In: Proceedings of the Workshop "Environments and Tools For Scientific Parallel Computing", published in: J.J. Dongarra, B. Tourancheau (Editors), Advances in Parallel Computing, Vol. 6, pp. 103–124, Elsevier, 1993.

Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?

Bernd Mohr[a] [1]

[a]Universität Erlangen-Nürnberg, IMMD 7, Martensstr. 3, D-8520 Erlangen, Germany
email: mohr@immd7.informatik.uni-erlangen.de

Abstract

Programming non-sequential computer systems is hard! Many tools and environments have been designed and implemented to ease the use and programming of such systems. The majority of the analysis tools is event-based and uses event traces for representing the dynamic behavior of the system under investigation, the object system. Most tools can only be used for one special object system, or a specific class of systems such as distributed shared memory machines. This limitation is not obvious because all tools provide the same basic functionality.

This article discusses approaches to implementing object-independent event trace monitoring and analysis systems. The term object-independent means that the system can be used for the analysis of arbitrary (non-sequential) computer systems, operating systems, programming languages and applications. Three main topics are addressed: object-independent monitoring, standardization of event trace formats and access interfaces and the application-independent but problem-oriented implementation of analysis and visualization tools. Based on these approaches, the distributed hardware monitor system ZM4 and the SIMPLE event trace analysis environment were implemented, and have been used in many 'real-world' applications throughout the last three years. An overview of the projects in which the ZM4/SIMPLE tools were used is given in the last section.

1. Introduction

Over the last decade, the development of various high-performance parallel and distributed computer systems has progressed at an explosive rate. Their computation speed can outperform state-of-the-art serial supercomputers, and they are far less expensive. However, software for driving these parallel machines is still in its infancy. Programming parallelism can be very painful and frustrating. In addition, debugging a parallel program and searching for performance bottlenecks is a difficult and time-consuming process.

[1]This work was supported by the German Science Foundation (DFG) under contract number SFB 182, project C1, and partly under the SUPRENUM and the IBM ENC PACS project.

Many projects from University and research institutions have developed and implement tools and environments to ease the use and programming of parallel systems. Dozens (hundreds?) of parallel programming tools are being developed and some of them are becoming commercially available (a survey on parallel debugging tools [16] which is already three years old, lists 28 important tools). The majority of the analysis tools is event-based and uses event traces for representing the dynamic behavior of the system under investigation, the *object system*. In the following, we will call such a tool (environment) an *event trace monitoring and analysis system*. Each system has its own design goals and philosophy for solving a particular class of problems on a particular class of parallel machines. Due to the diversity of tools and complex parallel computer platforms, using these tools often results in confusion and frustration. Additionally, the user has to learn and use different tools when working with more than one object system.

The limitation on particular problem classes and machines is not obvious because all tools comprise the same basic functionality. Therefore, in this article we discuss approaches to implementing object-independent event trace monitoring and analysis systems. *Object-independent* means that the system can be used for the analysis of arbitrary (non-sequential) computer systems with arbitrary operating systems, programming languages and running different applications. This means that there is no need to change the program code of the analysis system and recompile it, when a different measurement has to be analyzed.

In order to allow a systematic and structured discussion, a hierachical layered model for event trace monitoring and analysis systems is introduced first. This model shows that there are three main components in such a system which are affected by the problem of object-independence. They are discussed in the following three sections. Section 3 deals with some aspects of object-independent monitoring. In section 4 we discuss different approaches to standardize the access to event traces, as standardization would allow object-independent tools to be developed, and it would also ease the sharing and exchange of traces and of the tools themselves. Then we will introduce our own proposal: the object-independent TDL/POET event trace access interface. In section 5 we present our approach to application-independent but problem-oriented implementation of analysis tools. The distributed hardware monitor system ZM4 and the SIMPLE event trace analysis environment were implemented with respect to these considerations, and have been used in many 'real-world' applications throughout the last three years. An overview of the projects in which the ZM4/SIMPLE tools were used is given in the last section.

2. A Hierarchical Layered Model for Event Trace Monitoring and Analysis Systems

In this section we will introduce a hierarchical layered model for event trace monitoring and analysis systems. It will allow a systematic and structured discussion of the problem of object-independence. The model is based on investigations of the structure and features of monitoring and analysis systems described in the literature, and on experiences from implementing our ZM4/SIMPLE environment. Each layer provides a higher level of abstraction than the level below. In fig. 1, the six layers, together with the abstraction each layer provides, are shown. The different layers provide the following function:

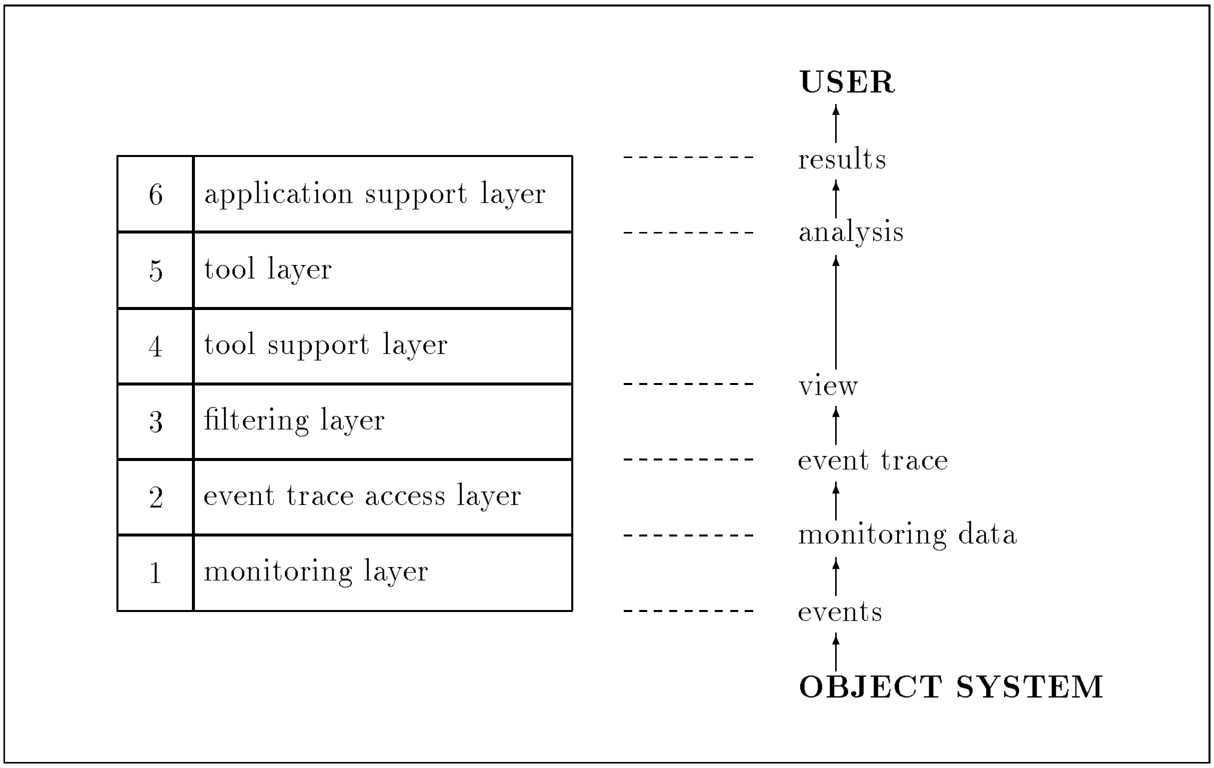


Figure 1. Hierarchical Layered Model

layer 1: monitoring
The task of the monitoring layer is to recognize the events defined for the object system and to store all the data necessary for later analysis. It should also provide a global time base (virtual or real) to allow the ordering of all events with global interdependence. The interface to the object system depends on the chosen monitoring technique (hardware, software or hybrid) and on the properties of the object system itself.

