TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

A Framework for Recording and Visualizing Event Traces in Parallel Systems with Load Balancing

Ralf Ebner, Thomas Erlebach, Andreas Ganz, Claudia Gold,
Clemens Harlfinger, and Roland Wismüller

TUM-I9909
SFB-Bericht Nr. 342/05/99 A
Mai 99

TUM–INFO–05-I9909-100/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

A Framework for Recording and Visualizing Event Traces in Parallel Systems with Load Balancing

Ralf Ebner* Thomas Erlebach* Andreas Ganz†
Claudia Gold* Clemens Harlfinger* Roland Wismüller*

May 21, 1999

Contents

*Institut für Informatik, TU München, 80290 München
Email: (ebner|erlebach|gold|harlfing|wismuell)@in.tum.de
†Lehrstuhl für Entwurfsautomatisierung, TU München, 80290 München, ganz@ei.tum.de

1 Introduction

During the development of parallel application programs or load balancing strategies, it would be extremely helpful to have powerful and easy-to-use facilities to observe what is happening at run-time. In particular, many developers have in mind a number of events pertaining to the run of the application that should be recorded on-line and visualized either on-line or *post mortem*. The exact characteristics of events, however, differ among different developers. Hence, it is very important to find a unifying framework for specifying and recording relevant events and for visualizing and analyzing the recorded events in a meaningful way.

In this report, we use data collected from interviews with a number of application developers and developers of load balancing systems in order to specify a general group of event types that are meaningful in many application contexts and that can be customized to fit individual needs. Furthermore, we present a concept for developing a flexible visualization tool that can help in making good use of recorded events.

Our considerations apply to parallel application programs running on a multiprocessor system with distributed memory, e.g., a cluster of Ethernet-coupled workstations or an IBM SP2 parallel computer. A substantial portion of the results are meaningful for different types of parallel systems (like shared-memory multiprocessors) as well.

2 Overview of Parallel Systems and their Analysis Requirements

Within the SFB 342 a lot of projects work on load distribution. In the following we give an overview of these projects and we summarize what kind of tools are already used within these projects to analyze the applied load distribution strategies and mechanisms.

The data we present here were collected from interviews. This technique was chosen to get objective and comparable information which is not only influenced by the developers' viewpoint. Questions of interest were the application domain respectively the load distribution system a project works on, the applied load model and the chosen load distribution mechanisms and strategies. Another topic was the observation of load distribution and the kind of information observation tools should offer.

2.1 Views of Load Distribution

In this section, we shortly describe parallel applications, systems, and tools of projects within the SFB 342 that are concerned with load distribution and balance.

The applications involved are automated theorem proving, test pattern generation for integrated circuits, parallel database systems, adaptive finite element simulations, and cooperative simulation of logic circuits. The systems and tools cover online monitoring with consistent checkpointing, load distribution in operating systems, and adaptive load distribution.

Finally, the requirements of theoretically founded approaches for worst and average case analysis are treated in a dedicated project.

2.1.1 Cooperative Parallel Automated Theorem Proving (project A5)

Finding a proof for the validity of a conjecture in some given theory (e. g. First Order Predicate Logic) is a search problem, with the corresponding search tree being immense.

At present, exactly one prover is encapsulated in a process and represents a load object. In the future, it would be useful to split the proof tree and to treat each layer as an individual load object.

Critical resources are CPU time and main memory. Whereas good distribution of processes to computers with sufficient free memory (typically 10MB or more) is essential, migration of processes would be too expensive. Another criterion for process distribution is CPU speed and user load in order to minimize the response time of the newly spawned prover.

References

[1] M. FUCHS and A. WOLF. System Description: Cooperation in Model Elimination: CPTHEO. In C. KIRCHNER and H. KIRCHNER, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE)*, number 1421 in Lecture Notes in Artificial Intelligence (LNAI), pages 42–46. Springer-Verlag, 1998.

[2] J. SCHUMANN, A. WOLF and C. SUTTNER. Parallel Theorem Provers Based on SETHEO. In W. BIBEL and P. H. SCHMITT, editors, *Automated Deduction. A Basis for Applications. Volume I I*, number 9 in Applied Logic Series, pages 261–290. Kluwer Academic Publishers, 1998.

2.1.2 Automatic Test Pattern Generation (project B1)

During the design process of an integrated circuit, the circuit designer has to generate a set of test patterns. These patterns are applied to the inputs of the manufactured circuit in order to detect circuit malfunctions due to physical defects. For an automatic generation of test patterns so-called fault models are used covering certain physical defects.

The fault model usually used for detection of dynamic defects is the path delay fault model. Since the number of paths in a circuit grows in the worst-case exponentially with the circuit depth the set of modeled faults is typically very large, i.e. up to 10^{20}. Since for each path delay fault a test pattern has to be determined, test pattern generation is an expensive task. Therefore, it is advisable to perform test pattern generation in parallel.

To perform test pattern generation in parallel, the whole set of path delay faults is partitioned into blocks. Since partitioning into blocks can be carried out such that there are only a few dependencies among blocks, they can be administrated in a client-server topology. Thereby, each server processes one block after the other. Additionally, redistribution of blocks is carried out by a work-stealing strategy towards the end of the computation when some computers become idle. The blocks are the load objects, and the remaining block length is a straight-forward load index.

References

[1] A. GANZ. Automatic Test Pattern Generation. In T. SCHNEKENBURGER and G. STELLNER, editors, *Dynamic Load Distribution for Parallel Applications*. (Teubner-Texte zur Informatik), 1997.

[2] P. A. KRAUSS, A. GANZ and K. J. ANTREICH. Distributed Test Pattern Generation for Stuck-At Faults in Sequential Circuits. *Journal of Electronic Testing: Theory and Applications*, pages 227–245, December 1997.

2.1.3 Parallel Database Systems (project B2)

One of the typical parallel applications is the distribution of database queries or parts thereof to several database servers. Each (partial) query is encapsulated in an individual interpreter process. CPU time is not the critical performance factor, but main memory and predominantly network load because of high I/O requirements of database queries. Any load balancing strategy has to prevent swapping. Once a process has been assigned to a machine, it should not be migrated any more, since the amount of allocated main memory and data which would have to be transmitted is too large.

At present a simple scheduler for load distribution and balancing is implemented. It assigns processes to the nodes of the system, considering the current system state.

General purpose tools for load distribution or visualization would have to cope with the wide range of database parameters influencing the query evaluation.

References

[1] G. Bozas, M. Jaedicke, A. Listl, B. Mitschang, A. Reiser and S. Zimmermann. On Transforming a Sequential SQL–DBMS into a Parallel One: First Results and Experiences of the MIDAS–Project. In *Proc. of 2nd Int. Euro–Par Conference, Parallel Processing*, pages 881–886, Lyon, France, September 1996. Spinger–Verlag, LNCS 1123, Berlin.

2.1.4 Distributed Adaptive Finite Element Simulations (project B3)

In technical or physical simulations using the finite element method, the domain of the physical object (whose properties are to be modeled and simulated) is split into so-called elements. The elements do not overlap and can be grouped to larger (meta-)elements. Conversely, existing elements can be split into smaller ones.

In the end, this substructuring procedure yields a tree data structure whose nodes contain a system of linear equations each. The task is to solve the combination of all node equation systems, where a node can be seen as individual load object. The solution is calculated by several complete tree traversals. For one traversal, the load index of a tree node can be derived from the number of unknowns in its equation system. Communication is directed along the tree edges.

In a distributed environment, it is sensible to balance the load of the tree nodes (i.e. migrate tree nodes) each time the tree is modified (extended).

