ii) invisible components like a database without any visual representation. The workspace of the HOTAGENT ASSEMBLY editor consists of two parts: the main part that accepts only invisible components and a frame on which visible and invisible components can be placed.

Once the developer has indicated a position on the

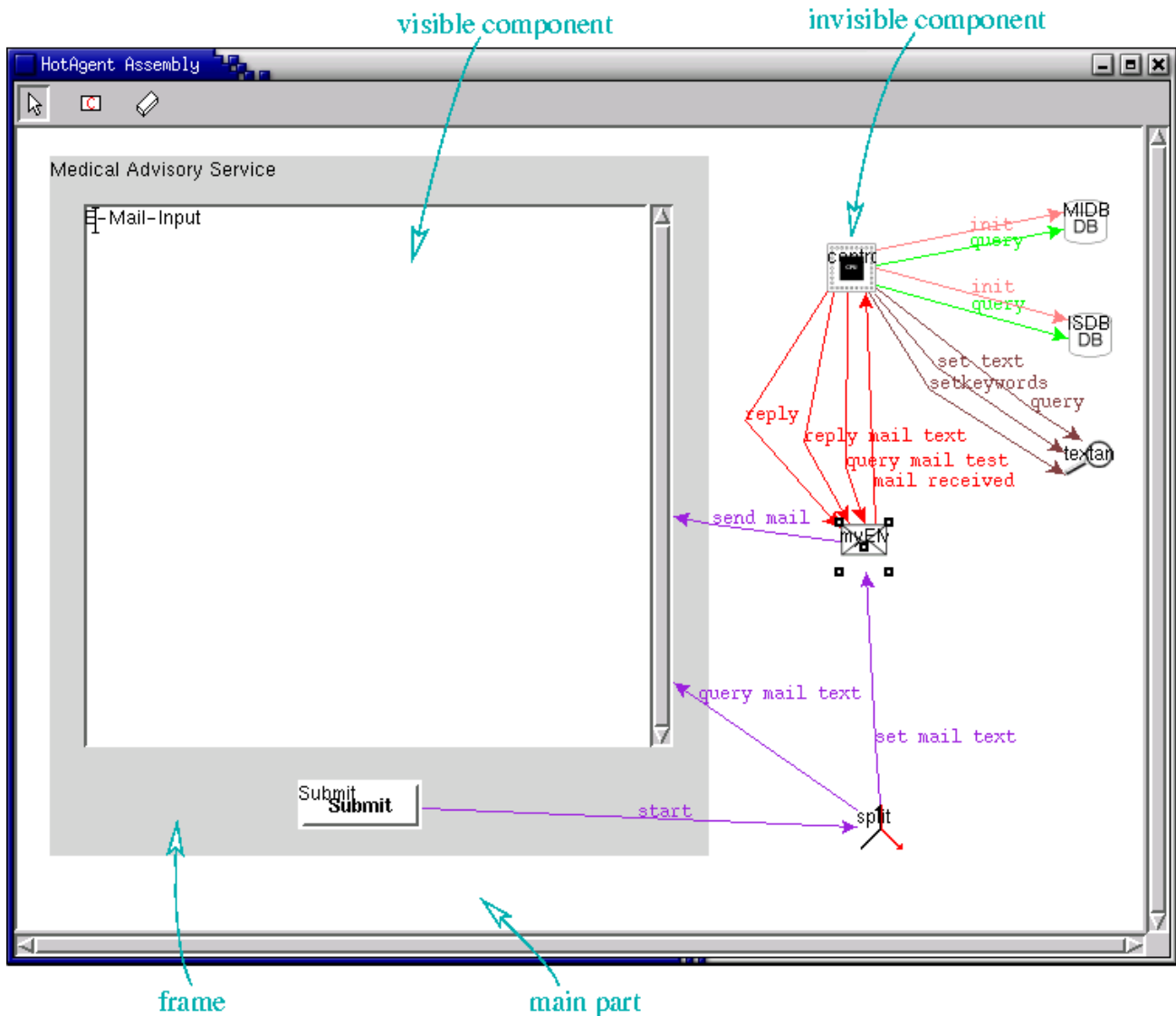


Figure 3. HOTAGENT ASSEMBLY editor

workspace, a dialog window for the selection of a component appears. When all required components have been placed onto the workspace, it is necessary to connect them. In order to create a connection, a source component needs to be selected. Once this has been done, a handle of the selected component can be dragged onto the target component. After finishing the drag, another dialog window pops up which can be used to select the corresponding exits and entrances. In addition, connections can be labeled and colored.