layer 2: event trace access
This layer performs the mapping of bits and bytes of the monitoring data to the abstraction of an event trace, i.e. a sequence of event records each describing the properties of one event occurred. The structure, format and physical representation of the monitoring data is hidden for the upper layers.

layer 3: filtering
Normally, an event trace will contain more information than needed for performing a particular analysis. The filtering layer allows to define a so-called *view* on an event trace. A view is defined by *filtering* and *clustering* events [1]. Filtering deletes all but a designated subset of events from the trace. By clustering events, a sequence of events is regarded as a single higher level event, which we will call *activity* in the following.

layer 4: tool support

This layer contains all functional modules which are independent of the semantics of the object system and of the analysis. It provides often used functions which are needed for the modules of the tool layer, such as graphical displays or statistical functions.

layer 5: tool

The tool layer implements the different possibilities of event trace analysis: validation of event traces, statistical evaluation, visualization of the system's behavior, animation, sonification and extraction of information necessary for other tools, such as modeling environments, debuggers or load balancing tools. The tools can provide some predefined analyses or can be programmable or even interactive.

layer 6: application support

The application support layer constitutes the interface between the "customer" of the analysis and the event trace analysis system. In case of a human user, this layer is implemented by the user interface, i.e. it includes modules such as on-line help, result explanation or interpretation and event trace administration.

Of course, this model presents a rather idealistic view. In reality, one will often find systems in which layers are missing or combined together in order to achieve better performance. In [21] the hierarchical layered model is discussed in much more detail. In particular, the functional modules of each layer and their interdependencies are described. Besides this, a classification scheme based on the layered model is introduced, which allows a comparison and rating of different event trace monitoring and analysis systems.

Coming back to the problem of object-independence, a closer look at the layered model shows that there are three components in a monitoring and analysis system which are affected by that problem: (i) the interface between the monitor system and the object system, (ii) the event trace access and (iii) the analysis tools. These three topics will be discussed in more detail in the next three sections.

3. Layer 1: Monitoring Systems

In this section we will discuss two aspects of the monitoring layer: the interface between the object system and the monitor system and the implementation of an object-independent monitor system itself.

3.1. Hybrid Monitoring

The first step during the analysis of any computer system using monitoring is the recording of data. For this, software, hardware or hybrid monitoring can be used. With hybrid monitoring, the recognition of events is done in software, but the recording and time stamping is done in hardware. This introduces a minimal delay only (unlike software monitoring), but establishes a relation between the measured data and the program under investigation (unlike hardware monitoring). Thus, the advantages of both software and hardware monitoring are combined. A fundamental problem of hybrid monitoring which remains, is the adaptation of the hardware monitor to the object system, i.e. to find a possibility to transmit the information about the event occurred from the object system to the monitor. Normally, parallel or distributed systems do not have such interfaces. Therefore, other interfaces are often "misused" for this purpose. There are basically three possibilities:

direct: If an object system has a special hardware interface which can also be used for monitoring, we will speak of *direct adaptation*. This interface can be a special monitor register, a status display or a field of LEDs. But the application program must be able to access this interface. In this case, the event data can be sent directly to the interface (without using the operating system) and therefore is very fast. The disadvantage is often the limited width of the output data (4 to 8 Bit). An example of direct adaptation is described in [6].

output: With *output adaptation*, an ordinary output interface for printing or communication is used for monitoring. Such an interface often exists or can be bought. However, this method should only be used if this interface can be accessed without involving the operating system. Otherwise the output of event data takes too much time and the application program is delayed more than acceptable. In some cases this can change the dynamic behavior of the program. Oehlrich and Quick [24] used a serial interface (Transputer link) for monitoring.

bus: *Bus adaptation* is possible if the internal bus of the object system is externally accessible. The idea is very simple: with a machine instruction "`STORE Addr, Value`" the event information is transmitted via the internal bus. The address which is otherwise not used or physically not exists, encodes the event type, and the value can be used for event attributes. A simple bus adapter, consisting of an address comparator and two registers, catches and stores the events and the additional data. This is very fast, and normally 16 or 32 Bit for address and data can be used. The disadvantage is that extra hardware has to be built. Examples for a bus adaptation are a first version of the TRAMS monitor [18] and the TMP [4].

Output adaptation and bus adaptation have another advantage: if the object system or its clusters are bus-coupled, the whole system or cluster can be monitored with one interface connected with the global bus. This may reduce the expense in hardware to a great extent.

A standard for hybrid monitoring could define the width of the address and data lines, the control signals and the electrical behavior of the interface. In [14] Malony and Nichols proposed such an "**E**xternal **H**ardware **I**nstrumentation **I**nterface (EHII)". The acceptance of such a standard depends on the willingness of the hardware vendors to provide a monitoring interface for the users. Therefore, a standard appears unrealistic.

Our hardware monitoring system ZM4, which is described in the following section, provides a parallel 48 Bit wide interface for hybrid monitoring. Four additional signal lines can be used to indicate the occurrence of an event so that four independent event streams can be monitored. A prototype was implemented in 1989 and used in several projects. We implemented each method of hybrid adaptation at least once (see table 1 in section 6 for a summary). The performed projects showed that a standard interface such as the one proposed by Malony and Nichols and used in our ZM4, is working and can be used to monitor all types of parallel object systems with minimal object-specific hardware. The table also shows that bus adaptation is the best approach concerning the delay and the maximum width of the output data. With regard to the output speed (Bit/s) the bus adaptation is 250 to 1,500 times faster than direct adaptation, and 1,000 to 20,000 times faster than output adaptation.

3.2. The Monitor System ZM4

In general, hybrid monitoring is the best technique for monitoring computer systems. Unlike pure software monitoring, it needs an extra hardware monitor. But the hardware monitor can be implemented in an object-independent way except for the adaptation to the object system. In using a standard hybrid interface, many types of parallel object systems can be monitored. As the main parts can be reused, it is worthwhile to provide expensive features such as a global clock. One example of such an universally applicable hardware monitor system is our ZM4 system [6–8].