Up to now, a load balancing strategy based on good bisectors has been implemented. In this application, load balance is achieved with respect to both CPU time and main memory consumption, since both values grow proportionally.

References

[1] S. Bischof, R. Ebner and T. Erlebach. Load Balancing for Problems with Good Bisectors, and Applications in Finite Element Simulations. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing, Euro-Par'98*, number 1470 in LNCS, pages 383–389, Berlin, 1998. Springer.

[2] R. EBNER and A. PFAFFINGER. Higher Level Programming and Efficient Automatic Parallelization: A Functional Data Flow Approach with FASAN. In E. D'HOLLANDER, G. JOUBERT, F. PETERS and U. TROTTENBERG, editors, *Parallel Computing: Fundamentals, Applications and New Directions*, volume 12 of *Advances in Parallel Computing*, pages 377–384, Amsterdam, 1998. Elsevier.

2.1.5 Parallel Simulation of Logic Circuits (project B4)

A logic circuit consists of gates as well as signals connecting those gates. The circuits used consist of about 30 K gates. Unfortunately, industry does not provide bigger designs for university research. But experiments show that the developed algorithms scale well to bigger circuits.

A single gate is too small to be an entity for load distribution. So gates and signals are grouped together into partitions before the simulation starts. If signal propagation is simulated in parallel, it is essential to keep track of the virtual time progress (VTP) of each simulator. Big differences in VTP indicate a bad load balance. Then, load is dynamically balanced by the migration of partitions between simulators.

The actually implemented strategy uses the optimistic time warp protocol for synchronisation with incremental state saving and migration. The better the partitions are chosen, distributed and migrated, the smaller the total CPU time usage and the saved states (and hence memory consumption) are.

An important issue in applying general purpose load balancing libraries is the appropriate translation of VTP into more abstract and general load indices.

References

[1] R. SCHLAGENHAFT, M. K. RUHWANDL, C. SPORRER and H. BAUER. Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations. In *ACM/SCS/IEEE Workshop on Parallel and Distributed Simulation (PADS)*, pages 175–180, Lake Placid, New York, june 1995.

[2] R. SCHLAGENHAFT. Dynamischer Lastausgleich optimistisch synchronisierter, verteilter Simulation. In D. P. SCHWARZ, editor, *GI-Workshop Verteilte Simulation und parallele Prozesse*, Dresden, 1999. Arbeitsgemeinschaft Simulation in der Gesellschaft für Informatik. Zur Veröffentlichung angenommen.

2.1.6 The OMIS Project (project A1)

OMIS stands for **O**nline **M**onitoring **I**nterface **S**pecification. Tools that use the OMIS interface are independent of the underlying platform and therefore portable.

For this interface OCM - an **O**MIS **C**ompliant **M**onitoring system - was developed. OCM can be used by several tools, also by load balancers.

A tool for **Co**nsistent **Check**pointing called CoCheck has been integrated into OCM. CoCheck allows the migration of processes of parallel applications on networks of workstations. The migration is done transparently for the application. By integrating CoCheck into OCM the service of transparent process migration is available via the OMIS interface and can be used by scheduling and load balancing tools. For load balancing decisions information about the status of processes is needed. Therefore a **N**ode **S**tatus **R**eporter (NSR) was developed that will also be integrated into OCM.

References

[1] T. LUDWIG, R. WISMÜLLER, V. SUNDERAM and A. BODE. *OMIS — On-Line Monitoring Interface Specification (Version 2.0)*, volume 9 of *Research Report Series, Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universität München*. Shaker, Aachen, 1997.

[2] R. WISMÜLLER, J. TRINITIS and T. LUDWIG. OCM — A Monitoring System for Interoperable Tools. In *Proc. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools SPDT'98*. ACM Press, 1998.

2.1.7 Distributed Operating Systems (project A8)

This project aims at developing a distributed operating system for parallel applications. The parallel activities of a parallel application should be evenly distributed among the available resources by such an operating system.

The approach pursued in this project uses actor spheres. An actor sphere is a set consisting of an active object and several related passive objects. The active object represents an action that needs the passive objects - as for example memory - to be executed. The model of these actor spheres can be understood as an abstraction of the underlying hardware system. The application has to be realized in form of actor spheres.

The scheduler of the distributed operating system maps the actor spheres to hosts where the active objects can be executed. The task of the load management is to choose the most suitable, that means less loaded, host for an actor sphere. For load distribution several actor spheres can be scheduled together. It is also possible to define dependencies between actor spheres (e. g. if there are actor spheres which must be terminated before others).

The load distribution strategies and mechanism should be adjustable to get a flexible and application based system. At the moment a selection of fixed strategies is provided. In the future, the resource management should be able to adjust strategies and mechanisms, too.

References

[1] R. RADERMACHER. *Eine Ausführungsumgebung mit integrierter Lastverteilung für verteilte und parallele Systeme*. Ph.d. thesis, Technische Universität München, 1995.

[2] P. P. SPIES, C. ECKERT, M. LANGE, D. MAREK, R. RADERMACHER, F. WEIMER and H.-M. WINDISCH. Sprachkonzepte zur Konstruktion verteilter Systeme. Technical Report TUM-I9618 and SFB 342/09/96 A, TU München, März 1996.

2.1.8 Adaptive Load Distribution System ALDY (project B5)

ALDY is a library of functions for the integration of load distribution into parallel applications. The focus of the design of ALDY was portability.

To reach portability ALDY does not work with real application objects. Instead the application objects have to be mapped to virtual objects. Furthermore ALDY is not fixed to a specific communication and synchronization platform. This is realized by call back functions which the application programmer must implement. Concerning strategies the idea of ALDY is to provide a collection of different load distribution strategies, where the most suitable one can be chosen at runtime. At the moment only a receiver initiated strategy is implemented where several parameters can be chosen at program start.

The load model of ALDY consists of three classes of virtual objects: *Virtual processes* represent the distributed resources, to which work load is assigned. *Virtual agents* are assigned preemptively to virtual processes, that means they can be migrated between processes. Using the neighborship directive affinities among agents can be defined. *Virtual tasks* represent the subproblems of the application. They are non-preemptively assigned to the agents. Each agent processes one task after the other. That means there is no parallelism within agents, but several agents may be assigned simultaneously to the same host.

References

[1] T. SCHNEKENBURGER. The ALDY Load Distribution System. Technical Report TUM-I9519 and SFB 342/11/95 A, Technische Universität München, May 1995. http://wwwpaul.informatik.tu-muenchen.de/projekte/sfb342/pub/sfb-342-11-95A.ps.gz.

2.1.9 Theory and Algorithms for Load Distribution (project A7)

If load distribution is considered from a theoretical point of view, one is interested in the modeling of load and the development and analysis of load distribution algorithms.

In most cases the load is modeled as a scalar and often as a real value. This simplifies the development of scheduling and partitioning methods, but it is hardly applicable to real applications. Sometimes it is necessary to model more than one resource, therefore a multi-dimensional load value is required. Furthermore, many theoretical analyses do not take affinities between the entities of load distribution into account, e.g., heavily communicating entities.

Current work considers the affinities between tree nodes by embedding tree-like graphs into hypercubes. A related work aims to find embeddings into hypercubes with smaller dimensions. This is needed for mapping problems to existing parallel machines, because the dimension of real hypercubes is limited. In addition, some theoretical load distribution strategies are developed, which are based on approximation algorithms.

Several graph algorithms have already been implemented using C++ and the LEDA library.

