Understanding the behaviour of distributed applications through reverse engineering

To cite this article: T H Kunz and J P Black 1994 Distrib. Syst. Engng. 1 345

View the article online for updates and enhancements.

Related content

- Configuring scientific applications in a heterogeneous distributed system
  Patrick T Homer and Richard D Schlichting

- Reusing sequential software in a distributed environment
  A Bartoli, G Dini, M Luise et al.

- A software architectural design method for large-scale distributed information systems
  Hassan Gomaa, Daniel Menascé and Larry Kerschberg
Understanding the behaviour of distributed applications through reverse engineering

Thomas H Kunz and James P Black

Department of Computer Science, University of Waterloo, Ontario, Canada
N2L 3G1

Received 11 May 1994

Abstract. Understanding the behaviour of distributed applications is a very challenging task, due to their complexity. The top-down use of suitable abstraction hierarchies is frequently proposed to manage this complexity. Given the size of distributed applications, manually deriving such abstraction hierarchies seems unrealistic. This paper discusses tools that automate the derivation of suitable abstraction hierarchies and reports on initial experience with these tools. These abstraction hierarchies enable a top-down approach to the application behaviour understanding task, keeping the overall amount of information manageable. We have modified an existing prototype visualization tool to provide abstract visualizations of an execution. A user can navigate through the abstraction hierarchies derived with our tools, displaying an execution at various levels of abstraction. Examples of such abstract visualizations for the execution of one specific distributed application are given. In general, the abstractions derived represent meaningful parts of the application: they can be interpreted in terms of the application domain.

1. Introduction

Empirical evidence shows that programmers spend the majority of their time on program maintenance tasks [Cor89]. A common first (and difficult) step in all maintenance tasks is to gain an understanding of the application at hand. In particular, highly complex applications are expensive to maintain [BDKZ93]. The associated costs necessitate appropriate tools to assist in reconstructing and analysing information in existing programs, to aid programmers in debugging, enhancing, modifying, and/or rewriting them. This paper focuses on one aspect of the program understanding task, namely understanding the behaviour of an application, and presents tools to facilitate application behaviour understanding for one specific class of applications, distributed applications.

It is frequently claimed that the 1990s will be the decade of distributed systems [Mull89, Ray89]. However, an exact definition of what constitutes a distributed system is hard to give. Typically, a list of symptoms characterizing such a system is provided, such as: multiple processing elements, interconnection hardware, independent failure of processing elements, and shared state among the processing elements.

Distributed applications are the programs executing on a distributed system. They have characteristics matching those of distributed systems. Schill [SHM89], for example, gives a list with the following characteristics: cooperation of loosely-coupled, distributed autonomous entities, complex structure, topological irregularity, intensive remote communication, dynamically changing communication patterns, separation of address spaces, and dynamic configuration changes.

It follows from the characteristics of distributed applications that they are difficult to understand and to maintain [OMG84, JLSU87]. Some of the more frequently cited reasons are parallelism, non-deterministic execution, lack of a global component (clock, memory), significant and variable communication delays, and a tendency towards large size. These characteristics make distributed applications simultaneously highly complex and hard to understand.

To reduce this complexity, a number of approaches are proposed in the literature to enable a programmer to work at abstract levels, ignoring details of the implementation. Some of these proposals are the use of high-level distributed programming languages, parallelizing compilers, or special middleware layers of software like OSF’s Distributed Computing Environment (DCE). None of these approaches provides (as yet) a satisfying solution to the complexity problem. Therefore, our work is based on the following assumptions. First, humans still have to cope with the complexity of distributed applications explicitly. Second, the only reliable source of information about an application is the source code itself [Byr91, Rab91]. Third, a top-down approach towards application behaviour understanding should be supported. A top-down approach
starts with the component at the top of a hierarchy and successively moves to the next lower level.