Connections are displayed using different colors, as shown in Figure 3. In this example, the developers used the secondary notation *color* to distinguish their purposes. They chose red for the communication between the control

and the email component, purple for the communication between user interface components, pink and green arrows between controller and data bases (pink for the initialization and green for the query during runtime), and brown arrows between controller and text analysis. Every connection has a short text to describe its task.

4.3. Dynamic Analysis

Dynamic Analysis starts with the gathering of data. The visualization scope (see above, Section 2.3) for HOTAGENT VISUALIZE is to show the communication between components. Therefore, the communication between all components must be recorded.

It is infeasible to make extensive changes to a program's

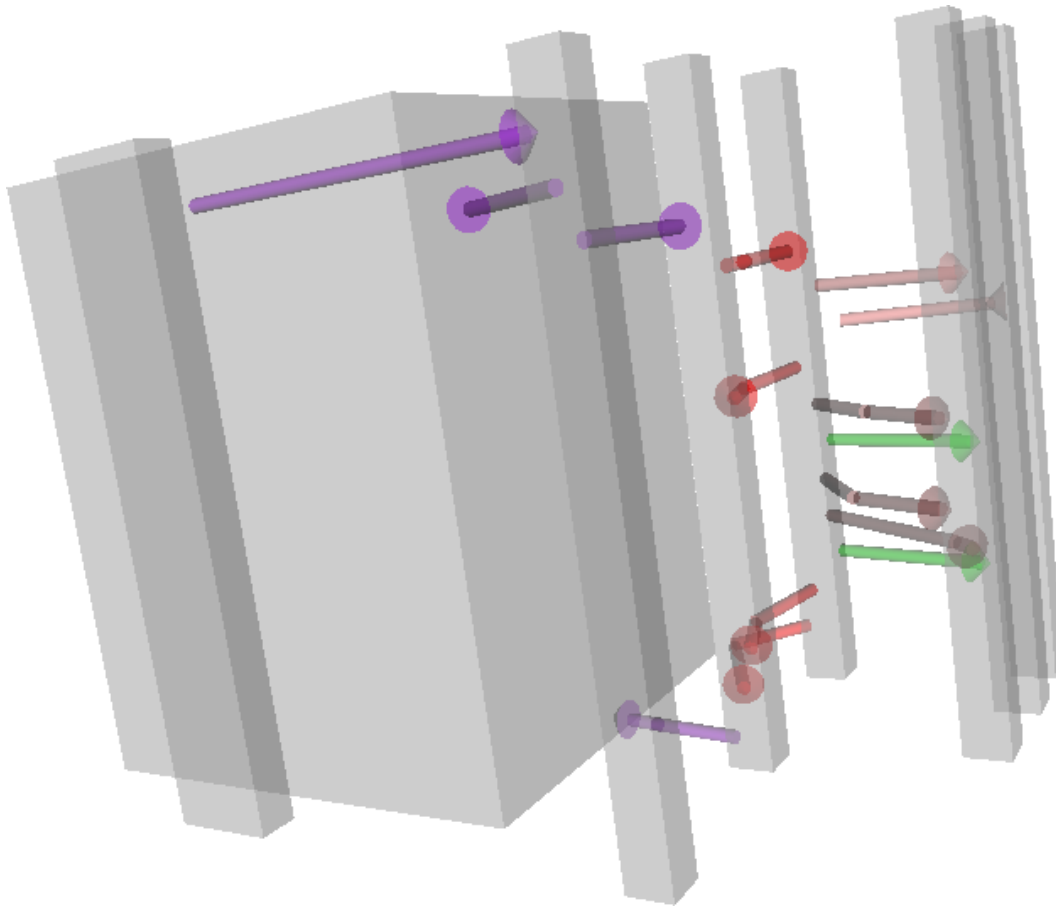


Figure 4. Program Visualization

source code just for the purpose of gathering data for dynamic analysis. On the one hand, it is difficult to ensure that the changes are made in the right places. On the other hand, the changes need to be removed again when the code is to be deployed. The advantage of the HOTAGENT component model is that all components are derived from one class, so changes need to be made to this class only. In addition, communication is managed by only one method. Thus, only this method needs to be changed. This reduces the risk for additional failures and prevents the application from being slowed down very much. Some programming languages offer the ability to compile single classes or methods. As we use Smalltalk, we can benefit from this feature. To perform the dynamic analysis, we replace the communication method by one that records all messages sent besides handling the communication tasks and compile it. Once the dynamic analysis is finished, the original method is used again.

The data we gather during the analysis consists of the source component name, its exit name, the target component name, its entrance name, and the transported data. This

set of data is written in a sequenced list. The list can be analyzed after program execution. Souder *et al.* [20] propose to include a filter mechanism. HOTAGENT VISUALIZE offers filters to include or exclude whole components or specific entrances or exits of components. By specifying a filter, the amount of data to visualize can be reduced significantly.

4.4. Visualization

The visualization tool HOTAGENT VISUALIZE presented in this paper is based on the visual component assembly editor HOTAGENT ASSEMBLY. To facilitate the program analysis for the developer, the two-dimensional visualization from the composition process is taken and extended in the third dimension which represents the execution sequence. Components are shown as cuboids with the third dimension representing their lifetime. As the visualization developed is similar to the view presented in the visual component assembly editor, the programmer does not have to rethink the program structure, but can easily recognize the communication (see also Giesl [6]).