References

[1] S. BISCHOF, T. SCHICKINGER and A. STEGER. Load Balancing Using Bisectors – A Tight Average-Case Analysis. In *Proceedings of the 7th Annual European Symposium on Algorithms ESA'99*, 1999. To appear.

[2] V. HEUN. *Efficient Embeddings of Treelike Graphs into Hypercubes.* Berichte aus der Informatik. Shaker Verlag, Aachen, 1996. Dissertation.

2.2 Existing and Expected Tools

Table 1 gives a rough overview of tools which have been developed (D), which are currently used (U), or the use or expansion of which is planned (P) in the individual projects. A dash indicates that this tool type will not be applicable or usable in the project.

The projects can be grouped in three categories: applications, tools, and theory. The first five projects are concerned with applications and therefore (will potentially) use existing or planned tools. The next three projects develop and provide tools. Finally, project A7 plays a special role, providing load distribution

Tool \ Project	A5	B1	B2	B3	B4	A1	A8	B5	A7
Debugger			D/U	U	U	D			
Performance Anal.			D/U			D			P
Visualization			D/U	P		D		D/P	U/P
Animation				P	—			P	P
Trace Files	P		U/P		U			D	(P)
Monitoring	U	U/P	D/U	P	U/P	D	P		P
Checkpointing					P	D			
Load Distribution	P	U	D/U	U	D/U		P	D	D
Load Balancing	–	U/P	D/U	P	D/U		P	D	D

Table 1: Developed, Used, and Planned tool types within the individual projects

and balancing algorithms and theoretical foundations rather than tools. Nevertheless, tools supporting simulations and experimental tests are of interest in this area as well.

3 Relevant Event Types

The previous section presented a number of research projects in parallel computing. Each of the projects requires load balancing in one form or another, but the application contexts vary substantially. Our goal is to design a framework for recording and visualizing load balancing events that is general enough to be applicable and useful for all these applications.

As a first step, representatives from several projects were asked to specify which particular events they would like to monitor and visualize in their specific application context. Although the lists of relevant events provided by the projects differed substantially, a large number of events could easily be classified to belong to the same basic event type. For example, every project was interested in recording events pertaining to the creation or removal of load objects, while the load objects themselves varied substantially (agents, nodes of a finite element tree, processes, etc.). An overview of the events relevant to the individual projects can be found in Section 3.1.

Then, in Section 3.2, we specify the general event types, special instances of which occur in a huge number of different application contexts. The description of the event types is intended to be as general as possible, thus allowing individual event types to be derived from the general event types by customization. We propose an object-oriented framework for specifying which events should be recorded in a particular parallel system running a particular application. The different event types should form a class hierarchy, and users should be able to devise customized event classes by inheriting from the available classes.

3.1 Relevant Events in Different Applications

3.1.1 Automatic Test Pattern Generation (project B1)

The application developed in project B1 can be classified as a client-server type (or task farming) application: clients send tasks to servers and receive back the results. This characteristic is reflected in the events considered relevant by B1:

- state change of client
- state change of server
- sending of tasks from client to server
- sending of results from server to client

The expected benefit of recording these events is a visualization of the states of all processes in the system.

3.1.2 Distributed Adaptive Finite Element Simulations (project B3)

The structure of the load created by the ARESO application of project B3 is an unbalanced binary tree. Its nodes are traversed in top-down and bottom-up direction repeatedly. Nodes are associated with processes, and communication takes place between parents and their children only. The main memory and CPU time requirements of a node are proportional to its size. The tree can grow dynamically through addition of new leaves. The events of interest to B3 are tailored to this special problem structure:

- creation of a new leaf
- enlargement of an internal node
- migration of a node
- beginning/end of local computation within a node

- beginning of a new tree traversal

The goal of analyzing these event traces is to visualize the growth of the tree and the migrations of nodes, and to draw conclusions about the progress of the computation and its various components (communication, local computation, migrations).

3.1.3 Parallel Simulation of Logic Circuits (project B4)

The time warp simulations employed by B4 are carried out by a number of simulators that are implemented as processes running on different machines. Each simulator simulates a part of the circuit and can measure the local progress of virtual time. A global controller initiates migrations if the progress of virtual time is too slow in certain simulators. The events of interest for this application are:

- migration-related events (sending and receiving of the following messages: migration initiation, migration of topology data, migration of simulation state, migration completion, new-location information, event forwarding)
- time warp events (local virtual time, rollback, global virtual time, local virtual time progresses)

The temporal relation among the observed events is expected to provide information about the efficiency of the migration protocol, the progress of the computation, and the mutual influence of the migration mechanism and the time warp protocol.

3.1.4 The OMIS Project (project A1)

Project A1 is concerned with the development of general tools that make developing, tuning, running and monitoring parallel application programs easier. Also, they work on system-integrated support for load balancing. Taking this into account, it is not surprising that they are interested in the following event types:

- creation and termination of load objects (processes, threads, objects), and changes in resource requirements of a load object
- addition and removal of targets (computers, processors); changes in the parameters of a target
- changes in the network load, changes in the load on a target
- login and logout of interactive users

3.1.5 Distributed Operating Systems (project A8)

Like the events specified by A1, the events given by A8 are not tailored to any particular parallel application. The events considered relevant by A8 reflect the characteristics of the distributed operating system designed in their research project. They are interested in the following event types:

- creation and termination of components (local thread, remote thread, etc.)
- usage of a component
- sending or receiving of load indices
- receiving of location information

A8 would be interested in deriving component-specific profiles regarding their usage and the creation of load. This information could then be used in guiding the load distribution strategies.

3.1.6 Adaptive Load Distribution System ALDY (project B5)

ALDY is a library that supports adaptive application-integrated load balancing. The execution of the application program is viewed as tasks being performed by agents, where the agents themselves are executed by processes and represent the units of placement and migration. The events that are considered relevant to observe a running application using ALDY load balancing are:

- creation, termination, and load of processes
- creation, placement, state change, migration, and termination of agents
- neighborship of agents
- buffering of tasks, blocking of agents until arrival of tasks, assignment of tasks to agents
- internal messages of ALDY

Visualization and statistical analysis of the recorded event traces are desired. One goal is assessing the overhead imposed by the ALDY system.

3.1.7 Theory and Algorithms for Load Distribution (project A7)

Project A7 studies load balancing problems and algorithms from a theoretical point of view. Typically, simplified models of real-life parallel systems and applications are assumed, and statements regarding the average-case or worst-case behavior of load balancing strategies are derived. In order to evaluate the performance of load balancing strategies in parallel application programs, the following events are considered relevant by A7:

- creation, migration, state change, and termination of load objects
- regular load measurements on targets
- internal messages of the load balancing strategy

Observing these events is assumed to provide information about the practical efficiency and overhead of the load balancing strategy. In addition, it can help to find suitable and meaningful models for the load created by typical application programs.

3.2 General Event Types

Concerning the variety of events considered relevant by researchers working in different application areas, we observe that a number of general event types (like creation, migration, and termination of load objects) are of interest to almost all projects in one form or another. Therefore, we have devised a general set of event types that can capture all the specific events listed in the previous subsections. The goal was to provide a small, consistent set of event types such that most specialized events can be derived from one of the types in this set.

It is not straightforward to devise such a general set of event types. First, note that even the underlying system model varies substantially between different applications. Therefore, the general event types should not be restricted to handle load in form of threads, processes, etc. Instead, we phrase the event types in terms of load objects and targets. A load object can be placed onto a target or into another load object. In this way, systems involving processes placed on workstations can be modeled with equal ease as tasks placed into agents being executed by processes running on computing nodes.

