DeWiz – A Modular Tool Architecture for Parallel Program Analysis

Dieter Kranzlmüller, Michael Scarpa, and Jens Volkert

GUP, Joh. Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria/Europe
kranzlmueller@gup.jku.at
http://www.gup.uni-linz.ac.at/~dk

Abstract. Tool support is an important factor for efficient development of parallel programs. Due to different goals, target systems, and levels of abstraction, many specialized tools and environments have been developed. A contrary approach in the area of parallel program analysis is offered by DeWiz, which focuses on unified analysis functionality based on the event graph model. The desired analysis tasks are formulated as a set of graph filtering and transformation operations, which are implemented as independent analysis modules. The concrete analysis strategy is composed by placing these modules on arbitrary networked machines, arranging and interconnecting them. The resulting DeWiz system processes the data stream to extract useful information for program analysis.

1 Introduction

Program analysis includes all activities related to observing a program's behavior during execution, and extracting information and insight for corrective actions. Corresponding functionality is offered by dedicated software tools, which support the user during the diverse activities of program analysis. The complexity of these tasks is affected by a series of factors, which often represent substantial problems for software tool developers.

A serious challenge is introduced by the characteristics of parallel and distributed programs, where the program's behavior is constituted by multiple concurrently executing and communicating processes. Different means of process interaction, e.g. shared memory or message passing, and substantially distinct hardware architectures call for dedicated solutions. Different goals regarding to the intended improvements, e.g. performance tuning or correctness debugging, and different levels of abstraction, from source code to machine level, require dedicated functionality. Additional difficulties are given by the typical scale of such programs, e.g. execution times of days, months, or even longer, and the possibly large number of participating processes.

In contrast to many existing and specialized tools, e.g. AIMS [16], Paje [3], Paradyn [11], Paragraph [4], PROVE [5], and VAMPIR [2], the approach of DeWiz tries to offer a unified solution for parallel program analysis. The idea of DeWiz stems from the fact that, despite the many differences, most tools rely of graph-based analysis methods or utilize space-time diagrams for visualization of program behavior. Consequently, a unified directed graph may be used to capture different program properties and to represent the basis for subsequent analysis activities.

This paper describes the software architecture of DeWiz and discusses its benefits for parallel program analysis. Section 2 provides an overview of related work and the basics of the universal event graph model. Section 3 presents the tool architecture of DeWiz and the concrete activities of constructing a DeWiz system for a particular analysis task. Some examples of DeWiz are presented in Section 4, before conclusions and an outlook on future work summarize the paper.

2 The Universal Event Graph Model

The approach of DeWiz stems from our work on the Monitoring And Debugging environment MAD [6]. MAD is a collection of software tools for debugging message-passing programs based on the MPI standard [13]. At the core of MAD are the monitoring tool NOPE and the visualization tool ATEMPT [7].

Similar to MAD, the basic representation utilized during all analysis activities of DeWiz is the abstract event graph, which can be defined as follows [7]:

Definition 1: Event graph [7]
An event graph is a directed graph $G = (E, \rightarrow)$, where E is the non-empty set of events e of G, while $\rightarrow$ is a relation connecting events, such that $x \rightarrow y$ means that there is an edge from event x to event y in G with the "tail" at event x and the "head" at event y.

The events $e \in E$ of an event graph are the events e_p^i observed during program execution, with i denoting the sequential order of the events relative to object p that is responsible for the event. The vertices of such a graph represent a subset of the state data at a concrete point of time during the program's execution, while the edges represent the transition from one state to another.

Table 1. Event graph data corresponding to target system and event type

Target system	**Event**	**Event Data**
parallel/distributed message-passing program	send/receive	source, destination, message type, parameters, ...
multi-threaded shared memory	read/write memory	memory address, size, contents, ...
database/transaction	db_sum	table, location, access time, ...
file I/O, disk access	write	filename, device, buffer size, ...
web server	get	client-IP, URL, ...

Using the event graph for program analysis activities requires to map events onto actual operations of the target program. The universal properties of the event graph allow its application for different kinds of programs and different levels of granularity and abstraction. Some examples are given in Table 1 (compare with [9]).

The relation connecting the events of an event graph is the happened-before relation, which is the transitive, irreflexive closure of the union of the relations $\xrightarrow{S}$ and $\xrightarrow{C}$ as follows:

Definition 2: Happened-before relation [10]

The happened-before relation $\rightarrow$ is defined as $\rightarrow = (\xrightarrow{S} \cup \xrightarrow{C})^+$, where $\xrightarrow{S}$ is the sequential order of events relative to a particular responsible object, while $\xrightarrow{C}$ is the concurrent order relation connecting events on arbitrary responsible objects.

The sequential order $e_p^i \xrightarrow{S} e_p^{i+1}$ defines, that the i^{th} event e_p^i on any (sequential) object p occurred before the $i+1^{th}$ event e_p^{i+1} on the same object. The concurrent order $e_p^i \xrightarrow{C} e_q^j$ defines, that the i^{th} event e_p^i on any object p occurred directly before the j^{th} event e_q^j on any object q, if e_p^i and e_q^j are somehow connected by their operation.

With the event graph, $e_p^i \rightarrow e_q^j$ means, that e_p^i preceded e_q^j (and e_q^j occurred after e_p^i). For program analysis, a very important view is that $e_p^i \rightarrow e_q^j$ describes the possibility for event e_p^i to causally affect event e_q^j, or in other words, event e_q^j may be causally affected by event e_p^i, if the state of object q at event e_q^j depends on the operation carried out at event e_p^i. Consequently, a program's behavior can be described by its set of program state transitions in the event graph, and this representation can be used for program analysis activities.

3 DeWiz – Tool Architecture

The event graph model as defined above is the basic fundament of DeWiz. The tool itself consists of three main components, the modules, the protocol, and a framework.

The *DeWiz Modules* represent the work units of DeWiz. Each module processes the event graph stream in one way or another. Depending on the particular task of a module, we distinguish between event graph generation, automatic analysis, and data access modules

Event graph generation modules are used at the head of the analysis pipeline to generate the event graph stream for a given program execution. The data is produced either on-line while the monitoring tool is running, or post-mortem by reading corresponding tracefiles. The automatic analysis modules perform the actual operations on the event graph to detect the interesting information, while data access modules are used at the tail of the pipeline to display the results to the user. In most cases, some kind of visualization tool will be used, which draws the resulting event graph as a space-time diagram.

The *DeWiz Protocol* is utilized between modules in order to transport the event graph stream. The protocol defines the particular communication interface (at present, only TCP/IP is supported), and the structure of the transported data.

The *DeWiz Framework* offers the required functionality to implement DeWiz modules for a particular programming language. At present, the complete functionality of DeWiz is supported in Java, while smaller fragments of the analysis pipeline are already available in C. The framework also hides the DeWiz protocol in order to ease the development of DeWiz modules.

Each module of the framework must implement the following functionality: (1) Open event graph stream interface; (2) Filter relevant events for processing; (3) Process event graph stream as desired; (4) Close event graph stream interface.

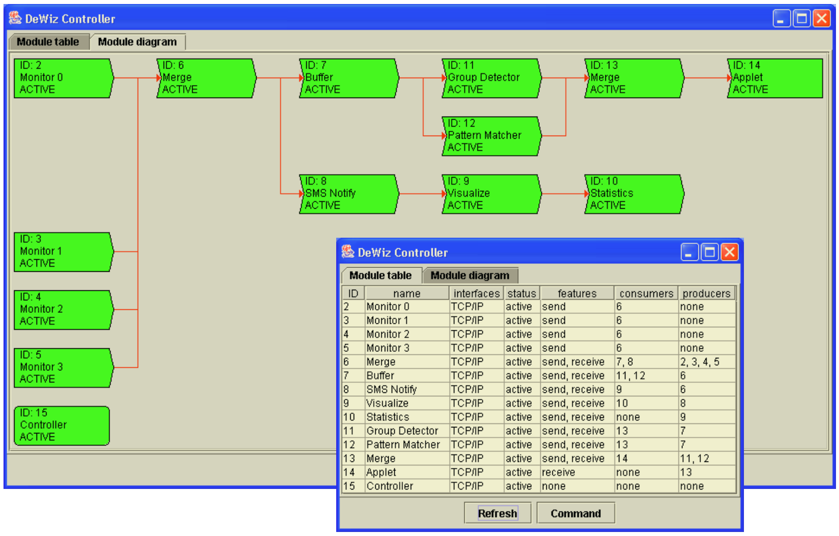
The functions to open and close interfaces are used to establish and destroy interconnections between modules. The interfaces transparently implement the DeWiz protocol, while filtering and processing of events is performed within the main loop of each protocol. With the DeWiz framework, new modules can be implemented by programming the filtering rules and the intended processing functionality. The remaining functionality of the modules is automatically provided by the framework.

If the modules for a concrete analysis task are available, the user may start to construct a corresponding *DeWiz System*. The modules are placed and initialized on arbitrary networked computing nodes. A dedicated module, the *DeWiz Sentinel* is used to control a particular DeWiz system and to coordinate interconnection between modules. Upon initialization, each module connects to the Sentinel and offers its services. With a controller interface, the registered modules may be arbitrarily interconnected by identifying corresponding input and output interfaces.

An example of the DeWiz controller interface is shown in Figure 1. The smaller window in front shows the module table, including all registered modules (by id and name), their available interfaces and status, the implemented features (send, receive, or none), and the id's of corresponding consumer or producer modules. The larger background window of Figure 1 provides the same information in form of a module diagram.

4 Example Event Graph Analysis Activities

For using DeWiz within a particular programming environment, dedicated event graph generation modules have been implemented. A first module represents a post-mortem trace reader for our monitoring tool NOPE [8]. Additionally, an on-line interface for the OMIS Compliant Monitor OCM [14] has been implemented recently. This allows to analyze arbitrary event graph streams on the fly as they are generated by the OCM tool. To test DeWiz for shared-memory programs, event graph generation functionality has been included in the OpenMP Pragma and Region Instrumentor OPARI [12]. With this module, DeWiz can also be applied to analyze set and unset operations in shared memory programs



ID	name	interfaces	status	features	consumers	producers
2	Monitor 0	TCP/IP	active	send	6	none
3	Monitor 1	TCP/IP	active	send	6	none
4	Monitor 2	TCP/IP	active	send	6	none
5	Monitor 3	TCP/IP	active	send	6	none
6	Merge	TCP/IP	active	send, receive	7, 8	2, 3, 4, 5
7	Buffer	TCP/IP	active	send, receive	11, 12	6
8	SMS Notify	TCP/IP	active	send, receive	9	6
9	Visualize	TCP/IP	active	send, receive	10	8
10	Statistics	TCP/IP	active	send, receive	none	9
11	Group Detector	TCP/IP	active	send, receive	13	7
12	Pattern Matcher	TCP/IP	active	send, receive	13	7
13	Merge	TCP/IP	active	send, receive	14	11, 12
14	Applet	TCP/IP	active	receive	none	13
15	Controller	TCP/IP	active	none	none	none

Fig. 1. DeWiz controller showing tool architecture during runtime.

using OpenMP, thus emphasizing the universal applicability of the event graph approach.

In terms of data access modules, DeWiz provides an interface to the analysis tool ATEMPT, a Java applet to display the event graph stream in arbitrary web browsers, and a SMS notifier for critical failures during program execution.

The analysis functionality already implemented in DeWiz is described with the following two examples: Extraction of communication failures and pattern matching and loop detection.

Communication failures can be detected by pairwise analyzing of communication partners. A set of analysis activities for message passing programs is described in [7]. An example is the detection of different message length at send and receive operations. Other examples include isolated send or receive operations, where the communication partner is absent.

A more complex analysis activity compared to the extraction of communication failures is pattern matching and loop detection. The goal of the corresponding DeWiz modules is to identify repeated process interaction patterns in the event graph.

5 Conclusions and Future Work

The DeWiz tool architecture described in this paper offers a universal approach to parallel program analysis. While some analysis tasks are limited to fixed targets, many of them are useful in a broader context. With DeWiz it is possible to reuse a large set of analysis modules for different activities.

Another benefit due to the modular architecture, is scalability. Since DeWiz modules can be placed on arbitrary networked resources, even large amounts of analysis data can be processed, assuming that the computing power is somewhere available. In this context, DeWiz is also a good candidate for grid computing: On the one hand, large-scale grid applications can be analyzed, on the other hand, DeWiz may utilize grids to perform the analysis activities.

The future work in this project is focused on how to formulate event-based analysis techniques within DeWiz modules. At present, the user can rely on a well-defined set of formal graph-based methods, which must be transformed into a set of Java classes. For the future we plan to include some kind of formal language interpreter for this task. Related work in this area, e.g. the Event Analysis and Recognition Language (EARL) [15], or the Event Based Behavioral Abstraction technique EBBA [1], have already hinted at the feasibility of this approach.

Acknowledgments. We are most grateful for the support by our colleagues, Rene Kobler, Johannes Hoelzl, and Bernhard Aichinger.

References

1. P. Bates. *Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior.* ACM Transactions on Computer Systems, Vol. 13, No. 1, pp. 1–31 (February 1995).
2. H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. *Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach.* Proc. ICCS 2001, Intl.Conf. on Computational Science, Springer-Verlag, LNCS, Vol. 2074, San Francisco, CA, USA (May 2001).
3. J. Chassin de Kergommeaux, B. Stein. *Paje: An Extensible Environment for Visualizing Multi-threaded Program Executions.* Proc. Euro-Par 2000, Springer-Verlag, LNCS, Vol. 1900, Munich, Germany, pp. 133–144 (2000).
4. M.T. Heath, J.A. Etheridge. *Visualizing the Performance of Parallel Programs.* IEEE Software, Vol. 13, No. 6, pp. 77–83 (November 1996).
5. P. Kacsuk. *Performance Visualization in the GRADE Parallel Programming Environment.* Proc. HPC Asia 2000, 4th Intl. Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, Peking, China, pp. 446-450 (2000).
6. D. Kranzlmüller, S. Grabner, and J. Volkert. *Debugging with the MAD Environment.* Parallel Computing, Vol. 23, Nos. 1–2, pp. 199–217 (1997).
7. D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs.* PhD thesis, GUP Linz, Joh. Kepler University Linz (September 2000). `http://www.gup.uni-linz.ac.at/∼dk/thesis`.
8. D. Kranzlmüller, J. Volkert. *Debugging Point-To-Point Communication in MPI and PVM.* Proc. EuroPVM/MPI 98, Intl. Conference, Liverpool, GB, pp. 265–272 (Sept. 1998).
9. H. Krawczyk and B. Wiszniewski. *Analysis and Testing of Distributed Software Applications.* Research Studies Press Ltd., Baldock, Hertfordshire, UK (1998).
10. L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, Vol. 21, No. 7, pp. 558–565, (July 1978).
11. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall. *The Paradyn Parallel Performance Measurement Tool.* IEEE Computer, Vol. 28, No. 11, pp. 37–46 (November 1995).

12. B. Mohr, A.D. Malony, S. Shende, and F. Wolf. *Design and Prototype of a Performance Tool Interface for OpenMP*. Proc. LACSI Symposium 2001, Los Alamos Computer Science Institute, Santa Fe, New Mexico, USA (October 2001).
13. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard – Version 1.1*. `http://www.mcs.anl.gov/mpi/` (June 1995).
14. R. Wismüller, J. Trinitis, and T. Ludwig. *OCM – a Monitoring System for Interoperable Tools*. Proc. SPDT 98, 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, ACM Press, Welches, Oregon, USA, pp. 1–9 (August 1998).
15. F. Wolf and B. Mohr. *EARL – A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*. Technical Report FZJ-ZAM-IB-9803, `http://www.kfa-juelich.de/zam/docs/printable/ib/ib-98/ib-9803.ps`, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Germany (1998).
16. J.C. Yan, H.H. Jin, and M.A. Schmidt. *Performance Data Gathering and Representation from Fixed-Size Statistical Data*. Technical Report NAS-98-003, `http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-003.pdf`. NAS Systems Division, NASA Ames Research Center, Moffet Field, CA, USA (1998).