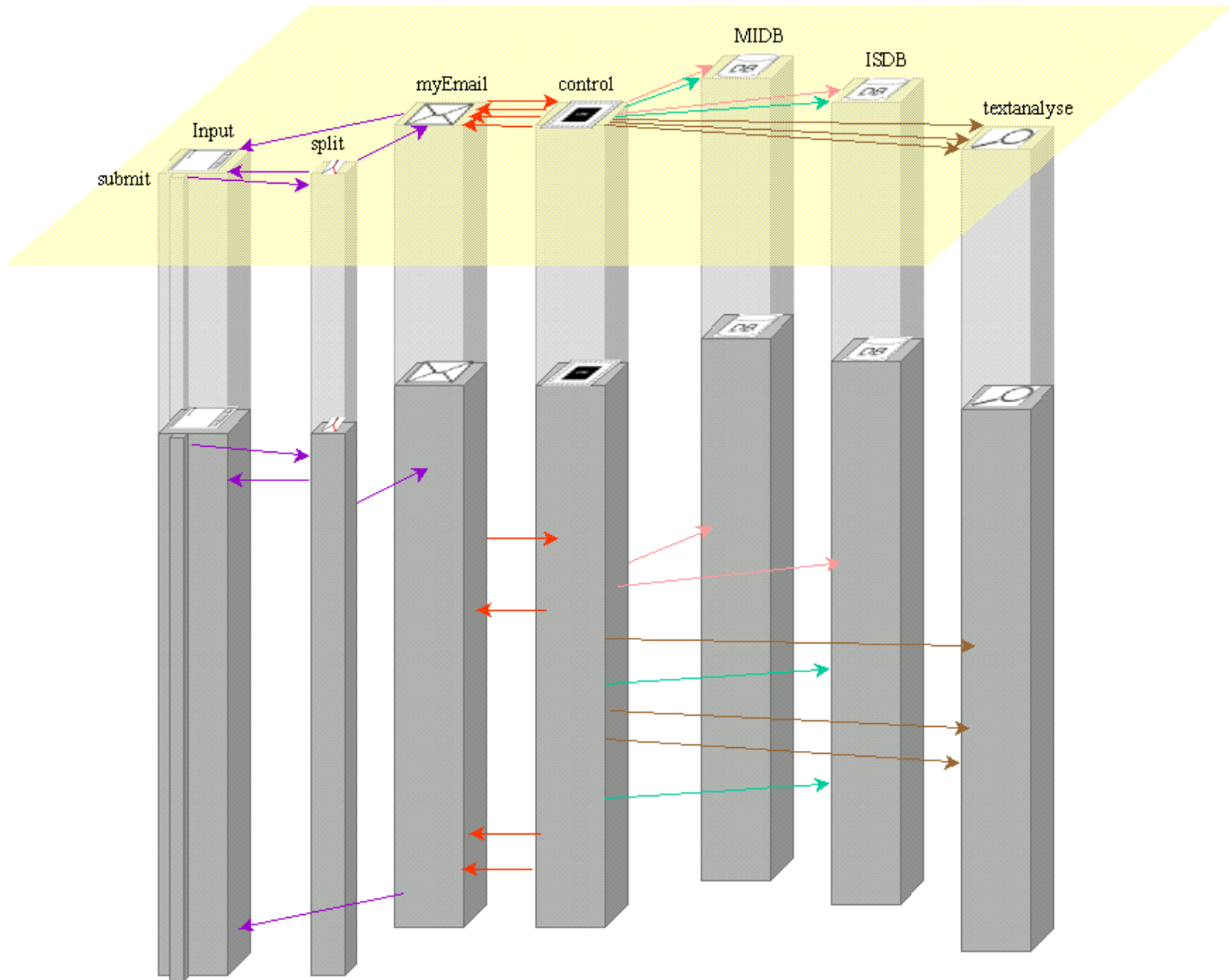


Figure 5. Visualization Derived from HOTAGENT ASSEMBLY editor

The runtime behavior of the program is visualized by placing the descending arrows in the third dimension. These arrows correspond to the measured communication between the components (see Figure 4). They have the same color as the connections specified in the assembling process. In our example, the developer chose purple arrows for the communication between user interface components, red arrows for between email and controller, pink and green arrows between controller and data bases, and brown arrows between controller and text analysis. The position of the arrow on the z-axis, i.e. the height, symbolizes the relative point of time during the measured execution. One arrow appearing under another indicates that the communication occurred later during the execution process, i.e. the topmost communication occurred first.

On top of the cuboid, an icon can be added as shown in

Figure 5. A convenient way for the developer to compare the possible communication channels given by the assembled program with the actually used ones is given by showing the assembled visual program in the same figure, too. In the top level of Figure 5, the visual program itself can be seen with the execution sequence shown below.

Figure 4 and 5 show the same execution sequence of the example. It starts with the topmost arrow between the **submit** and the **split** component which represents the communication happening after the “submit” button in the user interface is pushed. The first arrow between the **controller** and the **text analysis** component corresponds to the start of the execution of the agent’s analyzing work. The message it represents passes the email text to the **text analysis** component. The arrow immediately below it queries the **symptom database (ISDB)** for possible sickness symptoms. These

symptoms are posted to the text analysis component to process the analysis and so on.

In addition to the information on the execution sequence, it is possible to obtain data about the communication itself by selecting an arrow which causes information about the exit name, entrance name, and transported data to appear (see Figure 6). It is intended to provide this information only on demand. Otherwise the visualization would become overloaded and unclear.