Every load object, target, or message in the system is assumed to be assigned a unique identifier automatically. This can be realized easily with little additional overhead.

3.2.1 Events Pertaining to Load Objects

Generally speaking, a *load object* is something that can be viewed as coming into existence, consuming resources, and terminating at some time during the execution. Load objects can be processes, threads or objects (in the sense of object-oriented programming languages), for example. Usually some other load object can be identified as the creator of the load object, and upon creation the new load object is placed onto either another load object or on a *target* (see Section 3.2.2). The following types of events pertain to load objects (prefixed with LO, which stands for load object).

- LO_Creation_Event: the load object is created.
  Parameters: id, type, creator, time stamp
- LO_Placement_Event: the load object is placed onto another load object or a target.
  Parameters: id, destination, time stamp
- LO_Migration_Event: the load object is moved from one place to another.
  Parameters: id, source, destination, time stamp
- LO_State_Change_Event: the load object changes its state.
  Parameters: id, old state, new state, time stamp
- LO_Termination_Event: the load object is destroyed.
  Parameters: id, time stamp
- LO_Message_Event: the load object sends a message.
  Parameters: id, type, size, receiver, time stamp
- LO_Create_Neighborship_Event: two or more load objects are considered to be neighboring load objects.
  Parameters: id, ids of the load objects, time stamp
- LO_Destroy_Neighborship_Event: neighborship between load objects is canceled.
  Parameters: id, ids of the load objects, time stamp

The LO_Creation_Event and LO_Placement_Event can be combined to a LO_Creation_Placement_Event in many applications. The LO_Message_Event and the LO_Migration_Event can be specialized into a send event and a receive event (with the “receiver” parameter changed to “sender”), if required.

3.2.2 Events Pertaining to Targets

The targets are the computing nodes (hardware) of the parallel system. They can be workstations or processors in a multiprocessor system, for example.

- Target_Addition_Event: a new target is added to the system.
  Parameters: id, attributes, time stamp.
- Target_State_Change_Event: the state of the target is changed (for example, an interactive user logs in).
  Parameters: id, old state, new state, time stamp.
- Target_Removal_Event: a target is removed from the system.
  Parameters: id, time stamp.
- Target_Create_Neighborship_Event: two or more targets are considered to be neighboring targets.
  Parameters: id, ids of the targets, time stamp.
- Target_Destroy_Neighborship_Event: neighborship between targets is canceled.
  Parameters: id, ids of the targets, time stamp.

3.2.3 Events Pertaining to Load Management

The load management routines can be incorporated into the application code directly, into the runtime-system, or into the operating system. The names of the event types related to load management are prefixed by LM (for load management).

- LM_Target_Load_Event: the load of the target is measured (possibly a regular event).
  Parameters: id, new load, time stamp.
- LM_Network_Load_Event: the load of the network is measured (possibly a regular event).
  Parameters: id, new load, time stamp.
- LM_Message_Event: one component of the load management system sends a message to another component.
  Parameters: id, receiver, type of message, time stamp.

The LM_Message_Event can be specialized into a send event and a receive event (with the "receiver" parameter changed to "sender"), if required.

3.2.4 Application-Specific Events

The event types in the previous three categories should capture most events of interest for a wide variety of parallel computing applications with load balancing. For the case that a certain application-specific event type cannot be derived, however, we provide the following custom event that can be specialized to capture any application-specific event type.

- Custom_Event: application-specific, user-defined event.

This should be needed only rarely, because most application-specific events can be modeled as LO_State_Change_Events.

4 Customization of Trace Events

In the previous section we have presented a collection of general event classes. These event classes can be customized for the different projects to provide appropriate event types. This section will describe the customization process in further detail.

First, in Section 4.1, we will explain how all the specific event types mentioned in Section 3.1 can be viewed as special instances of the general event types listed in Section 3.2. Then, in Section 4.2, we explain in more detail how a customization of the general event types to obtain specific event types can be carried out, and how it interacts with a potential visualization environment.

4.1 Fitting Specific Events into the Framework

For a specific application of our event model, it is basically sufficient to specify which types of load objects and targets occur in the system and which parameters in addition to the standard parameters are required. We will outline the results of this customization for the events considered relevant by the different projects according to Section 3.1.

B1: Regarding the automatic test pattern generation application in project B1, the load object types are servers and clients. The relevant events are then LO_State_Change_Events and LO_Message_Events.

B3: In the ARESO application of B3, the load objects are the nodes of the unbalanced binary tree. Creation of a new leaf is a LO_Creation_Event, migration of a node is a LO_Migration_Event. Enlargement of an internal node and the beginning and end of a local computation are LO_State_Change_-Events. The beginning of a new tree traversal can either be considered a LO_State_Change_Event in the root of the tree, or a Custom_Event.

B4: The load objects in the time warp simulation application of project B4 are the distributed simulators and the central controller. In addition, the partitions of the simulated logic circuit are load objects, which are themselves placed in the simulators. B4 distinguishes migration events regarding the topology data and the simulation state; these event types should both be derived from the LO_Migration_Event. The initiation of a migration, the completion message, the event-forwarding, and the new-location information are all instances of the LM_Message_Event. The events concerning the time warp protocol can be modeled as LO_State_Change_Events (local virtual time, rollback, global virtual time, local virtual time progresses).

A1: The load objects relevant to A1 are processes, threads and objects. The events specified by A1 with respect to load objects belong to the general event types LO_Creation_Event, LO_State_Change_Event, and LO_Termination_Event. Their remaining events are Target_Addition_Event, Target_Removal_Event, Target_State_Change_Event (including login and logout of interactive users), LM_Target_Load_Event, and LM_Network_-Load_Event.

A8: For A8, the relevant types of load objects are the different components in their system model (actor and depot). The relevant events are LO_-Creation_Events and LO_Termination_Events. Every usage of a component can be modeled as two LO_Message_Events, possibly accompanied by LO_State_Change_Events in the client and server object of the usage call. Sending and receiving of load indices and location information are LM_Message_Events.

B5: For applications using ALDY developed by B5, the load objects are tasks, agents, and processes. Most of their events can be seen as LO_Creation_-Events, LO_Termination_Events, LO_Placement_Events, and LO_Migration_Events. In addition, they use LO_Create_Neighborship_Events, LM_Message_Events, and LO_State_Change_Events (including load of processes, blocking of agents).

A7: The events specified by A7 are LO_Creation_Events, LO_Migration_Events, LO_State_Change_Events, LO_Termination_Events, LM_Target_Load_Events, and LM_Message_Events.

4.2 Methodology for Fitting the Event Model to an Application

So far we have discussed which events are considered meaningful for creating event traces by researchers working in different research projects. We derived a general and more abstract set of event classes that can capture most of the specific events through customization, and we have shown how the specific events mentioned by the individual researchers can be incorporated in this general event model.

In this section, we describe our general implementation concept first. Then we show a way how the general event model can be incorporated in a flexible object model. At last, we explain how the customization for a specific project could be carried out in detail.

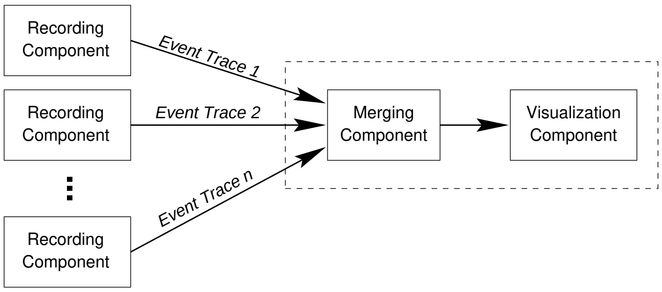
Object-oriented principles are applied frequently in this section. This suggests the usage of OO-languages like C++ or Java. Nevertheless, most of the concepts and models we will describe can be realized with non-OO-languages (C, Fortran, etc.) as well.