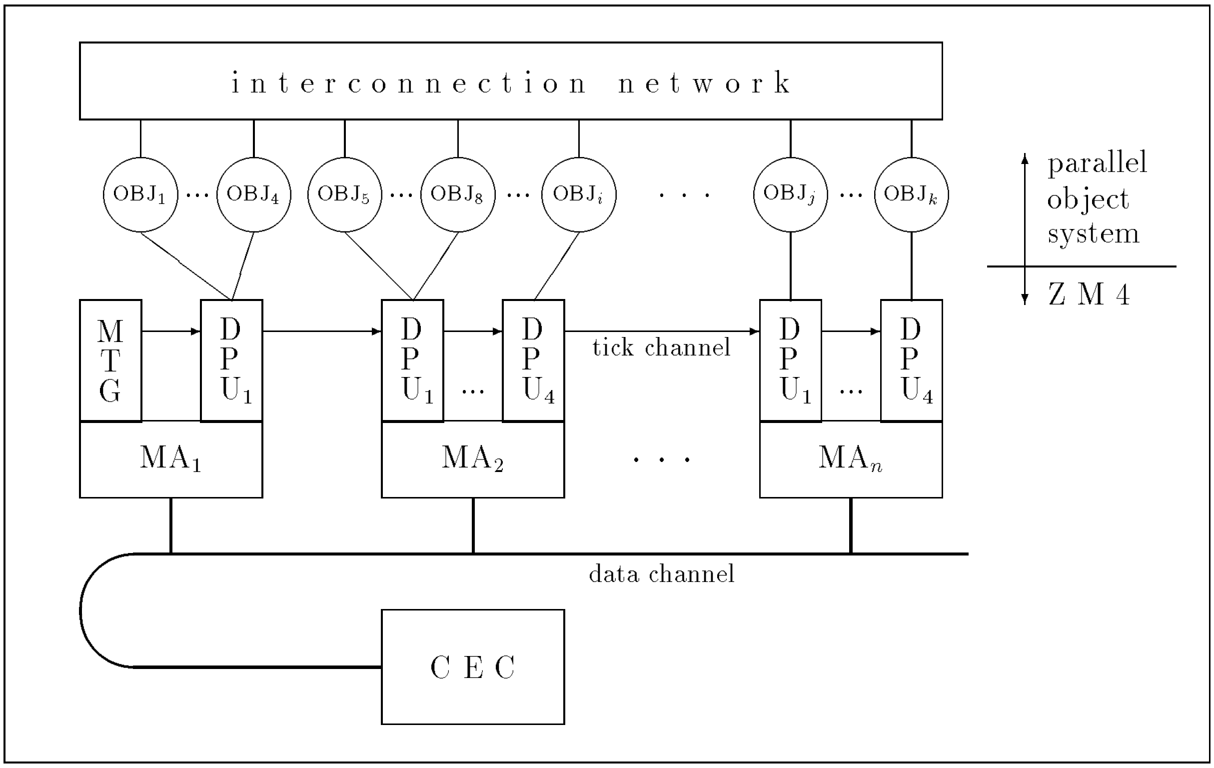


Figure 2. ZM4 Architecture

The ZM4 is structured as a master/slave system with a control and evaluation computer (CEC) as the master, and an arbitrary number of monitor agents (MA) as slaves (see fig. 2). The distance between these MAs can be up to 1,000 meters. Conceptually, the CEC is the host of the whole monitor system. It controls the measurement activities of the MAs, stores the measured data and provides the user with SIMPLE, a powerful and universal toolset for evaluation of the measured data (see section 5).

The MAs are IBM PCs which are equipped with up to 4 dedicated probe units (DPUs). The MAs control the DPUs and buffer the measured event traces on their local disks. The DPUs are printed circuit boards which link the MA to the nodes of the object system. The DPUs are responsible for event recognition, time stamping, event recording and for high-speed buffering of event traces.

A local clock with a resolution of 100 ns and a time stamping mechanism are integrated into the DPU. The clock of each DPU obtains all information for preparing precise and globally valid time stamps from the measure tick generator (MTG) via the tick channel. Time stamps in a physically distributed configuration may be adjusted after the measurement, according to the known wire length. While the tick channel together with the synchronization mechanism is our own development, we used commercially available parts for the data channel, i.e. ETHERNET with TCP/IP. The data channel forms the communication subsystem of ZM4, and it is used to distribute control information and measured data.

The ZM4's architectural flexibility has been achieved by two properties: easy interfacing and a scalable architecture. The DPU can easily be adapted to different object systems. Up to now, interfaces have been built for SUN Sparc, DIRMU, Transputer, IBM PC, SUPRENUM and some embedded systems. ZM4 is fully scalable in terms of MAs and DPUs. The smallest configuration consists of one MA with one DPU, and can monitor up to four object nodes. Larger object systems are matched by more DPUs and MAs respectively.

The idea of a scalable architecture influenced the designs of many other monitor systems. Some important implementations are M^3 [23], Netmon-II [30], Spy [28], TMP [4], TOPSYS [3] and TRAMS [18]. The ZM4 is distinguished from other approaches by the following features:

1. the modular design of interfacing, detection and time stamping has proved to be adaptable to arbitrary object systems with small expense,
2. a global clock mechanism which guarantees high resolution / precise synchronization over large distances,
3. a global clock transmission code which supports detection of synchronization errors in the DPUs.

For a detailed description of the ZM4 hardware and comparisons to other monitor systems, see [8].

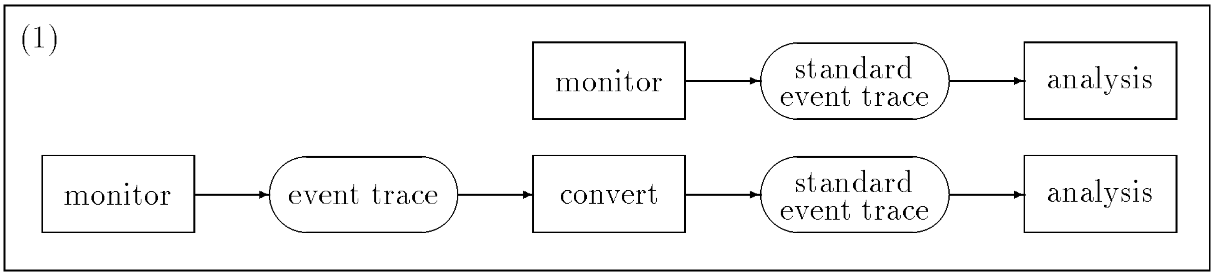
4. Layer 2: Event Trace Access

The event trace access layer acts as an interface between the upper analysis-oriented layers and the monitoring layer. It divides event trace monitoring and analysis systems in two parts: the monitoring part and the analysis part. The connection between these two parts is the monitoring data. Each part is often implemented as a separate tool. Therefore, the implementation of the event trace access layer plays an important role in our discussion of object-independence. In this section we will first discuss possible standardizations of this interface. Then the object-independent trace access interface TDL/POET is described.

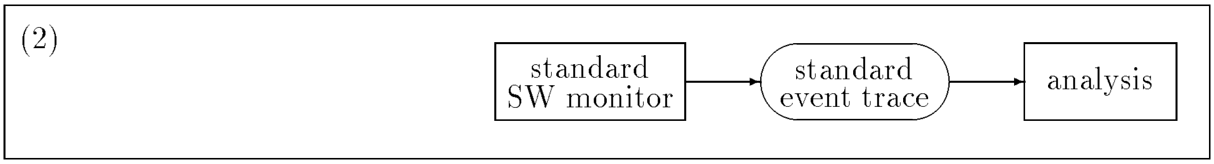
4.1. Possibilities of Standardization

Dozens of monitoring and analysis tools for parallel or distributed computer systems have been implemented, but they are incompatible with each other. Standardization would allow object-independent tools to be developed, and it would also ease the sharing and exchange of traces and of the tools themselves. In the past there were two main attempts to solve this problem: a working group on *Standards in Performance Instrumentation and Visualization* at the 1989 LANL workshop on performance monitoring tools [14], and a BOF session on *Standardizing Trace Formats* at the Supercomputing 1990 conference [25]. Until now it seems that there is no final solution. In the following we will discuss the five possibilities of standardizing the event trace access.

The first two variants (1+2) rely on the definition of a fixed standard record format for event traces. They both have the same major disadvantage: since there is a great variety of existing parallel or distributed computing systems, operating systems and applications, it will be difficult to define a fixed trace format which is general enough to meet all requirements.

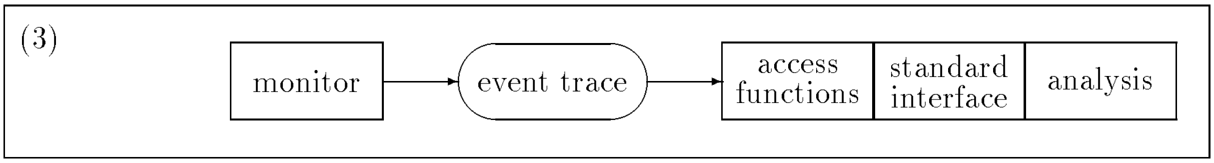


The simplest possibility (1) would be to agree on a fixed standard trace format and to implement or to change all monitoring systems in such a way that they produce this format. If that is not possible, as in the case of an existing hardware monitor, a program to convert the traces into the standard trace format can be used. This approach can only be successful if one restricts oneself to one area of applications, relying on common features defining the fixed trace format. But this would only be a partial solution. One example is the PICL-Format [5] which is very popular but can only be used in message-passing systems. Another approach is the **S**imple **T**race **I**nterchange **F**ormat STIF [9].