In the context of application behaviour understanding, a hierarchy of abstract behaviour visualizations can be used to minimize the complexity involved. Typically, a programmer starts the understanding task by trying to gain a global overview [KCC89]. This can be achieved by viewing an application execution at a very high level of abstraction. In the course of the understanding process, more difficult or interesting parts of the application are isolated and examined at lower levels of abstraction. In this way, more and more detailed information is collected for smaller and smaller parts of the application, keeping the overall amount of information collected manageable. The process continues until all aspects of interest are sufficiently understood. Ideally, more than one abstraction hierarchy should be employed, each hierarchy emphasizing different aspects of the program behaviour. A programmer or maintainer could then switch between these hierarchies, depending on his or her current focus of attention.

Good abstraction hierarchies, however, are difficult to construct. Programmers typically approach the understanding task by mentally applying abstraction to the application at hand, using information from the software source, related documents, and previous experience. This abstraction process is tedious and error-prone for complex applications. Our work centres around the development of tools to derive suitable abstraction hierarchies automatically. This paper briefly describes two abstraction tools and reports on our initial experience with them.

Our tools are examples of reverse engineering tools for distributed applications. Reverse engineering aims to reconstruct a possible design of an application by using the software system as the only information source [Byr91]. To achieve this goal, reverse engineering tools usually extract semantic clues from the source code by a static analysis [CdC91]. The results of this analysis are frequently either transformed into dataflow diagrams [BCC89] or stored in a relational database [CNR90] for further processing. In addition, to analyse the behaviour of applications, dynamic information is collected [Cor91, RaB91]. Existing reverse engineering tools are not only relatively primitive, but also very difficult to adapt to a parallel and distributed environment. This paper presents new reverse engineering tools for a distributed environment.

The distributed programming language used in this paper is Hermes, a high-level, process-oriented language for distributed computing [SBG+91]. We have chosen Hermes from the large number of distributed programming languages for a number of reasons. First, following [GC92], a complete programming model for distributed applications consists of two separate pieces: the computation model, which allows programmers to build a single computational activity, and the coordination model, which binds separate activities into an ensemble. In Hermes, this coordination model is its process model. Hermes processes are created and terminated dynamically and communicate and synchronize by synchronous and asynchronous message passing. Not only has this model been incorporated into a number of other computational languages, but the message-passing paradigm it embodies is a very common one in distributed computing. This facilitates porting our results to environments with a similar coordination model. Second, processes in Hermes are both the unit of parallelism and the unit of modularization. Even a moderate Hermes application consists of a large number of processes. Hermes therefore is a convenient vehicle for testing our ideas in a complex environment. A last, more pragmatic reason is that we had access to the source of an existing visualization tool for Hermes applications [Tay92].

Hermes is a declarative and process-oriented language for distributed applications, using message passing for process communication and synchronization. We are currently exploring the generalization of our approaches to distributed environments with similar characteristics, including distributed object-oriented programming languages. Extending our work to environments based on the shared-memory programming paradigm, however, requires additional research to develop a suitable model of the application execution, similar to the one mentioned below for the message-passing programming paradigm.

This paper is organized as follows. After this introduction, a visualization frequently used to depict distributed executions is shown. Next, two tools to derive suitable abstractions automatically are presented. Using a sample distributed application, different abstract visualizations of one execution are given. The paper concludes with a summary of our work and reports on future work.

2. Visualizing distributed executions

When reasoning about distributed executions, an event-based approach is typically employed. Following [Lam78], each process or autonomous entity is viewed as a totally ordered sequence of atomic events. An event represents some activity performed by some process and is considered to take place at an instant in time. The precedence relation defined by Lamport [Lam78] imposes a partial order on the set of events and captures the notion of potential causality: only events that precede an event \(a\) can have influenced this event. The precedence relation is therefore an important relation when reasoning about distributed execution.

Events constitute the lowest level of observable behaviour. They could be defined according to the particular aspect that interests a specific observer. An observer interested in the file system might define events related to the completion of file open, read, write, and close actions. Similarly, an observer interested in security issues could define events related to encryption and decryption computations. However, a more general and fixed set of events is most commonly used. The lowest level of observable behaviour in a distributed system, the primitive events, are events that relate processes to each other, such as sending and receiving messages or process creation and termination. It is assumed that the local (sequential) computation for each process can be dealt with using traditional approaches.
As discussed before, a distributed application consists of a number of autonomous and sequential processes, cooperating to achieve a common goal. Cooperation includes both communication and synchronization and is achieved by the exchange of messages. Both synchronous and asynchronous message passing is allowed. The communication channels may or may not have FIFO property. Processes can be created and terminated dynamically. The distributed application behaviour is frequently depicted using process-time diagrams [Fid91, SM94].