4.2.1 Implementation Concept

The implementation of a general framework for tracing events in a parallel system with load balancing requires two main components: an event recording component and an event trace visualization component. The application program must be instrumented using the event recording component, and the resulting traces must be analyzed and visualized (either off-line or on-line) using the visualization component. The main functionality of both components could be realized as Java packages or C++ class libraries, for example.

The event recording component provides one class for each of the general event types mentioned in Section 3.2, plus convenient facilities for recording events into main memory, into a file, or sending them to the visualization component through the network. A recording component provides only local event traces.

The visualization component provides a graphical user interface, a visualization class for each event class (the corresponding name can be obtained by replacing "Event" with "Visualizer" in the event class name), and facilities for reading event traces from main memory, from a file, or receiving them over a network channel. As a conceptual simplification we assume that a visualization component may process only a single stream of event traces. Therefore, to visualize non-local aspects of the system we have to merge all locally recorded event traces into one sequence of events in a consistent way, i.e. the sequence must be complete and adequately ordered (usually based on the events' timestamp). Conceptually, the construction of such a global stream of events is left to an additional merging component. Nevertheless, the realization of the merging may be a part of the visualization component itself.



An example for a typical realization of the visualization component is a customizable stand-alone application program. It presents a view of the parallel system on screen and animates the occurring events through the observed time period. In particular, targets and load objects are depicted on screen using graphical icons, and additional information can be displayed by request of the interactive user of the program. The user possibly can move forward or backward in time (on-line visualization often lacks this feature due to timing restrictions).

The components provided for event recording and visualization should allow to deal with a standard scenario without requiring substantial effort by the application developer. Consider, for example, an application program consisting of processes running on a cluster of workstations. It should be sufficient for achieving a simple visualization if the code of the application program is augmented by calls for recording of LO_Creation_Placement_Events (see Section 3.2.1 for details) and LO_Termination_Events at the points in the code where processes are created or terminated. The visualization component should be able to provide a graphical animation of targets with their assigned processes through the run of the application from this simple event trace. In general, however, an application programmer may be willing to invest more effort into customizing the event recording and visualization components in order to achieve a more application-specific visualization. The methodology for doing this is presented in the Sections 4.2.4 and 4.2.5.

4.2.2 Serialization of Events

Recording of events is mainly a serialization of structured information. We need a way to generate an output stream that contains all the information to rebuild (deserialize) the event structures needed for visualization. Of course, the serialization must be non-ambiguous.

An object-oriented approach to serialize events is to encapsulate the trace generation in the event classes: We assume that every event object has the ability to serialize itself for trace output. This could be done by providing a method write_Trace() or using language-specific serialization methods (e.g. the Serializable interface and the writeObject() method in Java). All parameters of the event (passed to the event object as constructor parameters) may influence the serialization. As a default we serialize the parameters themselves and include their trace output into the event trace. So all parameter objects should be serializable, too.

A simple recording component invokes the serialization for every event object, combines the resulting trace sequence to a data stream, and writes the stream to main memory, a file, or a network port. The corresponding visualization component reads the stream, breaks up the trace sequence into single event traces and deserializes these parts to rebuild the event objects.

Generating an event trace requires information about event parameters. Generally, the gathering of this information is left to the application developer. One way is to directly apply the object model presented in the next subsection. We could generate proxy objects representing the events and their parameters. This approach should simplify the integration of existing recording components significantly. Unfortunately, the explicit object creation may impose an overhead not tolerable for high-performance applications. To avoid this overhead we have to customize the information gathering resp. the recording component.

4.2.3 An Object Model for Events

Based on the event lists presented in Section 3.2 we can build a simple object model to describe events and their parameters that can be easily adapted to the requirements of the different projects. This model could be incorporated in a framework to ease the integration of flexible event tracing for different applications.

The visualization should be defined separately for each component of the model. If the visualization component allows polymorphism we can define generic visualization capabilities for all superclasses of the model that can be reused directly by all derived classes. Thus, for each subclass we just have to develop those parts of the visualization not covered by its superclass.

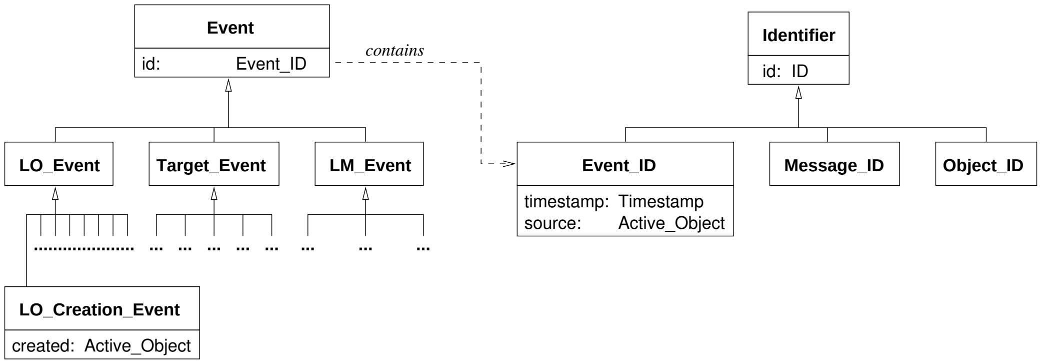
Events Generally, every event trace should provide some basic information:

- a unique identifier for each event (its id)
- the class the event belongs to (its type)
- the time when the event occurred (its timestamp)
- the origin of the event (its source)

We encapsulate all data that are needed to identify an event in a class Event_ID (the event type is implicitly specified for each event class). This allows us to change the information provided by event identifiers without touching the event classes themselves. An example for the integration of extended event identifiers can be found in Section 4.2.5.

The id is inherited from a superclass Identifier. We override its initialization method in Event_ID to include information about the time (timestamp) and place (taken from source) of the event creation in the event identifier itself. We can reuse the Identifier class resp. its subclasses for all other components of our model that incorporate identifiers.

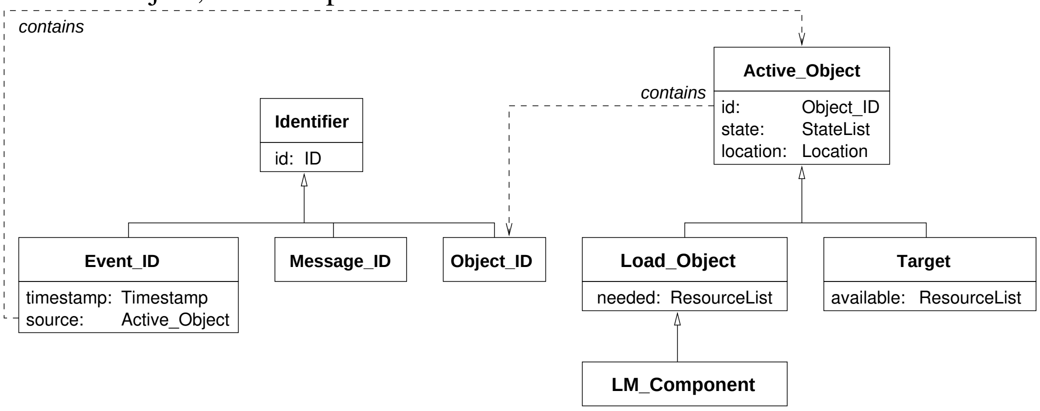
The three different superclasses of events can be derived easily from the Event class. Our general event classes are direct subclasses of these superclasses (see Section 4.2.4 for details on event subclassing).