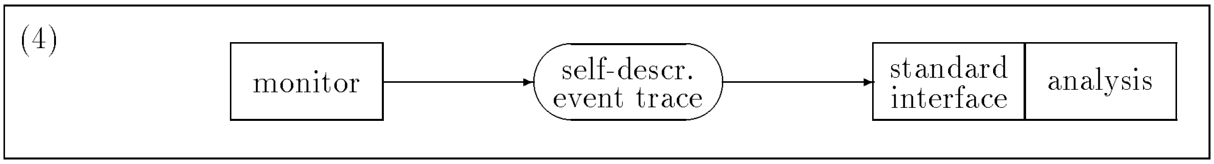


The second approach, which is rather an extension to approach (1), provides a standard software monitor which generates the correct standard trace format. It is possible to implement these functions for different operating systems and programming languages, making instrumented code portable. Nevertheless, the main problem of finding a general trace format remains. An example for this variant is the proposal of the working group at the LANL workshop mentioned above [14].

A fixed trace format is not flexible enough to meet all requirements during the analysis of arbitrary parallel and distributed systems. Another very promising approach is to standardize the interface functions of the analysis system. There are also some variants:



For approach (3), each vendor or monitor system developer implements functions which can read his particular trace format, but presents the data according to a standard interface specification. These functions are linked with standard analysis tools. This is similar to the approach for graphic tools, where different device drivers are used for different output devices. The main disadvantages of this variant are the need for access functions for each trace format one wants to use (this can be many), and that they have to be linked with each analysis tool.

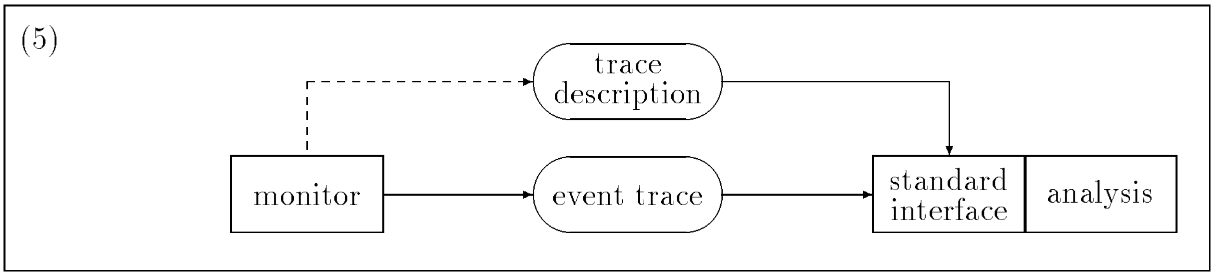


Approach (4) is a combination of standard trace format and standard access functions, however the trace format is not fixed. It is self-describing, i.e. the trace contains information about structure and representation of its data. There are two possibilities: the format description for the whole trace is located in a (standardized) trace header, or each value is prefixed with a so-called *tag*. Examples for a self-describing trace format stored in a header are the Traceview tool [15] and the Pablo environment [26]. Tags are used for coding protocol data units with ASN.1 [10, 11].

By using self-description, this approach is very general and should be able to handle even future trace formats. A small disadvantage arises from existing monitor tools which cannot provide this trace format, but filter tools can be used to convert the trace into the self-describing format.

This last disadvantage is avoided by approach (5). The description of the trace format is not stored in a trace header but in a separate file which is generated by the monitor tool. If necessary a user can also create this description manually and can analyze arbitrary event traces this way. The adaptation to a new trace format consists of generating the corresponding trace description file.

As this approach is the best solution, we use it in our event trace analysis environment SIMPLE. For describing the record format we developed the event **T**race **D**escription



Language TDL. We also developed the **P**roblem-**O**riented **E**vent **T**race interface function library POET, which serves as a standard access interface for arbitrary event traces. Both tools are described in the next subsection.

4.2. The object-independent Trace Access Interface TDL/POET

When designing an object-independent trace access interface, we have to consider the measured traces, as this is what the analysis system sees of the monitored system. All the differences in the object systems, operating systems or applications monitored have an effect on the structure, format, representation and meaning of the measured traces. In order to abstract from these properties we first developed a general logical structure for all the different formats of traces. This logical structure can then be used to define a standardized access method to the event traces.

4.2.1. A General Logical Structure for Event Traces

Using event-driven monitoring, the data resulting from the monitor is a sequence of *event records*, each describing one event. An event record consists of an arbitrary number of components, called *record fields*, each containing a single value describing one aspect of the event. In most cases an event record has record fields containing the event identification and the time the event was recognized. It is also possible that a record field or a group of record fields is not always present in the current event record, or that a record field is interpreted differently, depending on the actual value of another record field. Therefore, it is possible that event records have different lengths even in one event trace.

During the measurement, the event records are stored sequentially in a file (*event trace file*), resulting in a sequence of event records sorted according to increasing time. A section in the event trace which has been continuously recorded is called a *trace segment*. A trace segment describes the dynamic behavior of the monitored system during a time interval in which none of the detected events was lost. The knowledge of segment borders is important, especially for validation tools based on event traces. Usually each trace segment begins with a special data record, the so-called *segment header*, which contains some useful information about the following segment or is simply used to mark the beginning of a new trace segment.

With the hierarchy *event trace* / *trace segment* / *event record* / *record field*, we have a general logical structure which enables us to abstract from the physical structure and representation of the measured event trace (see fig. 3 left). Note that we only specified the structure of an event trace independent from its contents. This does *not* include a specification of event types. We will return to this problem in section 5.

The main differences between different trace formats are the number of the event record fields. Furthermore, an unsegmented trace can be viewed as a trace consisting of one segment without segment header. Therefore, the general logical structure is always the same. An event record with its fields represents an event with its assigned attributes, whereas the event trace file represents the dynamic behavior as a stream of events.

4.2.2. TDL/POET – a Basic Tool for Accessing Event Traces

Based on the logical structure introduced in the last subsection, we designed and implemented the event trace access interface function library POET. The basic idea is to consider the measured event trace a *generic abstract data structure*. The analysis tools can access the traces *only* via a uniform and standardized set of generic procedures.

In order to be able to decode the different trace formats, the POET functions use a trace description file called *key file* (see fig. 3), as discussed in approach (5) above. For efficiency, the key file is in a binary and compact format. It not only describes data formats and representation of the single values, but also includes user-defined (problem-oriented) identifiers for each record field (*field names*) and the interpretations for the values of record fields. In order to access event record fields efficiently, in POET a *field type* is assigned to each record field. These are the following basic field types:

TOKEN: Record fields of type token contain only *one* value from a fixed and well-defined set of constant values. It is a construction similar to the enumeration types in the usual programming languages. They can be used to describe encoded information such as event or processor identifications. Each value has a special fixed meaning called *interpretation*.

FLAGS: Record fields of type flags are like token record fields, but they *can* contain *more than one* value out of a fixed well-defined set. This is done by encoding the individual values as bits which are set or not set. Similar to token values, each bit can have a special meaning, also called interpretation.

TIME: Record fields of type time are used to describe timing information contained in an event record. This timing information can be of arbitrary resolution and mode (point in time or distance from previous time value).

DATA: Record fields of type data contain the value of a variable of the monitored application, or the contents of a register of the object system. They can be compared with variables in programming languages. It is only specified how to interpret their value. This format specification is a simple data type such as integer, unsigned or string.

Additionally, there are other types of event record fields which are only relevant to the decoding system: First, there are *record length fields*, which contain the length of the following record field or of the current or previous event record, and checksums. Second, fields containing irrelevant or uninteresting data, such as blank fields, are called *filler*.

POET provides the following five types of functions for trace initialization, information retrieval and positioning (see fig. 3):

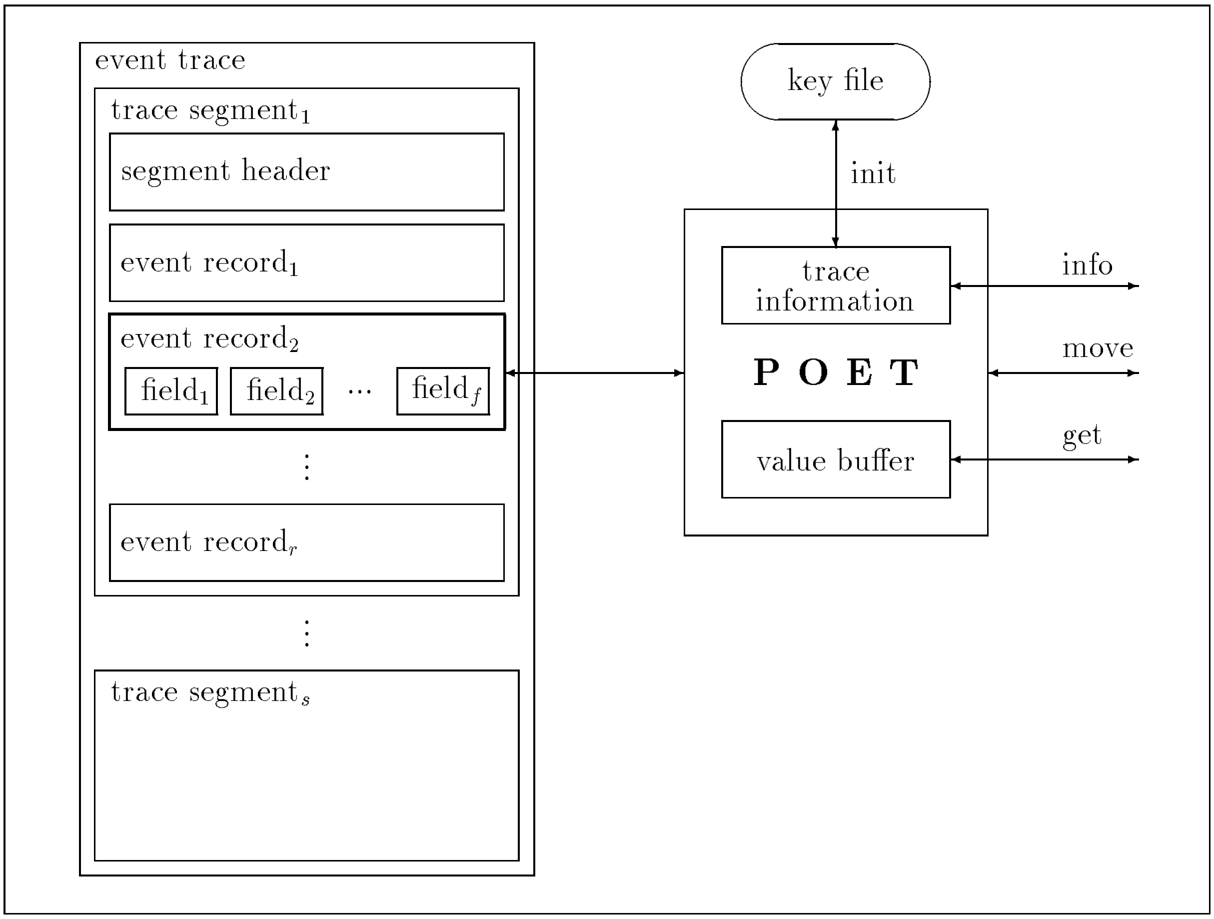


Figure 3. General Event Trace Structure and Structure of POET

init functions: The init functions have to be used to get access to an event trace and to bind the corresponding key file to it. They return a *trace number* which has to be used in the other functions for further reference. In this way, it is possible to have access to more than one trace at the same time. All the information from the key file is read once and stored in internal variables for later use. This is for efficiency and to save file descriptors.

information functions: This group of functions can be used by analysis tools to obtain all useful information about a certain event trace, e.g. the number, the types and the names of record fields, or a list of interpretations defined for a token record field.

move functions: These functions move the current decoding position (shown as highlighted box in fig. 3) in the event trace. With these functions it is possible to process the event records in an event trace in the order they have been recorded (*get_next*), or to move the decoding position in the event trace relative (*forward/backward*) or absolute (*goto_record*) to a desired event record. In moving, the event trace is decoded and the contents of the record fields of the current event record are stored in an internal value buffer.

get functions: For each record field type, POET provides an efficient and representation-independent way of accessing the decoded values of a certain record field (*get_token*, *get_flags*, *get_time*, and *get_value*) stored in the value buffer. But it is also possible to get the decoded values in a generalized form (*get_value*).

special functions: In addition to the routines described above there are some functions to support the user in often needed tasks. This includes routines such as functions for interpreting time values in different resolutions or for error handling.

As an illustration, see the C program fragment in fig. 4. It steps once through an event trace stored in the file *trc_file* and prints for each event record the record number and the problem-oriented interpretation of the event occurred. The program text is not complete; the declarations are missing and there is no error handling (e.g. it should print an error message, when record field "EVENT" does not exist). But it demonstrates how POET can be used to write a program for analyzing event traces.

```
trc_no = init_poet_direct (key_file, trc_file, "");

/*-- get informations about record fields here --*/
/*-- example: get id of token field EVENT and its interpretations --*/
event_id = get_token_id (trc_no, "EVENT");
get_token_interpretations (trc_no, event_id, &event_list);

while ( get_next_segment (trc_no) != NOT_OK ) {
    printf ("+++ NEW SEGMENT +++\n");

    while ( (rec_no = get_next_e_record (trc_no)) != NOT_OK ) {
        /*-- process rec_no-th event record here --*/
        /*-- example: get contents of token field EVENT --*/
        idx = get_token (trc_no, event_id);
        printf ("%d: %s\n", rec_no, event_list[idx]);
    }
}
close_poet (trc_no);
```

Figure 4. Typical Structure of an Analysis Program Using POET

In order to make the construction of the access key file more user-friendly, we developed the event trace description language TDL which is designed for a problem-oriented description of event traces. The TDL compiler checks the TDL description for syntactic and semantic correctness and transforms it into the corresponding binary key file (see fig. 5). In this way, the initialization of POET can be much faster as there is no need for error checking.

The development of TDL had two principal aims: the first was to make a language available which clearly and naturally reflects the fundamental structure of an event trace. The second was that even a user not familiar with all details of the language should be able to read and understand a given TDL description. Therefore, TDL is largely adapted to the English language. The notation of syntactic elements of the language and the general structure of a TDL description are closely related to similar constructs in the programming languages PASCAL and C. By writing an event trace description in TDL, one provides at the same time a documentation of the performed measurement.

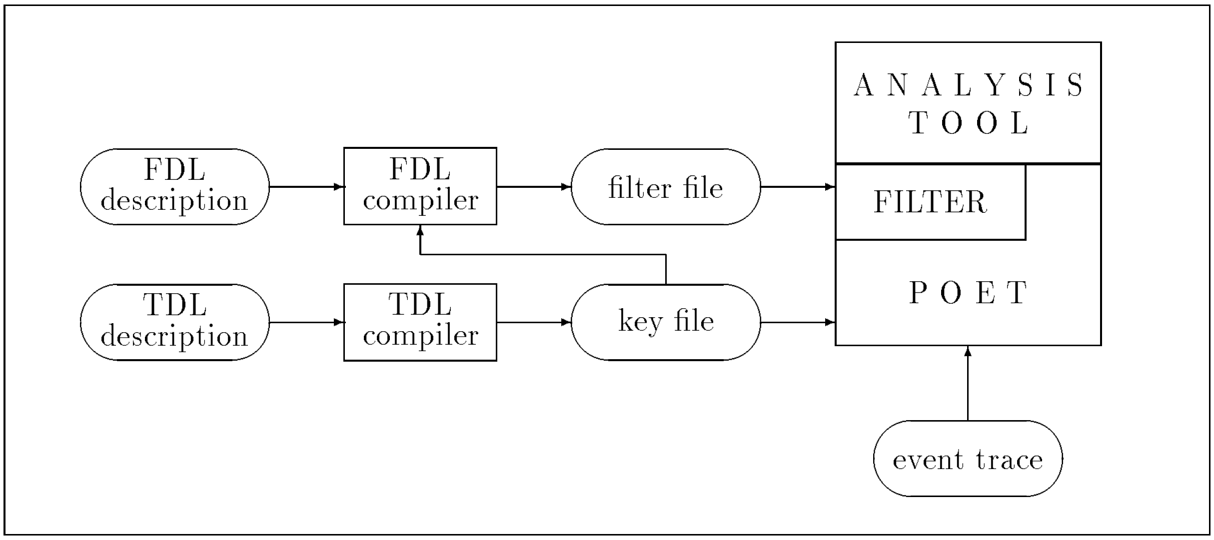


Figure 5. Event Trace Access with TDL/POET

Beyond that, we use a similar approach for filtering event records depending on the values of their record fields. For efficiency, the filtering layer is integrated into POET. There is an additional function to the POET library (*get_next_filtered*) which can be used to move the current decoding position within the event trace to the next event record which matches the user-specified restrictions given in a so-called *filter file*. These rules can be specified in the **F**ilter **D**escription **L**anguage FDL. Since the FDL compiler does not only read the filter description but also the key file, the problem-oriented identifiers for field names and interpretations can be used for specifying the filter rules. The FDL compiler can also check for syntactical correctness and consistency.

A prototype of the tool TDL/POET was designed and implemented under the operating system UNIX in the programming language C in 1987. A redesign took place in 1989. The now available version 5.3 is much faster and provides more functions than the prototype. For details, see the comprehensive user's guide [22]. The tools enable us to analyze event traces which were recorded by ZM4 or other monitor systems such as network analyzers, logic analyzers, software monitors or even traces generated by simulation tools (e.g. QNAP). POET is an open interface. This means that the user can build his own customized analysis tools using the POET function library.

Using a complex data interface like POET does not automatically mean loosing performance. Using UNIX profiling we measured that our analysis tools spend about 5% of their time in POET functions. If the tool uses graphical output (e.g. X windows), this fraction was even less than 1%. Therefore, the win in performance when substituting the POET functions with access routines optimized for a special trace format is small, and the object-independence of the tool is lost.

4.3. Related Work

Configuration files or some sort of data description language are often used in order to make a system independent of the format of its input data. Our work on TDL was inspired by the ISO standard ASN.1 (**A**bstract **S**yntax **N**otation One) [10], which is used in some protocol analyzers to describe the format of the data packets. A similar approach to describe and filter monitoring data was used by Miller et al. in the DPM project (**D**istributed **P**rogram **M**onitor) [17]. Their language allows the description of name, number and size of the components in an event record. The description of trace structures such as segments and of the physical representation of data values is not supported. Its main targets are distributed systems with Send/Receive communication.

In our opinion, the most important work on describing events was the definition of the event trace description language EDL by Bates and Wileden [1]. Their work inspired many others, among them our group. The main purpose of EDL is the definition of complex events out of primitive events. In EDL, attributes of the primitive events can be defined, but not their format or representation [2].

5. Layer 4+5: Analysis Tools

In the last two sections we have shown how to implement the lower three layers of an event trace monitoring and analysis system in an object-independent way. This is possible because these layers need not know any semantics of the data in the monitored traces. Contrary to this, the analysis of trace data needs data semantics. Therefore, it is not possible to implement analysis tools which are *totally* independent of the application or object system to be analyzed. In this section, we will discuss the approach we use in our performance analysis environment SIMPLE [19], which we implemented in such a way that it is as independent as possible.

5.1. The General Approach of SIMPLE

SIMPLE (**S**ource-related and **I**ntegrated **M**ultiprocessor and -computer **P**erformance evaluation, mode**L**ing, and visualization **E**nvironment) is a tool environment designed and implemented for analysis of arbitrarily formatted event traces. It runs on UNIX and with limitations on MS-DOS systems. SIMPLE has a modular structure and standardized interfaces. Therefore it can easily be extended, and tools which were developed and implemented by others can be integrated into SIMPLE with little effort. The object-independence of SIMPLE is based on two principles (see also fig. 6).

First, all analysis tools of SIMPLE use the TDL/POET/FILTER interface for accessing and filtering event traces. This has several advantages: (i) the tools are independent of the trace format and can analyze event traces of arbitrary origin; (ii) all tools are adapted to a new trace format by writing *one* TDL description *once*; (iii) all tools have the same

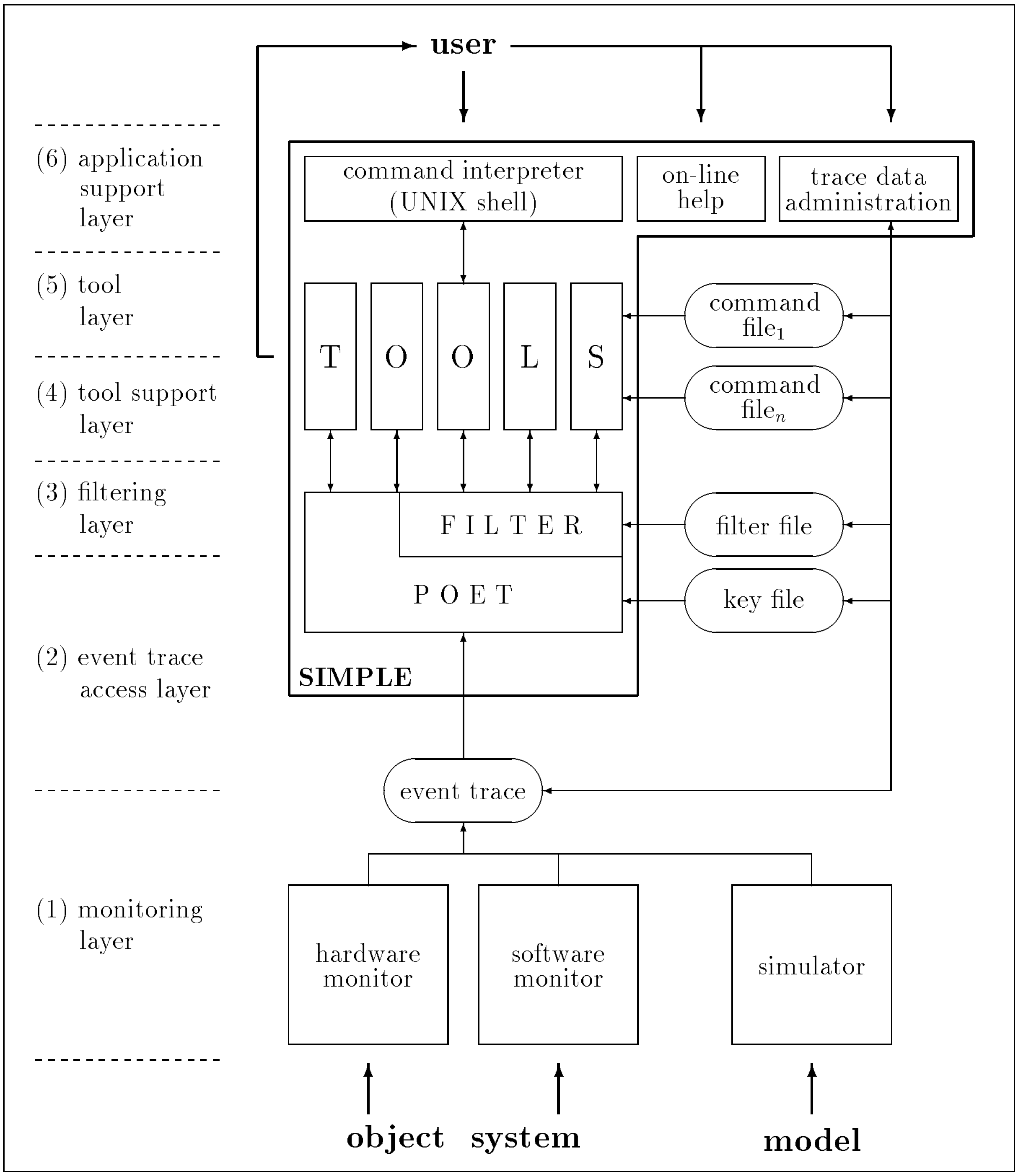


Figure 6. SIMPLE: Overview

user interface concerning filtering event records. Once a filter description is specified, it can be used in all analysis tools; (iv) the tools can use the field names and interpretations, which are defined in the key file, for problem-oriented input and output, e.g. for labeling plots.

Second, the analysis tools are implemented in such a way that they do not rely on the semantics of the trace data, or at least try to avoid this as much as possible. Each tool provides a command and configuration language. The key words of this language represent the features of the tool. If a command contains a record field name, the analysis tool uses the contents of the corresponding record field for the execution of this command. All commands for one analysis task are stored in a *command file*. Once created, it can be used to analyze different event traces.

This procedure has also several advantages: (i) the tools can be adapted very easily to a new object system or application, as they are programmable; (ii) the tools can check whether the specified record field is defined for the trace and whether it has the right type for the current command. For the analysis tool, the specified names are just user-defined identifiers without any semantics, but as the names represent special semantics for the user, he has the impression of specifying problem-oriented and application-dependent commands.

The programmability of the tools has also a small disadvantage: it takes some time to make oneself acquainted with the tools, and the user should already have had some experience. Combined with object-independence however, programmability has another big advantage: it ensures that the tools can be used for realistic and complex applications and that experienced users want to use them also. On the other hand, a monitoring expert can adapt the tools to a new object system or application by simply writing the required description and configuration files. Calling the tools can then be hidden by a menu system, enabling a beginner to use them without difficulty. Later, if the analyses provided are not sufficient, they can be very easily changed or extended. This cannot be done with an object-dependent analysis system – at least not without great effort.

Often, the analysis tools can be implemented independently of the semantics of the trace data and therefore they are object-independent. Unfortunately, this is not always the case. Consider a tool which recognizes complex activities described by regular event expressions as in EDL [2]. It has to know in which record field the event identifications are stored. But many monitoring projects, which have been performed with SIMPLE during the last few years, have shown that in almost all cases only a very small amount of information about the semantics of the trace data is necessary. Not surprisingly, this is the knowledge of the event and the time and place it occurred. If a SIMPLE tool needs this special information, it uses one of the following *predefined standard names* to access the data:

EVENT: This record field contains the event identifications. Therefore, the field type has to be token or flags. The interpretations specified for this record field define the set of possible events.

ACQUISITION: This record field contains the time at which the monitor system has *acquired* [2] the event. This time information is used for ordering the event records according to increasing time.

[2]The time the event occurred cannot be determined exactly, as it takes some time (usually negligible) to recognize an event.

PROCESS/NODE: These record fields indicate the location where the event occurred. If they are declared as token or flags record fields, they also define the set of possible process or node identifications.

Naming a record field with one of the predefined names in the TDL description indicates that this field has the defined semantics. Then the analysis tools can use this information accordingly. Of course, the enumeration of names above is based on our experience only. Someone else may want to extend this list.

It is also possible to standardize some interpretations of the EVENT record field, e.g. *send* and *receive*. This information could be used by a tool which visualizes the communication in computer systems. The only problem is that the semantics of the names must be specified accurately enough to allow the comparing of results obtained through analyses of traces of different origin.

5.2. An Example: Trace Performance Statistics

It is astonishing that a tool for trace performance statistics could be implemented almost independently of the semantics of the trace data, because the word *performance* does not suggest this. The tool only has to know which record field contains the time stamp (for SIMPLE tools this is the time record field named ACQUISITION). In our numerous analyses which we performed with SIMPLE, we found that 95% of the values needed for performance statistics can be assigned to one of the following cases:

1. Determine the *frequency* of certain event records, e.g. for computing the relative frequency or branching probabilities of alternatives in the dynamic program flow.
2. Determine the *distance* in time between each occurrence of a certain event record, e.g. to compute the inter-arrival time between messages.
3. Determine the *duration* in time of an interval defined by a start and an end event record. This can be used to compute the transfer time of messages or the duration of procedures.
4. *Count* the occurrences of certain event records. Here we have to distinguish between event records which increase and decrease the corresponding counter, e.g. computing message queue lengths or the like.
5. *Access* values stored already in a record field such as message lengths.

When a certain performance value could not be computed with these functions, this was usually due to the fact that the events needed for this computation were not (or only partly) recorded. For the specification of the event records which have to be used for the computations, the modules of the filtering layer can be used. This has the additional advantage that the same syntax for expressions can be used for specifying filter rules and performance statistics commands. With the help of the graphics module of the tool support layer, the computed values should be displayed visually, e.g. as scatter plots, bar graphs or box plots.

The user often needs a short summary only. This could be the number of values, the minimum, maximum, and mean value, the sum, and the variance or standard deviation of the values. All these values can be computed within one run through the event trace.

This is a big advantage especially for large event traces. There is even an algorithm which allows an almost exact estimation of the median and other quantiles without storing single values [12].

It would be an useful addition to the command language to not only allow the computation of frequency and time values in total for the whole event trace, but also separately for each value of a particular token record field. This enables the user to display the values e.g. for each process or node, if the appropriate information is stored in the event record.

The ideas presented in this subsection are implemented in the SIMPLE tool trcstat (**TR**a**C**e **STAT**istics). As the main topic of this article is object-independence of trace monitoring and analysis systems, we do not describe other tools of SIMPLE here. The curious reader is referred to [19]. Examples can be found in [20] and in the detailed SIMPLE User's Guide [22].

6. Conclusion

We have already gained some experience using our tools in several different environments. A summary of our main projects is shown in table 1. The type of the object system, its operating system, the application which had to be analyzed and the project partner are listed. Most projects were performed together with other groups in our **I**nstitute for **M**athematical **M**achines and **D**ata processing (IMMD). But there are also some projects with external industrial users: IBM European Networking Center (ENC) in Heidelberg, IBM Research Laboratory in Zurich and Siemens AG in Erlangen and Munich. SIMPLE is now installed at 12 institutions in 5 countries.

Experiences gained during these projects showed that the design principles of ZM4 and SIMPLE are sound. Practical use of ZM4/SIMPLE confirmed that the hardware monitor system ZM4 can easily be adapted to arbitrary object systems and that SIMPLE is a highly flexible and comfortable tool with which all kinds of event traces can be evaluated. Using the TDL/POET interface makes it possible to access event traces of any format and origin by simply giving a TDL description of the trace. The concept of object-independence proved to be a big step forward. Tool environments such as ZM4/SIMPLE provide a valuable aid to designers and users of parallel and distributed systems.

In the title of this article we asked: *Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?* We hope we could convince the reader that the answer to this question is *YES!* Not only is it possible, but such a system can be designed and implemented in an efficient, powerful, *and* object-independent way.

A final word on standardization: we feel that standardization of the physical event trace format is not the right approach. No standard format can be flexible enough to represent all possible event trace formats unless format information is included in the trace – this being somewhat inconvenient. Furthermore, there is a great variety of existing (hardware) monitors which cannot produce a standardized format. Therefore, many conversion programs would have to be implemented. The TDL/POET interface shows that a generalized access method for arbitrary event traces works well without requiring standardized physical formats. We therefore plead for standardizing the event trace access interface instead of standardizing the trace format.

monitor technique	object system [OS]	interface width	output time	application [reference]	project partner
hybrid / direct adaptation	DIRMU multiprocessor [DIRMOS]	parallel interface (front panel) 16 Bit	4 μs	numerical and simulation programs [6]	IMMD
	SUPRENUM [PEACE]	7 segment display 48 Bit	120 μs	ray tracing [27]	IMMD
	IBM-PC 7552 [OS/2] and [MSDOS]	status display 8 Bit	3 μs	communication systems [7, 13]	IBM ENC
hybrid / output adaptation	Transputer T800 network	serial link and INMOS link adapter 8 Bit	220 μs	TRACOS communication system [24]	IMMD
	IBM-PC network [XENIX]	Centronics printer port 8 Bit	320 μs	protocol software	Siemens Erlangen PAP
	IBM-PC network [MSDOS]	Centronics printer port 8 Bit	320 μs	electr. load supervising control system	Fudan Univ. Shanghai
	SUN Sparc network [SunOS]	parallel interface (VME board) 16 Bit	100 μs	X window protocol [29]	IMMD
hybrid / bus adaptation	Siemens robot	SMP bus adapter 8 Bit	16 μs	robot control software	Siemens Munich
	Transputer T800 network	Transputer bus adapter 48 Bit	300 ns	TRACOS communication system [24]	IMMD
	Transputer T425 network	Transputer bus adapter 48 Bit	300 ns	ISO and TCP/IP protocols	IBM Research Zurich
software	IBM R6000 [AIX]	AIX trace facility 176 Bit	??	UNIX device drivers	IBM Research Zurich
	CCC 3280 [XELOS]	system call 32 Bit	13 μs	multiprocessor UNIX	IMMD

Table 1
ZM4/SIMPLE Projects

Acknowledgements
I would like to thank my colleagues Peter Dauphin, Franz Hartleb, Richard Hofmann, Rainer Klar, Andreas Quick and Markus Siegle for the endless discussions, for the patience with me (!), and for all their contribution and interest, without which I could not have achieved without this work. I would also like to thank all the SIMPLE users around the world for adventuring to use my software.

References

1 P.C. Bates and J.C. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach, *Journal of Systems and Software*, 3:255–264, 1983.

2 P. Bates. *Debugging Programs in a Distributed System Environment*, PhD thesis, University of Massachusetts, February 1986.

3 T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitor System of TOPSYS, In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing*, pages 756–764, Zurich, Switzerland, September 1990. Springer, Berlin, LNCS 457.

4 D. Haban. *The Distributed Test Methodology DTM*, PhD thesis, University of Kaiserslautern, FRG, 1988.

5 G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. *PICL: A Portable Instrumented Communication Library*, Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, Tennessee, July 1990.

6 R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, and G. Werner. An Approach to Monitoring and Modeling of Multiprocessor and Multicomputer Systems, In T. Hasegawa et al., editors, *Int. Seminar on Performance of Distributed and Parallel Systems*, pages 91–110, Kyoto, December 1988.

7 R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, A. Quick, and F. Sötz. Integrating Monitoring and Modeling to a Performance Evaluation Methodology, In T. Härder, H. Wedekind, and G. Zimmermann, editors, *Entwurf und Betrieb verteilter Systeme*, pages 122–149, Springer, Berlin, IFB 264, 1990.

8 R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. *Distributed Performance Monitoring: Methods, Tools, and Applications*, TechReport 8/92, Universität Erlangen–Nürnberg, IMMD VII, 1992, submitted to IEEE Transactions on Parallel and Distributed Systems.

9 R.W. Hon. *A Simple Trace Interchange Format*, unpublished document available for anonymous ftp from `eagle.cnsf.cornell.edu` in `pub/BOF` as `apple.txt.Z`, 1990.

10 *International Standard 8825, Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, ISO, 1986.

11 *International Standard 8825, Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, ISO, 1986.

12 R. Jain and I. Chlamtac. The P2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations, *Comm. of the ACM*, 28(10), 1985.

13 N. Luttenberger and R.v. Stieglitz. Performance Evaluation of a Communication Subsystem Prototype for Broadband–ISDN, In *Proceedings of the 2nd Workshop on Future Trends of Distributed Computing Systems in the 1990's*, Kairo, 1990.

14 A.D. Malony and K. Nichols. Standards in Performance Instrumentation and Visualization for Parallel Computer Systems, In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, chapter 17, pages 261–278. ACM Press, Frontier Series, Addison–Wesley Publishing Company, New York, 1990.

15 A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski. Traceview: A Trace Visualization Tool, *IEEE Software*, September 1991.

16 C.E. McDowell and D.P. Helmbold. Debugging Concurrent Programs, *ACM Computing Surveys*, 21(4):593–622, December 1989.
17 B.P. Miller, C. Macrander, and S. Sechrest. A Distributed Programs Monitor for Berkeley UNIX, *Software – Practice and Experience*, 16(2):183–200, February 1986.
18 A. Mink, R. Carpenter, G. Nacht, and J. Roberts. Multiprocessor Performance–Measurement Instrumentation, *Computer*, pages 63–75, September 1990.
19 B. Mohr. Performance Evaluation of Parallel Programs in Parallel and Distributed Systems, In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing*, pages 176–187, Zurich, Switzerland, September 1990, Springer, Berlin, LNCS 457.
20 B. Mohr. SIMPLE: a Performance Evaluation Tool Environment for Parallel and Distributed Systems, In A. Bode, editor, *Distributed Memory Computing, 2nd European Conference, EDMCC2*, pages 80–89, Munich, April 1991. Springer, Berlin, LNCS 487.
21 B. Mohr. *Event Trace Monitoring and Analysis Systems for Evaluation of Parallel and Distributed Systems* (in German), PhD thesis, Universität Erlangen–Nürnberg, VDI Verlag, Fortschritt-Berichte, Reihe 10, Oktober 1992.
22 B. Mohr. *SIMPLE – User's Guide Version 5.3.*
Part A: TDL Reference Guide
Part B: POET Reference Manual
Part C: Tools Reference Manual
Part D: FDL / VARUS Reference Guide,
TechReport 3/92, 270 pages, Universität Erlangen–Nürnberg, IMMD VII, 1992.
23 M. Moser. The ELAN Performance Analysis Environment, In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing*, pages 188–199, Zurich, September 1990. Springer, Berlin, LNCS 457.
24 C.-W. Oehlrich and A. Quick. Performance Evaluation of a Communication System for Transputer–Networks Based on Monitored Event Traces, *ACM SIGARCH*, 19(3):202–211, 18th Int. Symp. on Computer Architecture, Toronto, May 1991.
25 C. Pancake, D. Gannon, S. Utter, and D. Bergmark. *Supercomputing '90 BOF session on Standardizing Parallel Trace Formats*, unpublished document available in PostScript form for anonymous ftp from `eagle.cnsf.cornell.edu` in `pub/BOF` as `bof.ps`, November 1990.
26 D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, B.W. Schwartz. *The Pablo Performance Analysis Environment*, CS Dept., University of Illinois, Draft Version, March 1992.
27 M. Siegle and R. Hofmann. Monitoring Program Behaviour on SUPRENUM, *ACM SIGARCH*, 20(2):332–341, 19th Int. Symp. on Comp. Arch., Queensland, May 1992.
28 M. Keller and H. Rüscher. *SPY Core Users Manual*, Release 1.2, Asea Brown Boveri Process Automation AG, Switzerland, 1992.
29 N. Wang. *An Experimental Environment for a Performance Study of X Window Systems*, TechReport 1/92, Universität Erlangen–Nürnberg, IMMD VII, 1992.
30 M. Zitterbart. Monitoring and Debugging Transputer-Networks with NETMON-II, In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing*, pages 200–209, Zurich, Switzerland, September 1990. Springer, Berlin, LNCS 457.