Figure 1 shows the visualization provided by the tool described in [Tay92, Tay93]. This tool draws a set of horizontal lines, one for each process, placing a symbol on the appropriate line for each event. Time flows from left to right, and a scrollbar allows for scrolling in the vertical (process) dimension. (Scrolling in the time dimension is more complex since it depends on the actual partially-ordered execution; it has also been implemented by the tool.) The lines are preceded by the process name. Events that represent the two endpoints of a communication activity are connected by an arrow, using a vertical arrow for synchronous communication and a sloping arrow for asynchronous communication. The arrow points from the sending to the receiving event. Different symbols are used to distinguish different types of events. Filled squares indicate process creation events, open squares depict process termination events. Filled circles represent (synchronous or asynchronous) message sends and receives, while open circles are used for the return and receive of a return for synchronous messages. The process lines are drawn in three different states, approximating the process state. Before a process starts (indicated by a filled square) and after; its termination (the open square), the line is invisible. After sending a synchronous message, a process is blocked until it receives the return message. This is represented by a dashed line. In all other cases, a process is assumed to be active, displayed by a solid line.

Process-time diagrams contain two different entities: processes and primitive events. The two abstraction tools implemented derive abstractions based on these two entities. They recursively group primitive entities into more abstract entities. The sample abstract visualizations discussed later will demonstrate that the abstractions derived by our tools are not just arbitrary groups of more primitive entities, but do in fact represent meaningful entities of the execution.

3. Automatic process clustering

Process abstraction allows groups of processes to be combined into clusters, eliding the interaction among processes in the cluster. Combining clusters into higher-level clusters leads to an abstraction hierarchy. Our tool identifies the appropriate processes and subclusters to be combined. It clusters processes using semantic information that characterizes application processes and information about the actual interprocess communication at runtime. Information about the actual IPC at runtime is provided by a modified Hermes interpreter, and the semantic characteristics of each process are determined by a combination of static analysis and interpretation of the IPC information.

A first characteristic collected for each application process is its type. Several process taxonomies based on interprocess communication have been proposed [BEWSS, Lai91]. These taxonomies vary in the number of categories they include, but they all divide processes into at least two broad categories: workers and servers. Servers offer one or more services in an endless loop. All other processes are called workers, advancing the state of the computation. We deliberately do not call them clients because the term client usually implies that the process calls a server. This, however, is not necessarily true for worker processes. The process type is determined by a static analysis of the process source.

A second characteristic is the process complexity. A process is defined to be complex if it calls other application processes; otherwise, it is simple. This characteristic cannot be determined by a static source analysis but depends on the actual IPC at runtime. It is determined by analysing the interprocess communication that occurs during the traced execution.

These two characteristics are used by the clustering rules. These rules describe patterns of processes that can be clustered if they are of certain types and have a certain interprocess communication structure. Furthermore, these characteristics are determined not only for application processes, but are also assigned to the process clusters derived by the clustering rules. By using the same semantic description for both processes and process clusters, the clustering rules can be applied recursively, creating a multi-level hierarchy of process clusters.

The derivation of appropriate clustering rules is based on the following idea. Programming paradigms are collections of conceptual patterns that influence the design process and ultimately determine the structure of a program [ABZ92]. By analysing programming paradigms used for distributed programming, potential program structures can be identified. Using rules that describe these structures, our tool analyses the runtime behaviour of a distributed
application and automatically generates a hierarchy of process clusters. The specific process clustering rules are based upon design principles for distributed applications advocated in the literature [CG89, Fin87, NS87, Sha89]. A total of ten different clustering rules have been identified and implemented. Figure 2 summarizes them; for a more detailed description of the rules and the complete clustering tool see [Kun93b, Kun94a]. The figure gives a typical example for the application of each rule. Processes are represented by single characters, indicating the process type: W for a process of type worker, S for a process of type server and * for a process of arbitrary type. IPC connections that have to exist for a specific rule to be applicable are drawn as solid arrows. Where other connections might exist, this is indicated by a dashed line with double arrows. All processes that are clustered by the corresponding rule are surrounded by a dashed box. The resulting cluster semantics are omitted from this picture.

The layered system rule, based upon the layered system paradigm, analyses and clusters all application processes at the same time, creating a number of new higher-level process clusters. It assigns processes into clusters such that all processes in cluster \( i \) only communicate with processes in the same cluster or cluster \( i + 1 \).

All other rules describe substructures within a distributed application. Multiple instances of the structure described by the rule might exist in parallel and will be discovered when applying the rule. They can be further divided into three subcategories: linear substructures, biconnected substructures, and others. One example for a rule describing a linear substructure is the master-slave rule, derived from the master-slave paradigm. It scans all processes for simple workers that are called from exactly one other process and clusters these processes with the calling process. The underlying notion is that the calling process passes on some of its work to the worker process being called. The calling process can be of type server or worker. The resulting cluster is assigned the name and type of the calling process. Other rules in this category are the client-server rule, the complex server rule, the administrator concept rule, the pipeline & filter rule, and the compute-aggregate-broadcast rule.

Biconnected substructures describe sets of processes that communicate in such a way that the removal of a single communication link still leaves the process set at least weakly connected. Two rules describing such structures have been derived: the peer groups rule and the iterative relaxation rule. The peer groups rule is derived from a generalization of the result parallelism paradigm discussed in [CG89]. This paradigm proposes a structure for distributed applications where a series of values is produced with predictable organization and interdependencies of the application processes. One likely IPC structure under this model is a group of interconnected worker processes. The second rule, iterative relaxation, describes a special case of the peer groups rule. The iterative relaxation paradigm postulates a design in which adjacent regions are assigned to different processes. Each process carries out activities local to its region, communicating with neighbours when necessary. This communication with the neighbours is, in general, symmetric. Therefore, we impose the additional requirement that in a set of worker processes that forms a peer group, the communication between any pair of communicating processes is bidirectional.

The only clustering rules not belonging to any of the previous categories is the divide & conquer rule. The divide & conquer paradigm postulates a problem-solving approach that splits the original problem into identical smaller problems which are then solved in parallel. Given that the subproblems are identical, the application should contain numerous instantiations of the same process set (necessary to allow the processing of the subproblems in parallel). The processes within each set communicate with each other following the same pattern. Furthermore, they only communicate with the same external processes.

Conflicting cluster alternatives exist whenever one process might be clustered by different rules with a different set of processes. Since we are interested in the derivation of a tree-structured cluster hierarchy, these conflicts must be resolved. Rather than determining a priori a fixed order of rule application (which would be hard to justify), we added a quantitative cluster evaluation measure [Kun94a] to the tool. This measure is used to resolve clustering conflicts as follows. In a first round, all clustering rules are applied in parallel, but no clusters are formed yet. If a process could belong to two possible clusters, only the cluster with the best quantitative evaluation is built in the second round.

The evaluation measure is based on a similarity measure defined in [PCB92]. This measure uses a characteristic vector for each entity (process, software module, etc) that counts the references to particular data types within the entity. Vectors counting references to all such data types are determined by a static analysis of the process sources. The pairwise similarity between any two processes is calculated as:

\[ \text{Sim}(X, Y) = \frac{X^T Y}{||X|| ||Y||} \]
where $X^T Y$ is the inner product of the characteristic vectors $X$ and $Y$ and $||X||$ and $||Y||$ are their Euclidean norms.

The characteristic vectors used to calculate the pairwise similarity are determined by a static source analysis. Consequently, multiple instantiations of the same process source are indistinguishable. However, we are applying the measure to all processes created during the runtime of a distributed application. Here, multiple instantiations of the same source frequently exist. While these processes will be assigned identical characteristic vectors, they will, in general, differ in their communication behaviour. To be able to differentiate between multiple instantiations of the same process source, pairwise similarities are filtered. The pairwise similarity of two processes is reduced to zero (filtered) if at least one of them has sibling instantiations, they are instantiations of different source modules, and they do not communicate with each other. For a more detailed discussion of this filtering operation see [Kun93a].

Using the filtered measure $\text{Sim}_f(X, Y)$, coupling and cohesion for a process cluster $P$ are defined as follows. The cluster cohesion is the average pairwise similarity of processes within the cluster, counting each process pair only once:

$$\text{Cohesion}(P) = \frac{\sum_{i \neq j, 1 \leq i, j \leq m} \text{Sim}_f(p_i, p_j)}{\sum_{i=1}^{m-1} i}$$

where $P$ is a set of processes $\{p_1, \ldots, p_m\}$. Similarly, the coupling of a process cluster with its environment is calculated as:

$$\text{Coupling}(P) = \frac{\sum_{i=1}^{m} \sum_{j=m+1}^{n} \text{Sim}_f(p_i, q_j)}{m \times n}$$

where $P$ is a set of processes $\{p_1, \ldots, p_m\}$ and $\{q_1, \ldots, q_n\}$ is the set of application processes not in $P$.

4. Automatic event abstraction

Our second tool groups events into abstract events. The theoretical foundation is the chunking theory of program understanding [PPBH91]. According to this theory, a programmer and/or maintainer recognizes functions of groups of statements which he/she chunks together. Lower-level chunks are then grouped into higher-level chunks. The event abstraction tool mirrors this bottom-up derivation of abstractions. It combines information derived by a static source analysis with information about the execution to derive abstractions. The tool projects the static structure of the source on the events in the execution trace. It uses the fact that most primitive events are created when specific statements in the application source are executed. For the above-mentioned fixed set of primitive events, the relevant statements are process creation, process termination, and IPC statements. For example, a call event is created whenever a call statement in the source is executed. It follows that statements in the source can be matched with one or more primitive events in the event trace.

Figures 3 and 4 provide a high-level description of how the abstraction tool works. For a more detailed description see [Kun94b]. Before the abstraction process starts, control-flow graphs for each application process are derived by a static analysis. Statements in the same

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{First abstraction step.}
\end{figure}
abstract events in the two event sets \( a \) and \( b \) and forms the top-level abstraction for this example. The tool stops because the abstraction focus reaches the top-level abstract statement for each application process.

### 5. Results

We tested our tools on a number of Hermes applications. These applications consist of a few hundred to a few thousand lines of source, implemented by different programmers. The trace files, recording all IPC and process creation/termination events, contain from 2500 to 3600 events. This section demonstrates our results by discussing various resulting abstract visualizations of the execution of one of these applications, `makehermes`. The following few paragraphs also show that the abstractions derived represent meaningful entities.

`Makehermes` is the Hermes version of the Unix `make` tool. A Hermes application consists of separately compiled process and definition modules which are imported by `linking` and `usage` lists respectively. Parsing these lists in the source reveals all dependencies, and so a separate `makefile` is not necessary. `Makehermes` builds a graph structure representing the dependencies, and checks, starting from the leaves, whether a source module must be recompiled. To limit the extent of this recursive dependency check, an `environment` variable restricts its scope as follows: only sources in the current directory and in directories specified by the environment variable are scanned for further dependencies. In the traced execution, the dependencies for a single definition module are checked. The source modules included by this definition module must be recompiled. Therefore, no compiler is invoked. This execution creates a total of 175 processes, and the event trace contains 2534 primitive events.

Figure 1 depicts a segment of the execution history at the lowest abstraction level. The execution segment occurs at the end of the `makehermes` execution. For this window size, not all relevant processes fit on the screen, as indicated by the arrows leaving the window at the top and bottom boundaries. A higher-level visualization of this segment is shown in figure 5.

In figure 5, lines preceded by a name in capital letters indicate process clusters. On a colour display, clusters are highlighted compared to processes. Only events with a partner outside the cluster are displayed, while events purely internal to a cluster are ignored. Process clusters therefore not only reduce the display space in the process dimension, but also in the time dimension. The cluster is always represented by one or more solid lines: the display details are discussed in [Tay93]. In this figure, all processes are shown, either by displaying them individually or as part of a process cluster. Using such a high-level visualization, it becomes far easier to analyse and reason about the execution.

Figure 5 contains four large clusters, derived automatically by the process clustering tool described above. `System` clusters all processes that form the Hermes runtime system and is a standard cluster derived by the tool. The other three clusters group application processes. These clusters were derived by applying all clustering rules and choosing the ones with the highest cluster evaluation measure. This step is completely automated, no user intervention is required. Figure 5 displays the internal structure of these clusters, representing subclusters with boxes.

All three clusters encapsulate processes that together offer a single service. The cluster `STD10.PROGLIT`, for example, groups all 19 processes involved in file I/O activities during the scan of a source module. The common interface to all processes is `stdio.proglit`, which forwards file I/O requests to the appropriate process in the subcluster `LAYER.5`. `GETUSES` determines and returns
Understanding distributed applications

Figure 6. Three process clusters of the makehermes execution.

Figure 7. Source scan in the makehermes execution.

all dependencies for a single source file (the contents of the 'linking' and 'usage' lists). The third cluster, GETDEP, obtains the timestamp for a specific source file and invokes GETUSES to scan it for additional dependencies if the source file is in the current scope. The list of dependencies is returned to the make process, which builds and maintains the global dependency graph.

Figure 5 shows the following sequence of actions. After scanning the source of the definition module, process make knows all sources this module imports. It repeatedly invokes GETDEP to detect the dependencies of these modules. Cluster GETDEP uses the process File Modification Time and processes within the standard Hermes runtime system to locate these sources. The sources are not within the current scope, so GETUSES is not invoked to scan a source file. After checking all dependencies, make determines that the existing object file is more recent than any of the source files on which it depends and returns to makehermes without invoking the Hermes definition module compiler (via process dcomshell).

Next, we will give two examples of abstract visualizations containing abstract events. To reduce the amount of display space necessary, these visualizations will show the same process clusters as figure 5. The examples show different parts of the execution at intermediate event abstraction levels.

Figure 7 shows three abstract events, involving processes from STDIO.ProGLIT and GETUSES as well as

events in tokenize. Abstract events generally involve primitive events from a number of different processes. Therefore, they are displayed as vertical rectangles, stretching over the range of all processes involved. The intersection of this open rectangle with a process is filled if events from this process are part of the abstract event. Otherwise, the intersection is left open, as in the case of process tokenize and the first abstract event in figure 7. The displayed segment of the execution corresponds to initialization and source scan of the target definition module. At its end, GETUSES returns the list of dependencies to GETDEP.

Figure 8 displays the end of the execution, similar to figure 5. Due to the display space saved by event abstraction, a larger piece of the execution fits on the screen. The same execution history segment without event abstraction occupies approximately two and a half windows in the horizontal (time) dimension. Figure 8 shows that make invokes GETDEP multiple times to determine dependencies for a specific source file. In each case, GETDEP checks whether the source file is in the current directory (the first call and associated reply to File Modification Time). Since this is not the case, GETDEP then proceeds to find the source file, obtains its timestamp, and checks whether it is in the current scope. The three abstract events in figure 8 group all primitive events involved in this activity. Because none of the sources imported by the target module are in the current scope, GETUSES is never invoked. Figure 8 displays three such iterations before make has exhausted all dependencies derived from the source scan shown in figure 7. It then determines that the existing object file is up-to-date and returns to makehermes without invoking a compiler.

To indicate the reduction in the display space needed, a few statistics were collected for this execution. Using the same window size as in figure 1, the lowest-level visualization of the complete execution application occupies 65 windows in the time (horizontal) dimension. In the process dimension, seven windows are necessary. Using process clustering, all process and cluster traces fit into one window, and the execution spans 15 windows in the time
dimension. Using a combination of process clustering and event abstraction, the complete execution history fits into twelve windows in the time dimension.

Of course, reductions in these numbers could also be achieved by arbitrarily grouping processes into clusters and primitive events into abstract events. However, the discussion above shows that the abstractions derived not only result in a reduction of the display space needed, but also correspond to meaningful entities in terms of the application domain. The resulting abstract visualizations are therefore both useful and meaningful. As an aside, the visualization tool includes facilities to reorder processes and clusters in the window, which can also improve the quality of the visualization dramatically.

6. Conclusions and future work

Abstract visualizations of a distributed execution are based on the notions of process clusters and abstract events. Ideally, these entities should have the same properties as their primitive counterparts, processes and primitive events. This, however, is not the case. The individual processes are always sequential, but process clusters might exhibit concurrency at their interface. Understanding parallel entities, however, is more difficult than understanding a sequential entity. Similarly, abstract events do not occur instantaneously, but extend in time. Therefore, the precedence relation on abstract events is not necessarily transitive, which might make reasoning about the causality harder.

One possible remedy for this additional complexity of the abstract visualizations is to restrict the abstraction operation. Restricting the abstraction operator could ensure that only a set of processes with a sequential interface is clustered. Similarly, such restrictions could limit abstract events to structures that guarantee transitivity of the precedence relation. This approach, however, severely limits the expressiveness of our abstraction tools. To retain flexibility in the derivation of abstractions, we deliberately accept the potentially higher complexity of the resulting abstract visualizations. The visualization tool provides graphical representations designed to minimize this additional complexity. Visualizing process clusters is done with the minimal number of (sequential) lines. Frequently, process clusters can be visualized with a single line, indicating that the cluster has a sequential interface. Judging from our experience to date, it appears that good process clusters have only a limited degree of concurrency at their interface. Similarly, the set of predecessors and successors for each (primitive or abstract) event can be highlighted on the display, facilitating reasoning about causality. Our initial experience confirms that the abstract visualizations indeed simplify the program understanding task. However, more experience with our tool is needed to evaluate the relative merits of the visualizations provided.

Design recovery is a complex and abstract activity. In view of this, it seems unlikely that completely automatic abstraction tools will ever provide a definitive abstract view of the execution of an application. However, the tools and heuristics suggested here can provide valuable assistance in the semi-automatic construction of good initial abstraction hierarchies. One area of future work is to examine whether additional information sources will increase the quality of the abstraction hierarchies derived. A variety of additional information sources can be imagined: predefined fixed abstractions, information about standard abstractions in the runtime system, expected abstractions based on previous runs, utilizing information obtained during the development process, or a more detailed semantic analysis of the application processes.

Our abstraction tools have been used in conjunction with the process and event visualization component of a distributed debugger. The debugger allows the user to create and save abstraction hierarchy description files through an X-window interface, and to use such files to reduce the apparent complexity of the behaviour as presented. Our automatic tools also produce description files which can be read and modified as the user refines his/her understanding of the design. As design recovery evolves over a number of executions, the manual and machine resources invested to construct appropriate abstraction hierarchy descriptions are repaid in improved understanding and user productivity.

Currently, the debugger itself has been used to present behaviour in five different programming language environments, of which Hermes is only one. While the abstraction tools have currently been implemented only for Hermes, we see no conceptual difficulties porting them to other environments that are based on the message-passing paradigm. Hermes has served as a vehicle for our experiments, but our debugger is already being used in much more practical multi-threaded and distributed programming environments where the complexity of the distributed applications imposes a need for suitable abstraction.

References


[CdC91] Cimitile A and De Carliini U 1991 Reverse engineering: algorithms for program graph reduction Software—Practice and Experience 21 519-37


[Cor89] Corbi T A 1989 Program understanding: challenge for the 1990s IBM Syst. J. 28 294-306
Understanding distributed applications


[Lai91] Lai R 1991 Ada task taxonomy support for concurrent programming ACM SIGSOFT Software Engng Notes 16 (1) 73–91

[Lam78] Lamport L 1978 Time, clocks, and the ordering of events in a distributed system Commun. ACM 21 (7) 558–65


[Ray89] Raynal M 1989 Distributed algorithms: their nature and the problems encountered Parallel and Distributed Algorithms ed M Cosnard, Y Robert, P Quinton and M Raynal (Amsterdam: Elsevier) pp 179–185


[Sha89] Shaw M 1989 Larger scale systems require higher-level abstractions 5th Int. Workshop on Software Specification and Design; ACM SIGSOFT Software Engng Notes 14 (2) 143–6

[Sch89] Schill A, Heuser L and Muhlhaeuser M 1989 Using the object paradigm for distributed application development Kommunikation in vernetzten Systemen (Berlin: Springer)