Active Objects The source attribute that we have introduced before references a load object or a target. Additionally, some event types contain information about other load objects or targets, for example the destination attribute in a LO_-Migration_Event. Therefore, it makes sense to model load objects and targets as separate objects. This allows us to reuse the event creation/visualization classes even if the load objects' and/or targets' structure has changed substantially. In this case, only the creation/visualization of the load object/target classes must be adapted.

Load objects and targets can be seen as different types of active objects. Some general event classes like LO_Placement_Event don't make a distinction between these two types of objects. So we create Active_Object as a common superclass for the classes Load_Object and Target. With event creation/visualization based on active objects we can treat load objects and targets in a common way. For a more specialized object handling for one of the classes we can subclass it and override its defaults (the usage of such subclasses is described in the Sections 4.2.4 and 4.2.5).

The locations of active objects are essential for analyzing both static and dynamic aspects of load distribution. Therefore, any active object should offer some information where it is placed in the distributed system. Locations can be defined in many ways: targets may be referenced with an IP address and a load object's location could be determined by referencing the active object that currently holds this load object, for example.



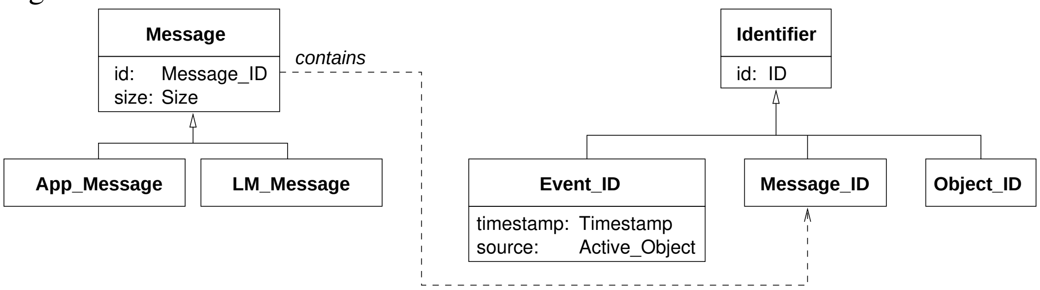
Load management components (for monitoring, distribution strategies, etc.) produce some overhead. Therefore, they are modeled as a specialized form of load objects (LM_Component). This way they can be handled as usual load objects without restricting a separate analysis of the load management's impact on the application performance.

Resources and Object States The event types LO_State_Change_Event and Target_State_Change_Event signal state changes for active objects, so we need a class to represent the state of an active object. This is done with the StateList class that contains a list of state information. The ResourceList class can be used in a similar way to model the resources needed by load objects resp. provided by targets. The main difference between resources and states is that states may change their value dynamically (causing an event) while resources normally are fixed. The design of the ResourceList and StateList classes and their components depends on the application resp. the used load distribution system and will not be investigated further in this text.

Resource lists may be used to model connections between active objects. Thus we are able to model events for establishing/destroying neighborship relations by creating/deleting a resource object that represents the connection between two neighbors. To model dynamic changes of the network load it would be necessary to add state information to the connection resource object.

We can create an event for a state change simply by specifying its new state. The current state may be referenced with the source attribute from the event identifier id. Again, differently designed state information can be created and visualized without touching the event classes themselves.

Messages The transfer of messages can be modeled with a simple Message class. As messages are transferred between load objects (assuming that load management components are specialized load objects), we just augment the event classes for messaging with the identifier of the load object to/from which the message is sent/received.



The other classes used in the model (Timestamp, Size etc.) are not described here in further detail. These classes should be designed according to the given applications resp. to the requirements of the visualization.

Developing Trace Formats with the Object Model We don't recommend one general trace format to be used for all event tracing. The choice of a trace format appropriate to a particular purpose depends on many factors. Usually there's a tradeoff between the expressive power of a trace format and the overhead for its creation, transfer, and visualization. Both the traced application and the desired visualization for this application must be analyzed carefully to build a suitable trace format with appropriate recording and visualization components.

The object model presented before can be used as a basic reference for own trace formats. Single components of the model can be serialized only partially or may even be completely ignored if they are not needed further. Trace recording and/or visualization according to our model can also be used with programming languages without object-oriented concepts, like C. This requires a mapping of the object structures to the corresponding structures of the application/visualization program.

4.2.4 Customization of Event Classes

The event classes from Section 3.2 are part of the object model from the previous subsection. In Section 4.1 we have described how these event classes fit into the different projects. Now we describe how to customize the general event classes.

If the customization of an event class can be reduced to an adaption of its parameters (for example, introducing specialized versions of load objects, targets, or messages), the customization can be achieved by subclassing the corresponding parameter type. In this case, no customization of the event class itself is necessary (we assume that polymorphism is supported). We just could call the event constructor with the changed parameter.

For example, if the application involves load objects in the form of processes and objects, the following specification might be sufficient:

- LO_Object (derived from Load_Object)
- LO_Process (derived from Load_Object)

If additional parameters are required for a specific event class, customizing the recording and visualization components requires the specification of a new event class. The new event class is created by subclassing an existing event class. A new visualization class should be created by subclassing the corresponding class in the visualization component as well.

For example, we consider a scenario where the LO_Creation_Event corresponding to process creation (the attribute created references an LO_Process object) should also record the load object that requested the process creation. created references the initiator of the process creation while the location of the process creation can be retrieved from the source attribute (part of the event's id). In this case, the following subclasses might be created:

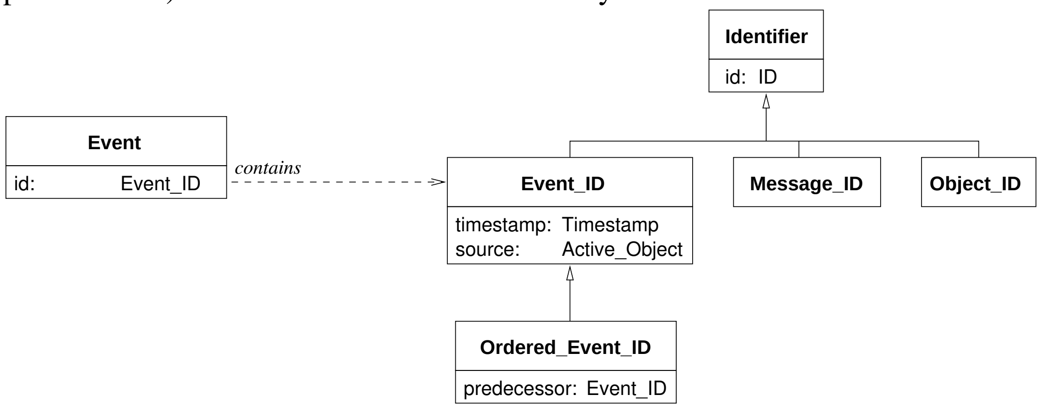
- Process_Creation_Event (derived from LO_Creation_Event)
- Process_Creation_Visualizer (derived from LO_Creation_Visualizer)

Creation of these two subclasses should be sufficient to allow recording and visualizing the new event class Process_Creation_Event: In the application code, the programmer will insert code to record Process_Creation_Event events, and the visualizer will automatically be able to display the additional parameter (for example by drawing a colored line between the process and the requesting load object).

4.2.5 Extensions of the Object Model

In the previous subsection we described how to customize single event classes. If we want to add features that affect the whole event class hierarchy (or at least large parts of it) we must extend the model appropriately. Again, we try to keep the changes in the recording and visualization components as small as possible.