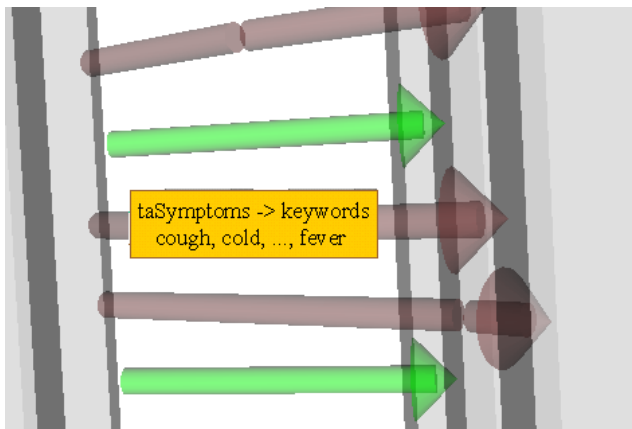


Figure 6. Information About a Communication

Souder *et al.* [20] propose the use of multiple views for a visualization. HOTAGENT VISUALIZE offers the ability to filter the analyzed data, so several different views are possible. It is possible to rotate the three dimensional visualization so that the viewpoint changes. Using this technique, special parts of large programs can be visualized clearly. HOTAGENT VISUALIZE is able to show the complete execution process or zoom into selected parts.

5. Other HotAgent Parts

In the previous section, a tool to analyze and visualize component programs is discussed. HOTAGENT offers some additional tools [13] to work with components.

The simplest way to create new visual components is to use the HOTAGENT PATTERN COMPONENT editor (under development). The first step is to place graphical elements into a workspace. The elements can have different shapes and colors. The second step is to assign behavioral patterns to the elements, e.g., an element can be moved on a line or plane. The editor creates all necessary classes for the new component on its own. Using this editor, new control mechanisms for a user interface can be created.

The HOTAGENT COMPONENT editor is an all-purpose editor. It is good for constructing invisible components. It

supports the programmer by creating new components using other existing components. They can be placed on a workspace and can be connected with each other and new component entrances and exits. Using this editor, a programmer can construct components without knowing very much about the underlying component model.

Using the constructed components, it is possible to build complete applications. This editor has been presented in Section 4.2.

The HOTAGENT TEST [14] tool tests components. The tool supports the component developer and component assembler by testing components as black boxes using defined entrances and exits. HOTAGENT TEST is suitable for component integration testing. The test tool enables the tester to specify a test case in a visual way. In addition, HOTAGENT REGRESSION supports regression tests. The tool allows the execution of several test cases at the same time and inspection of a possibly faulty test case.

6. Related Work

Souder *et al.*[20] present the *Form* framework. It is a tool for dynamic analysis of Java programs. To analyze programs, no modification of the source code is necessary. The execution is recorded by the Java Virtual Machine Profiling Interface. This interface pipes the events through different kinds of filters and through a broadcast architecture. The events carry information, for example, about caller, method, parameter, and time. The events are stored in the XML format. The broadcast architecture can serve one or more independent views that visualize the events stored in the XML file in a meaningful way. The *SD* tool provides such a view. It processes the information and draws an UML sequence diagram, which can lead to the visualization to be complex if there is a large number of events.

JaVis, presented by Mehner [15], is another visualization tool based on UML. It is an environment for visualizing and debugging concurrent Java programs, specially used to find deadlocks. *JaVis* uses the CASE tool Together to generate sequence and collaboration diagrams. The use of UML causes the visualizations to be big and unwieldy. *JaVis* uses the Java Debug Interface, so no changes are needed to trace a program.

Another tool using the Java Debug Interface is *jinsight* [3]. It records the messages sent between objects. Tracing a program, it is possible to define start and end criteria. It uses a UML similar visualization which is more dense.

Bruegge *et al.* [1] describe a *Framework for Dynamic Program Analyzers* called BEE++. Their target applications are monolithic as well as distributed C++ programs whose code is instrumented to generate a stream of events that characterizes the behavior of the program. Since a major drawback of event-based models is that events are in many

ways specific to the system being analyzed, BEE++ provides a framework that can be customized based on inheritance to allow for its use on a variety of problems. The paper shows many details with respect to event generation and communication as well as performance and impact evaluation, but does not discuss how to visualize the events gathered properly.

Systä [22] argues that dynamic analysis of programs gains in importance with the rising need to reverse engineer object-oriented programs. The dynamic nature of object-oriented programs makes it difficult or impossible to understand a program by only looking at its static structure, i.e. its source code. The runtime behavior of the program needs to be considered as well. Systä raises the problem of reconciling the static and dynamic information gathered about a program and proposes to visualize static and dynamic information in separate views while providing a meaningful and consistent connection between the views. In an example analysis, two reverse engineering tools are used to reverse engineer a large Java program. *Rigi* [18, 16] provides views of the static structure of the program, showing program elements such as classes and their members, variables, and function calls. SCED [8] extracts behavioral information by recording events such as method invocations and exceptions while the program runs under the control of an instrumented debugger, and visualizes this information using scenario and state diagrams. To show the connection between static and dynamic views, the static structure as analyzed in the *Rigi* views is used to filter event traces for SCED, and the behavioral information gathered during dynamic analysis is used to annotate and filter a static *Rigi* graph.

SAM (Solid Agents in Motion) [5, 4] is a 3D programming language for parallel systems specification and animation. A SAM program consists of a set of agents that interact by exchanging messages synchronously. The behavior of an agent is specified by production rules with a condition and a sequence of actions that are triggered when the condition is true. The execution of a program is done in cycles with two phases: in the first phase, all agents check the conditions of their rules quasi-parallel and in the second phase all triggered actions are executed. An action can be the sending of a message or the changing of the agent's state. There are two kinds of presentations, an abstract and a concrete one, both of which are three-dimensional. In the abstract presentation, agents are shown as semi-transparent spheres with cones on the surface representing the ports and 3D-objects for the rules inside. Messages are also represented as 3D-objects. In order not to overload the illustration, the communication channels are not visualized. The execution of a program is visualized by a 3D animation process of the agents' communication (by sending messages through the virtual space) and their state changes. As the motions are smooth and the objects are scaled, the observa-

tion is facilitated. In the concrete representation, arbitrary images can be associated with the syntactical objects. In this representation, only the agents and messages are visualized. The execution of a program is shown as a kind of a film of a 3D scenario. As the execution of a SAM program is only visualized by animation, the execution itself is volatile. There is no possibility of looking at the whole program's execution at a glance. This disadvantage is due to the fact that the runtime is presented in the fourth dimension and not in the third, as it is in our visualization.

Lingua Graphica [21] defines a visual 3D syntax for textual programming languages like C++. Elements of Lingua Graphica are solid objects representing functions and relations, loop and branching conditionals, variable types and arrows showing data flow lines and connections. Programming can be done in a virtual environment by building a visual program with the elements. It is also possible to load a textually written program for visualizing it in the 3D language. The sequence of a program is visualized by sequence bars in the third dimension oriented downwards, i.e. in the negative Y direction. Lingua Graphica is a visualization of a textual base language, and thus the elements are low level. Therefore, even short textual programs consume quite a lot of space. The visualization is a static presentation of the program code, but in contrast to our approach, the actual program execution is not visualized.

7. Summary and Future Work

In this paper, the HOTAGENT VISUALIZE tool is presented. It provides a three-dimensional visualization of component programs' runtime behavior based on dynamic analysis.

The communication between the components is dynamically analyzed, and the gathered data can be filtered. The visualization is three dimensional and the focus can be chosen: either the whole program execution can be viewed at the same time, or the developer can zoom in special regions. HOTAGENT VISUALIZE is also useful for testing component programs. Through a tight integration with the rest of the environment, we aim to achieve high acceptance by developers that are already familiar with the HOTAGENT component development environment.

Visual component programs built with HOTAGENT ASSEMBLY are two-dimensional. They consist of components presented by icons and communication channels between them which can be labeled and colored. The 3D-visualization of the runtime behavior of the program is based on this presentation. The extension to the third dimension allows the runtime to be visualized by placing communication arrows below each other. A strength of this approach is its tight derivation from the visually constructed program. The developer can easily compare an actual run

with what he had in mind to happen. The whole run is visualized in one static 3D-figure that can be rotated to provide a good view of all details. This static visualization technique has the advantage that the whole run can be seen at a glance and the sequence does not have to be kept in mind. Of course, a longer run causes a picture that has to be scrolled. More problematic are bigger programs with many components and connections. Their runs are not easy to visualize in a clear way. Although the cuboids are half transparent, the figure becomes more unclear. If the figure has to be rotated too much to see all the arrows the static presentation is too weak to provide a good presentation of the runtime behavior.

A useful extension of HOTAGENT VISUALIZE is an animated presentation of the program execution to provide a dynamic visualization. The tool could present the execution like a movie. The connections are highlighted one after another and, if desired, the corresponding information can be displayed. The developer has the ability to start, stop, or rewind the presentation. The dynamic highlighting can be done within the three-dimensional static presentation of the execution process as described in this paper. This dynamic presentation helps to understand large programs which have more complex visualizations.

References

- [1] B. Bruegge, T. Gottschalk, and B. Luo. A Framework for Dynamic Program Analyzers. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 65 – 82, 1993.
- [2] T. Clausius. Komponenten zur Konstruktion von Agenten. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, 2001.
- [3] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by Analysis of Running Programs. In *Proceedings for Workshop on Software Visualization, International Conference on Software Engineering*, 2001.
- [4] C. Geiger, G. Lehrenfeld, and W. Müller. Authoring Communicating Agents in Visual Environments. In *Proceedings of the Australasian Computer Human Interaction Conference*, 1998.
- [5] C. Geiger, W. Müller, and W. Rosenbach. SAM – An Animated 3D Programming Language. In *IEEE Symposium on Visual Languages*, pages 228 – 235, August 1998.
- [6] A. Giesl. Evaluierung von Komponenten-Entwicklungsumgebungen. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, November 2001.
- [7] R. E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, pages 39 – 42, October 1997.
- [8] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. Automated support for modeling oo software. *IEEE Software*, 15(1):87 – 94, January/February 1998.
- [9] C. Lüer and D. S. Rosenblum. Wren – An Environment for Component-Based Development. Technical report, Department of Information and Computer Science, University of California, Irvine, September 2000.
- [10] C. Lüer and D. S. Rosenblum. Wren – An Environment for Component-Based Development. In *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 207 – 217, September 2001.
- [11] L. Martin. *HotAgent homepage*. <<http://www.gkec.informatik.tu-darmstadt.de/HotAgent/>> (July 2002).
- [12] L. Martin. HotAgent Component Assembly Editor. In J.-G. Schneider and M. Lumpe, editors, *Proceedings Workshop on Component Composition Languages*, pages 25 – 32, September 2001.
- [13] L. Martin. Visual Development Environment Based on Component Technique. In *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 346 – 347, September 2001.
- [14] L. Martin. Visual Component Integration and Regression Test. In *ICSR7 2002 Workshop on Component-based Software Development Processes*, April 2002.
- [15] K. Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. *Software Visualization International Seminar*, pages 163 – 175, May 2001.
- [16] H. A. Müller and K. Klashinsky. Rigi - A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.
- [17] B. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97, 12 1990.
- [18] U. of Victoria. Rigi Web Server. <http://www.rigi.csc.uvic.ca> (June 2001).
- [19] S. P. Reiss. Cacti: A Front End for Program Visualization. In *Proceedings of the 1997 IEEE Symposium on Information Visualization*, pages 46 – 49, 1997.
- [20] T. Souder, S. Mancoridis, and M. Salah. Form: A Framework for Creating Views of Programm Executions. In *Proceedings IEEE International Conference on Software Maintenance*, pages 612 – 620, November 2001.
- [21] R. Stiles and M. Pontecorvo. Lingua graphica: A visual language for visual environments. In *IEEE Symposium on Visual Languages*, pages 225 – 227, 1992.
- [22] T. Systä. On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. In *Proceedings 6th Working Conference on Reverse Engineering (WCRE99)*, pages 304 – 313, 1999.
- [23] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [24] M. Wiggins. An Overview of Program Visualization Tools and Systems. In *Proceedings of the 36th annual conference on Southeast regional conference*, pages 194 – 200, 1998.