As an example for such an extension we assume a visualization tool that shows dependencies between causally ordered events. To integrate the ordering in our model in a simple way we can derive a new identifier class Ordered_Event_ID from Event_ID. The new class additionally references the predecessor of the event according to the causal ordering. More sophisticated extensions (e.g. a list of predecessors) could be realized in a similar way.



All we have to do for the new class itself is to extend the recording and visualization methods inherited from Event_ID. A more demanding task is the modification of the recording component so that all events automatically will have an id with the type Ordered_Event_ID. To avoid the derivation of all existing event classes we must find a way to create Ordered_Event_ID objects instead of Event_ID objects. The obvious place for this change is the Event class where the id attribute is specified. If we don't want to change the type of the id attribute directly, we must change our program at all places where objects of the type Event_ID are explicitly created to be used later as id attribute in an event object.

The creation of traces based on an extended object model is quite simple if the recording component is designed to be flexible and adaptable. A general design principle to mention here is to avoid hard-wired references to classes/objects that may be changed in the future. This decoupling is normally done by introducing additional levels of indirection.

An example for this technique is the use of factories instead of direct constructor calls. To integrate a derived class we just have to modify the factory for this class to produce objects of the derived class (assuming that all hard-wired constructor calls in the program are replaced by factory calls). Thus we can reduce the number of code changes. We can use this approach if we want to integrate the Ordered_Event_ID class into our recording component without changing the data type of the id attribute in the Event class.

The design of an adaptable recording component may be a difficult task. Many design decisions for better adaptivity induce a performance penalty because of the additional levels of indirection. The resulting overhead may be prohibitive for some applications. Yet, flexible generation of event traces is essential to many of our projects.

5 Related Work

Systems to generate and visualize event traces now exist for virtually every parallel computing environment. Examples of such systems are AIMS [4], Pablo [3] or VAMPIR [1]. These systems have in common that they are designed for performance analysis only. So the types of events that can be monitored only partially overlap with the event types defined in Section 3.2. In particular, the LO_Migration_Event and the LM_Target/Network_Load_Event are not supported.

The trace systems are typically targeted towards the monitoring of the low-level message passing instead of the higher level views outlined in Section 2.1. They do not employ a general object and event model that can be customized to different environments. Pablo with its accompanying trace format SDDF [2] is in some respect an exception. SDDF is a generic trace format, together with libraries to store a trace-file and to read it again in a visualization tool. It allows to store any kind of events, as the event types including their parameters are explicitly defined within the trace file itself (SDDF stands for Self Defining Data Format). The drawback of SDDF is that these definitions only specify the syntactical structure of an event, without providing information on the semantics. Thus, a trace processing tool still needs detailed knowledge about the environment that created the trace file. In the object-oriented approach outlined in Section 4.2, information about the semantics of a specific event type is made explicit by deriving it from one of the general event types introduced in Section 3.2. The approach therefore helps to overcome the problem of the dependency between the environment generating a trace and the tools processing this trace, provided that it can be implemented with sufficient efficiency.

References

[1] A. ARNOLD, U. DETERT and W. E. NAGEL. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. WINGET and K. WINGET, editors, *Proc. Cray Users' Group Meeting, Spring 1995*, pages 252–258, Denver, CO, March 1995.

[2] R. A. AYDT. *The Pablo Self-Defining Data Format*. Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois 61801, November 1997.

[3] D. A. REED, R. D. OLSON, R. A. AYDT, T. M. MADHYASTA, T. BIRKETT, D. W. JENSEN, A. A. NAZIEF and B. K. TOTTY. Scalable Performance Environments for Parallel Systems. In *Proc. 6th Distributed Memory Computing Conference*. IEEE Computer Society Press, 1991.

[4] J. C. YAN, S. R. SURUKKAI and P. MEHRA. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, 25(4):429–461, April 1995.

6 Summary

The projects within the SFB 342 cover many different aspects of load balancing (resp. load distribution), including application and tool development as well as theoretical research. The different approaches result in a large variety of project-specific event types and corresponding trace formats. Unfortunately, no standardized trace format based on a common set of event types had been defined so far for the various projects. Tools for trace recording and visualization were developed only for single projects. A reuse by other projects normally wasn't possible without considerable customization effort. Our approach to improve this situation is documented in this report.

As a first step, general information about project-specific events related to load balancing was collected in interviews. Then we analyzed the interviews to extract a list of the essential events for each project. Based on these event lists we constructed a common set of event classes covering the needs of all participating projects. This set is divided into three major categories to distinguish between events related to load objects (using resources), targets (providing resources) and load management components.

The integration of different trace formats can be simplified by encapsulating trace recording and visualization into separate components. These components can be independently attached to the application resp. to an appropriate tool for trace analysis. To simplify the construction and reuse of tools for the analysis of event traces we built a class hierarchy that models the common set of events in an object-oriented way. The object model allows us to enhance event classes or add new ones while existing tools can still be used without any change. This allows the development of a flexible framework for trace recording and visualization.

Based on this report a prototypic implementation of the aforementioned framework should be possible. A basic set of visualization components to provide some core functionality (e.g. to display sequences of events) will be a central part of the framework. Additionally, a general-purpose trace recording component should be included to alleviate the instrumentation of applications. Further extensions of the framework can be applied by the different projects individually.

Acknowledgments

The research presented in this report was carried out with our colleagues in the ALV project of SFB 342: Stefan Bischof (A7), Joachim Dräger (A5), Michael Jaedicke (B2), Ursula Maier (A1), Clara Nippl (B2), Stefan Petri (A1), Markus Pizka (A8), Günther Rackl (A1), Christian Rehn (A8), Niels Reimer (A8), Rolf Schlagenhaft (B4), Andreas Wolf (A5), and Stephan Zimmermann (B2). We gratefully acknowledge their cooperation in the interviews and their useful suggestions and stimulating comments.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

Liste aller erschienenen Berichte von 1990-1994
auf besondere Anforderung

342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik
342/17/95 A Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications
342/19/95 A Thomas Schnekenburger: Integration of Load Distribution into ParMod-C

Reihe A

342/20/95 A	Ketil Stølen: Refinement Principles Supporting the Transition from Asynchronous to Synchronous Communication
342/21/95 A	Andreas Listl, Giannis Bozas: Performance Gains Using Subpages for Cache Coherency Control
342/22/95 A	Volker Heun, Ernst W. Mayr: Embedding Graphs with Bounded Treewidth into Optimal Hypercubes
342/23/95 A	Petr Jančar, Javier Esparza: Deciding Finiteness of Petri Nets up to Bisimulation
342/24/95 A	M. Jung, U. Rüde: Implicit Extrapolation Methods for Variable Coefficient Problems
342/01/96 A	Michael Griebel, Tilman Neunhoeffer, Hans Regler: Algebraic Multigrid Methods for the Solution of the Navier-Stokes Equations in Complicated Geometries
342/02/96 A	Thomas Grauschopf, Michael Griebel, Hans Regler: Additive Multilevel-Preconditioners based on Bilinear Interpolation, Matrix Dependent Geometric Coarsening and Algebraic-Multigrid Coarsening for Second Order Elliptic PDEs
342/03/96 A	Volker Heun, Ernst W. Mayr: Optimal Dynamic Edge-Disjoint Embeddings of Complete Binary Trees into Hypercubes
342/04/96 A	Thomas Huckle: Efficient Computation of Sparse Approximate Inverses
342/05/96 A	Thomas Ludwig, Roland Wismüller, Vaidy Sunderam, Arndt Bode: OMIS — On-line Monitoring Interface Specification
342/06/96 A	Ekkart Kindler: A Compositional Partial Order Semantics for Petri Net Components
342/07/96 A	Richard Mayr: Some Results on Basic Parallel Processes
342/08/96 A	Ralph Radermacher, Frank Weimer: INSEL Syntax-Bericht
342/09/96 A	P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch: Sprachkonzepte zur Konstruktion verteilter Systeme
342/10/96 A	Stefan Lamberts, Thomas Ludwig, Christian Röder, Arndt Bode: PFSLib – A File System for Parallel Programming Environments
342/11/96 A	Manfred Broy, Gheorghe Ştefănescu: The Algebra of Stream Processing Functions
342/12/96 A	Javier Esparza: Reachability in Live and Safe Free-Choice Petri Nets is NP-complete
342/13/96 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Many-to-Many Data-flow Networks
342/14/96 A	Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project
342/15/96 A	Richard Mayr: A Tableau System for Model Checking Petri Nets with a Fragment of the Linear Time μ -Calculus

Reihe A

342/16/96 A	Ursula Hinkel, Katharina Spies: Anleitung zur Spezifikation von mobilen, dynamischen Focus-Netzen
342/17/96 A	Richard Mayr: Model Checking PA-Processes
342/18/96 A	Michaela Huhn, Peter Niebert, Frank Wallner: Put your Model Checker on Diet: Verification on Local States
342/01/97 A	Tobias Müller, Stefan Lamberts, Ursula Maier, Georg Stellner: Evaluierung der Leistungsfähigkeit eines ATM-Netzes mit parallelen Programmierbibliotheken
342/02/97 A	Hans-Joachim Bungartz and Thomas Dornseifer: Sparse Grids: Recent Developments for Elliptic Partial Differential Equations
342/03/97 A	Bernhard Mitschang: Technologie für Parallele Datenbanken - Bericht zum Workshop
342/04/97 A	nicht erschienen
342/05/97 A	Hans-Joachim Bungartz, Ralf Ebner, Stefan Schulte: Hierarchische Basen zur effizienten Kopplung substrukturierter Probleme der Strukturmechanik
342/06/97 A	Hans-Joachim Bungartz, Anton Frank, Florian Meier, Tilman Neunhoeffer, Stefan Schulte: Fluid Structure Interaction: 3D Numerical Simulation and Visualization of a Micropump
342/07/97 A	Javier Esparza, Stephan Melzer: Model Checking LTL using Constraint Programming
342/08/97 A	Niels Reimer: Untersuchung von Strategien für verteiltes Last- und Ressourcenmanagement
342/09/97 A	Markus Pizka: Design and Implementation of the GNU INSEL-Compiler gic
342/10/97 A	Manfred Broy, Franz Regensburger, Bernhard Schätz, Katharina Spies: The Steamboiler Specification - A Case Study in Focus
342/11/97 A	Christine Röckl: How to Make Substitution Preserve Strong Bisimilarity
342/12/97 A	Christian B. Czech: Architektur und Konzept des Dycos-Kerns
342/13/97 A	Jan Philipps, Alexander Schmidt: Traffic Flow by Data Flow
342/14/97 A	Norbert Fröhlich, Rolf Schlagenhaft, Josef Fleischmann: Partitioning VLSI-Circuits for Parallel Simulation on Transistor Level
342/15/97 A	Frank Weimer: DaViT: Ein System zur interaktiven Ausführung und zur Visualisierung von INSEL-Programmen
342/16/97 A	Niels Reimer, Jürgen Rudolph, Katharina Spies: Von FOCUS nach INSEL - Eine Aufzugssteuerung
342/17/97 A	Radu Grosu, Ketil Stølen, Manfred Broy: A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing
342/18/97 A	Christian Röder, Georg Stellner: Design of Load Management for Parallel Applications in Networks of Heterogenous Workstations
342/19/97 A	Frank Wallner: Model Checking LTL Using Net Unfoldings

Reihe A

342/20/97 A	Andreas Wolf, Andreas Kmoch: Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung zur Lösung eines Beispiels aus der Hardware-Verifikation – Fallstudie –
342/21/97 A	Andreas Wolf, Marc Fuchs: Cooperative Parallel Automated Theorem Proving
342/22/97 A	T. Ludwig, R. Wismüller, V. Sunderam, A. Bode: OMIS - On-line Monitoring Interface Specification (Version 2.0)
342/23/97 A	Stephan Merkel: Verification of Fault Tolerant Algorithms Using PEP
342/24/97 A	Manfred Broy, Max Breitling, Bernhard Schätz, Katharina Spies: Summary of Case Studies in Focus - Part II
342/25/97 A	Michael Jaedicke, Bernhard Mitschang: A Framework for Parallel Processing of Aggregat and Scalar Functions in Object-Relational DBMS
342/26/97 A	Marc Fuchs: Similarity-Based Lemma Generation with Lemma-Delaying Tableau Enumeration
342/27/97 A	Max Breitling: Formalizing and Verifying TimeWarp with FOCUS
342/28/97 A	Peter Jakobi, Andreas Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation)
342/29/97 A	Radu Grosu, Ketil Stølen: Compositional Specification of Mobile Systems
342/01/98 A	A. Bode, A. Ganz, C. Gold, S. Petri, N. Reimer, B. Schiemann, T. Schnekenburger (Herausgeber): "Anwendungsbezogene Lastverteilung", ALV'98
342/02/98 A	Ursula Hinkel: Home Shopping - Die Spezifikation einer Kommunikationsanwendung in FOCUS
342/03/98 A	Katharina Spies: Eine Methode zur formalen Modellierung von Betriebssystemkonzepten
342/04/98 A	Stefan Bischof, Ernst W. Mayr: On-Line Scheduling of Parallel Jobs with Runtime Restrictions
342/05/98 A	St. Bischof, R. Ebner, Th. Erlebach: Load Balancing for Problems with Good Bisectors and Applications in Finite Element Simulations: Worst-case Analysis and Practical Results
342/06/98 A	Giannis Bozas, Susanne Kober: Logging and Crash Recovery in Shared-Disk Database Systems
342/07/98 A	Markus Pizka: Distributed Virtual Address Space Management in the MoDiS-OS
342/08/98 A	Niels Reimer: Strategien für ein verteiltes Last- und Ressourcenmanagement
342/09/98 A	Javier Esparza, Editor: Proceedings of INFINITY'98
342/10/98 A	Richard Mayr: Lossy Counter Machines
342/11/98 A	Thomas Huckle: Matrix Multilevel Methods and Preconditioning
342/12/98 A	Thomas Huckle: Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning

Reihe A

342/13/98 A	Antonin Kucera, Richard Mayr: Weak Bisimilarity with Infinite-State Systems can be Decided in Polynomial Time
342/01/99 A	Antonin Kucera, Richard Mayr: Simulation Preorder on Simple Process Algebras
342/02/99 A	Johann Schumann, Max Breitling: Formalisierung und Beweis einer Verfeinerung aus FOCUS mit automatischen Theorembeweisern – Fallstudie –
342/03/99 A	M. Bader, M. Schimper, Chr. Zenger: Hierarchical Bases for the Indefinite Helmholtz Equation
342/04/99 A	Frank Strobl, Alexander Wisspeintner: Specification of an Elevator Control System
342/05/99 A	Ralf Ebner, Thomas Erlebach, Andreas Ganz, Claudia Gold, Clemens Harlfinger, Roland Wismüller: A Framework for Recording and Visualizing Event Traces in Parallel Systems with Load Balancing

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS