Event Handling in Prolog

Micha Meier

ECRC-ECRC-95-09

technical report ECRC-ECRC-95-09

Event Handling in Prolog

Micha Meier

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de

Abstract

This paper puts forward the argument for a general and flexible event handling mechanism in Prolog. This will make it more user-friendly, more powerful and more versatile for use in various real-life application domains. We present a scheme of handling synchronous and asynchronous events in Prolog, justify why such a scheme should be incorporated in Prolog, present its advantages and describe how it can be implemented. The major part of the presented scheme has been implemented and verified in the ECRC SEPIA system [8].

1 Introduction

An *event* is a special condition which can arise during the normal execution of a program and reports that a nonstandard situation has occurred. It can be caused either by the program itself (for example when the program accesses data of incorrect type), or by some external activity which is independent on the program, like a signal from the user or from another process.

The most common example of an event is an *error*. An error is usually signaled by the system to warn the user that the program could not be executed for some reason. Errors normally occur only in built-in predicates because user predicates are defined using clauses and literals and so any failure or message occurs there explicitly.

Most of the current Prolog systems have either no ability to handle events at all or only a restricted one, which is probably caused by the features that distinguish Prolog from conventional languages: Prolog is a declarative language based on the resolution principle, the execution of a Prolog procedure is a sequence of logical inferences, its results can be described in terms of success, substitution and failure. These features seem to make any sort of event handling obsolete, since every exceptional situation can be handled by failure.

But there are also good reasons to have event handling in Prolog:

- If a Prolog program has to respond to external events, it has to provide some event handling. Similarly, if a program acts as a metaprogram that controls the execution of another program, it must be able to handle events in the object program.
- Application programs which have their own user interface also need to maintain control even when exceptional situations occur. Without a dedicated error handling concept such programs have to check necessary preconditions for each built-in predicate, otherwise the program might abort unexpectedly.

  With an event handling scheme exceptional situations are handled by appropriate event handlers and so the normal execution is cleanly separated from exceptional situations.
- In a Prolog system that signals every unusual situation, e.g. a wrong argument type or a call of an undefined procedure, it is very easy to detect trivial misspellings and similar errors, but in a system that simply fails it is much more tedious to find the reason why a program failed.

- When a built-in predicate is called with an argument of an incorrect type, for example `?- X is 1 + a`, it can simply fail, however when the arguments are not sufficiently instantiated, like in `?- X is 1 + Y`, a correct action would be to generate all possible substitutions that satisfy this goal. In the case of arithmetic this has no practical sense and this is why some Prolog systems handle such calls with failure. Unfortunately the logical basis and completeness is then lost.

- Treating some exceptional cases using failure may seem very strange taken into account the practice common in other programming languages, for instance division by zero is traditionally treated as an execution error.

- In some cases is it doubtful that an exceptional situation should be treated by failure. For example, arithmetic overflow is at most a failure of the particular Prolog implementation to represent the result of an arithmetic operation (more on this topic see in [9]).

- Prolog is interactive, the user can type goals directly to the interpreting loop. If the goal is a built-in predicate and it just fails, it might be difficult for the user to find out the reason for the failure. A classical example is a syntax error due to incorrect input to the predicate `read/1`. No Prolog system just fails if there is a syntax error in the user input, although there are some that silently fail or even succeed when the syntax error occurs in a consulted file!

- All exceptional situations should be handled in a uniform way. Since (to the knowledge of the author) no Prolog system that would handle all possible errors just by failure exists, it is clearly more consistent to handle **all** errors as events using an appropriate event handling scheme. As there may be different requirements for the actual event processing, the event concept should be flexible enough to allow redefinition of the default handling.

We have listed some reasons for the introduction of event handling in Prolog systems. Another question is, how to handle an error when it occurs. The standard action is to print a message and then abort execution or switch on the debugger. What can the user do to change this default behavior, or even to correct the error condition? This, together with handling of other event types, is discussed in the rest of this paper. We first present a short classification of event types, then describe how they should be handled by the Prolog system and discuss the implementation of the presented design.

2 Event Types

There are two basic event types:

- **Synchronous events**, called **exceptions**. They occur as a result of the program's activity, they are directly related to the operation that the program is executing. We classify as an exception a situation that either cannot be handled by the program correctly, or it could be handled in several possible ways. Examples include arithmetic overflow, calling an undefined procedure, reading past the end of file, accessing a nonexistent file etc. The term *exceptions* therefore includes errors but also other events that might be handled in several different ways, depending on the context and maybe on personal taste. Introducing exceptions into Prolog contributes to its flexibility because the processing of an exception can be defined by the user.

  Besides exceptions there are other possible synchronous event types, not necessarily available on all Prolog systems, e.g. resuming a suspended goal in systems which support delayed goals.

- **Asynchronous events**, called **interrupts**. They result from some external activity, they are triggered by some action outside the program and possibly even outside the computer. For example, when the user types Ctrl-C, an interrupt is sent to the attached process no matter what it is currently executing.

Apart from implementation difference between synchronous and asynchronous events, there is also a conceptual difference between the two: a synchronous event results from the activity of the program and so it can be viewed as a logical consequence of the program. Asynchronous events are triggered by an external activity and so they are logically independent on the running program.

Sometimes an event is caused by the program itself but is signaled asynchronously, for example some incorrect arithmetic operations are caught by the hardware and an interrupt is sent to the program, another example is when the program is asking for some resource and it cannot obtain it from the operating system. These events will not be handled in this paper, because they are machine dependent and also because normally it is possible to avoid them by doing appropriate tests.

3 Event Handling

The goals of a generalized event handling system are:

- events are processed by *event handlers*, which are arbitrary procedures definable by the user,
- interrupts are handled immediately, in an asynchronous way, so that a fast response is guaranteed,
- event handling is efficient and flexible.

When an event occurs, the appropriate event handler is invoked. Due to the difference between the two main event types, there are differences in how the event handler influences the main execution:

- For **synchronous events**, the event handler *replaces* the goal which has initiated the event. This is logically equivalent to renaming the predicate in which the error occurred and adding a clause

  ```
  beylinespred(A, ...) :-        (error_condition(A, ...,
  ErrorId) -> error_handler(ErrorId, ...)
  ;           original_pred(A, ...)        ).
  ```

  If an error condition occurs, the predicate call succeeds or fails depending on the success or failure of the event handler. This means that after a synchronous event the execution never returns to the predicate that has caused the event. Apart from logical clarity, this approach has also advantages in implementation.

- **Asynchronous events** are independent on the application program and therefore the execution of the event handler is transparent to it. Unless the event handler aborts or has other side effects, the main execution is not influenced by it. There is no logical connection between application execution and the event handler execution and so they run as two independent programs.

  If an interaction between the interrupt handler and the interrupted program is required, e.g. setting a semaphore, the event handler can of course do anything a normal procedure can do, like assert a clause, call an external predicate which sets a flag etc.

3.1 Handler Invocation

In our scheme a separate handler is associated with each event. When the event occurs, the appropriate handler is invoked. To be able to interact with the executing program, the handler needs some arguments which must be provided by the system on the handler invocation:

- The event identification. It is necessary for the case that one handler is used to handle several event types.
- The goal in which the event occurred. This argument is passed only to synchronous events, because interrupts have no relation to the goal which has been interrupted.

  It can be used either to make a selective handling of the same event for different predicates, e.g. the event `type_error` may fail for the predicate `functor/3` and abort for all others. The second purpose of the goal argument is to be able to access this goal, to examine or print its arguments or to re-call it when the reason for the event has been fixed in the handler.
- An optional third argument for exceptions is the handler context in which the error occurred, to be used in the `propagate_event/3` predicate, see section 3.2. It is a list of active local handlers.

The event invocation is of course different for synchronous and asynchronous events: the synchronous events cause abandoning of the predicate where they occurred. Then the execution continues normally to the event handler thus replacing the abandoned goal by the event handler call. The asynchronous events cause the current predicate to be interrupted in whatever state the execution was, and the handler is immediately invoked. This requires that the system saves all the data that could be overwritten by the handler execution and when the handler finishes, the interrupted predicate can be continued as if nothing had happened.

3.2 Handler Specification

There are two types of events handlers, *local* and *global*. A local handler is valid only during the execution of a specified goal, whereas global handlers are always active and they are invoked if no local handler has been set.

The primitives to set the event handlers are as follows:

- The global handler is set using the predicate

set_event_handler(EventId, Proc)

which has the effect that now each occurrence of the event **EventId** will invoke the handler **Proc** unless a local handler has been set. This handler setting is *global* since it concerns all events of this type no matter in which predicate they occur, and *type-specific* since other events are not influenced by it.

- A dual concept is the predicate

  trap(Goal, Handler)

  which calls the **Goal** and all events that occur during its execution and the execution of its children (except for new **trap/2** calls) are processed by the handler **Handler**. This handler setting is *local* since it concerns only events that occur during the execution of one predicate, and *general* since any event is processed by the specified handler. A handler specified by **trap/2** is valid until **trap/2** exits or fails or a new **trap/2** is called.

Our event handling concept is based on these two predicates which give the user the ability to perform almost any desired action in specific cases. The predicate **set_event_handler/2** can be used to change the behavior of the system whenever the specified event occurs, moreover the action can be restricted only to specified predicates using the goal argument of the handlers. The predicate **trap/2** is used to trap all events that occur in the specified predicate call, which is necessary e.g. when the program wants to retain control even if an error occurs or when it has to undo some side effects of the predicate before it is aborted. The two predicates can be combined, which is useful e.g. when only some of all possible events should be handled locally, for others the default handler is to be taken.

The handler invoked by **trap/2** may decide not to handle the exception by itself and propagate it instead to the parent's handler, which can be either a handler specified by an ancestor **trap/2** call or the global handler. The predicate **propagate_event(EventId, Goal, Context)** invokes the previous handler specified by the Context, which is an optional third argument of the events handlers. If, for instance, any type or range errors that occur in the predicate **p/1** should cause failure, and all other events are to be handled by the parent handlers, the program can call **?- trap(p(X), handler/3)** and the handler is defined as

```
beylineshandler(EventId, Goal, Context) :-          EventId
\= type_error,          EventId \= range_error,
propagate_event(EventId, Goal, Context).
```

The predicate **propagate_event/3** issues the specified event and uses the handler specified by Context to handle it. Thus the event is propagated to the parent handler if it is different from the two specified errors, otherwise the handler fails and so does the call to **p/1**.

3.3 Handler Execution

Once the event handler has been invoked, it is executed like a normal Prolog procedure. As we mentioned above, the processing of interrupts is transparent to the normal execution which cannot be influenced by it, except if the interrupt handler aborts or has other side effects. This is also a consequence of the fact that an interrupt handler can have only one argument which is the event identification and so it cannot access any variables from the interrupted predicate. An interrupt handler can of course be interrupted by another interrupt handler, provided that the interrupts were not disabled using the predicate **disable_interrupts/0**.

The handler of a synchronous event has many possibilities to influence the normal execution, it can fail, abort, call any other predicate including the one in which the event occurred, it can bind its variables, create new compound terms on the global stack and it can also be nondeterministic. Nondeterministic handlers are used when a nondeterministic predicate issues an error and the error handler tries to correct the error and re-call the predicate.

Although the event handler can always succeed, the system might not always allow it, for example when in a modular system the program tries to access some private data, there might be a predicate that checks the permission and causes an event if the user has none. Then, by a simple redefinition of the handler the program could bypass this security check. Such cases can be handled locally, for example the success of the event handler can be ignored by forcing a failure:

```
beylinesaccess_check(Data) :-        (user_access(Data) ->
true        ;                event(no_access, access_check(Data)),
fail     % force failure if the handler succeeds          ).
```

If the handler writes some message describing the event to notify the user, this message should be directed to a dedicated stream so that all handler messages can be isolated and also suppressed if needed.

3.4 Comparison with Other Schemes

Since the recent ISO proposal [4] provides error handling, we would like to compare it with our scheme, not without hope that the proposed standard could take at least some of our ideas on the subject into account. The key concept in the ISO error handling are the blocks. The predicates **block/3** and **exit_block/1**, which are the Prolog counterparts of LISP's *catch & throw* mechanism, provide the possibility to abandon the execution of a predicate before it actually succeeds or fails, and to call another procedure instead. The predicate **block(Goal, Tag, Recovery)** calls the *Goal* and succeeds or fails if

Goal succeeds or fails respectively. If during the execution of *Goal* **exit_block(Tag)** is called such that its *Tag* unifies with a **block/3** tag, all active predicates up to this **block/3** call are abandoned, their frames popped, their bindings undone and finally the *Recovery* procedure is called. This predicate is also used in our scheme, but not directly for event handling but rather to abort the execution in an event handler by executing **exit_block(abort)**.

The proposed way of using this mechanism in ISO is as follows:

```
beylinesmain :-     repeat,     block(run, Fault,
recover(Fault)),    fail.
```

and it is assured that an error issues **exit_block(ErrorId)**. Apart from this, ISO proposes a global error handler which can be set with

```
write_prolog_flag(error, Handler)
```

and then Handler (a callable term) is called in the case of an error. The basic features are similar to our scheme, namely one global and one local handling mechanism, but besides this there are some conceptual differences:

1. The ISO proposal treats only errors, not interrupts, and the number of different error types is low. Our scheme supposes that the number of possible event types can be large.

2. The local handling scheme, using **block/3**, requires that the error executes an **exit_block/1**. Then, however, it is not possible to restart the erroneous goal since its ancestors are popped during the block exit and their bindings undone. To prevent this, it is necessary to place a block around any call that may issue an error which is a rather awkward method. In our scheme the **trap/2** predicate defines a handler which replaces only the erroneous goal.

   The **block/3** predicate cuts all the alternatives of the called *Goal* and so this mechanism cannot be used to restart nondeterministic predicates.

3. The ISO handlers do not know in which predicate the error occurred. Consequently, the goal cannot be corrected and restarted in the handler, nor can the error be sufficiently described in a warning message, and so the global error handling is overly restrictive. If the erroneous goal is passed as argument of **exit_block/1**, i.e. the *Tag* is a compound term, a complicated mechanism is needed to implement the block exit.

ISO and some other Prolog systems offer error handling schemes where *one* defined procedure is called whenever *any* error occurs. Our scheme provides

both the possibility to call a specific global handler for each event type and to call a local event handler common for all errors and so it is more general. In addition, it is necessary to take into account the following:

- In our scheme, if the Prolog session is started with

  `trap(toplevel_loop, handler/3)`,

  there is only one handler which is invoked for all events like in some other systems. This has the advantage that it is possible to define event handling by asserting new clauses of `handler/3`. For example, if we wish to treat all events that occur in `functor/3` by failure, we can achieve it by

  ```
  ?- asserta((handler(_, functor(_, _, _), _) :- !, fail)).
  ```

- This approach slows down the selection of the correct handler clause: the procedures modifyable with `assert/1` are usually interpreted even in compiler-based systems. Moreover, they are either not indexed at all, or only on one argument and so the time required to select the matching clause may become proportional to the number of clauses in the handler.

Event handling is related to writing logic operating systems [3, 12, 5]. For example, the exception handling as described in [5] can be implemented using our primitives. The difference is that [5] makes a clear distinction between metaprograms and object programs, whereas our scheme, being at a lower conceptional level, does not make this explicit difference, it is not required (however possible) that the event handlers are metaprograms.

4 Implementation

In this section we describe some important details that allow to implement the event handling concept in a WAM [11] based system.

4.1 Accessing the Handlers

The access is direct, similar to *vectored interrupts* on some CPU's: the system maintains a table of event handlers which contains the address of the corresponding handler for each possible event. Consequently, the invocation is very fast, because each event is identified by an identification number which corresponds to the offset in the handler table where the address of its event handler is stored. Since the interrupts have to be processed differently to other events and their number is constant (usually it is hardware-dependent, e.g. the number of signals), the best approach is to assign the lowest offsets to the interrupts and the higher ones to synchronous events. Then it is fairly easy to add new, possibly user-definable synchronous events and also to distinguish error numbers from the interrupt ones. The entries in the table are updated with the predicate **set_event_handler/2**.

In addition to the handlers above, there is one local handler which is the most recent one set by the **trap/2** predicate call. Since the calls to **trap/2** can be nested, the previous handlers must be remembered in a list. Moreover, if the trapped goal is nondeterministic, a previous handler may become active again when the system backtracks into the trapped goal. The local handlers can be easily managed at the Prolog level using two built-in predicates **set_local_handler/1** and **get_local_handler/1** that return or set the local handler list:

```
beylinestrap(Goal, Handler) :-
get_local_handler(OldHandlerList),
set_local_handler([Handler|OldHandlerList]),        (
call(Goal),                trap_reset(Handler, OldHandlerList)
;                set_local_handler(OldHandlerList)          ).
trap_reset(_, OldHandlerList) :-
set_local_handler(OldHandlerList). trap_reset(Handler,
OldHandlerList) :-
set_local_handler([Handler|OldHandlerList]),        fail.
```

The auxiliary procedure **trap_reset/2** sets the previous handler whenever the trapped goal succeeds. If the execution later fails and it backtracks into the

body of `trap/2`, it adds the correct handler at the beginning of the list and propagates the failure to the called goal. When the trapped goal fails, the previous handler is reset in the body of `trap/2`.

4.2 Invocation

When an event occurs, the system has to carry out the following actions:

- It must find the appropriate event handler. If a local handler exists, it is selected, otherwise the handler is selected from the system handler table.
- It must save all data that could be overwritten by the execution of the handler.
- The handler arguments have to be fetched. The event identification is supplied by the event itself, the culprit goal must be provided by the system. If the predicate in which the event occurred is written in Prolog, the event can be invoked directly using a predicate `event(EventId, Goal)` and the Goal is supplied literally. If the predicate is written in the implementation language, its arguments are available in the argument registers and so only the functor must be provided, and this can be done through a pointer from the WAM code.

The event identification is a number, an offset in the system event handler table. At the Prolog level, the event identification is an atom which is a shorthand for the event name, e.g. `overflow` or `type_error`. This makes it possible to use events easily at the Prolog level and also to add new events without renumbering the old ones or having similar events with totally different numbers.

4.3 Interrupts

Interrupts have to be processed asynchronously so that an immediate response is guaranteed, otherwise they could not be used for real-life applications. Some Prolog systems provide synchronous interrupt handling, by setting a flag when the user presses the break key. Then, in some well-defined check points the flag is tested and if set, the break handler is invoked. There are several reasons why such polling is not sufficient:

- If the flag is tested too often, the execution without interrupts is slowed down. If it is tested less frequently, e.g. on each procedure call, the response may be too slow, because a complex unification or shallow backtracking over several clauses of the called predicate may cause long pauses between two procedure calls.

- Most of the current systems allow connection of external procedures to the Prolog system. Such procedures are e.g. written in C and while they are executed, no interrupts are processed. The external procedure can in fact be an entire application program of which the Prolog system is only a small part.

- When the system is waiting for some user action, e.g. input from the keyboard, no interrupts can be processed because the Prolog execution is blocked. This is particularly painful when a window interface is connected to Prolog. Since the mouse is driven using interrupts, Prolog must execute a dummy loop while waiting for a mouse event and the user cannot enter input from the keyboard.

Asynchronous interrupt handling requires in fact only three basic features to be present in the Prolog system, however they have many further implications so that the whole system must be designed properly with respect to the interrupt handling:

1. At any time the system must be able to interrupt the current execution, save enough data to resume it afterwards, and invoke the interrupt handling procedure. This implies that the system knows which data must be saved and that it does not save too much of, otherwise the reponse time would increase.

2. Global storage areas can be used only to store data that is globally valid. Such data include e.g. the symbol table, code area, global flags etc. When this data is modified, the asynchronous events must be *delayed*, otherwise the event handler could be executed in an inconsistent state.

3. Important data must be stored in a way that prevents the interrupt handlers from overwriting it. For example, no important data may be stored above the stack top and it must be possible to find the stack top easily.

The first point concerns particularly the WAM. The Warren Machine uses an indefinite number of argument and temporary registers (they are actually two names for one abstract register set). When an interrupt occurs, it is usually not known how many abstract registers store meaningful information.

The solution we suggest and which we have implemented, is to physically separate the argument and temporary registers and handle them differently. The abstract registers which are assigned to hardware registers have to be saved on each interrupt, this is often done by the operating system. The remaining abstract temporary registers are allocated on the stack, and therefore they need not be saved on interrupts; this is also the traditional way to cope with similar problems.

The argument registers could be handled similarly, i.e. they can be located on the stack instead of using some fixed memory area. As this method requires rather severe changes in the WAM and its compiler, a simpler one can be used, namely invalidating the first unused argument register by storing a special value into it at some defined execution points, e.g. when creating a choice point. If all arguments are initialized with the invalid value, it is guaranteed that when all arguments up to the first invalid one are saved on an interrupt, no data will be lost. Prolog procedures have usually a small arity [7] which justifies such an approach[1].

Another problem in the WAM concerns the management of the local stack: there is no dedicated stack top pointer, the stack top is always computed dynamically from the current frame pointer and the current size of the topmost frame which is accessed as a constant in the code area. This approach makes it possible to dynamically change the size of the topmost frame (*environment trimming*), but it makes it difficult to interrupt the current execution and to use the local stack to execute the interrupt handler, since the size of the topmost frame is not known.

Fortunately, introducing a local stack top pointer solves the whole problem. This implies that the **allocate** instruction will have one parameter which is the environment size. The **call** instruction as well as **allocate** and **fail** set the stack top pointer appropriately. When an interrupt occurs, the stack top is known and so the local stack can be immediately used.

Before processing the interrupt some data must be saved by the system. It has already been argued that splitting the local stack has many advantages [6], and for interrupt handling this is also extremely convenient. The local stack is split into the *environment stack*, which contains only the environments, and the *control stack*, which contains choice points and other frames, particularly the interrupt frames.

4.4 Exceptions

Some Prolog systems expand certain built-in predicates in-line or they use some compiler optimizations that take into account the properties of the built-in predicates. If an exception occurs in such a predicate, the system may have to behave much like after an interrupt. It has to save all argument registers and flags which are supposed to survive the call to the built-in predicate. Moreover, if the compiler uses the fact that the built-in predicate is deterministic, the error handler (which effectively replaces the built-in in case of an event) must not be nondeterministic and so any choice points it may have left must be removed.

[1] Of course in the worst case the system saves a number of registers which is the highest arity present in the whole program, however this is balanced by a very fast execution in the average case because no additional instructions are executed.

4.5 Garbage Collection

In a system that supports both asynchronous event handling and garbage collection it is necessary to design carefully the interaction between the two.

The first question is whether the garbage collection can be interrupted by an asynchronous event and if so, how is the handler executed. During the garbage collection the entire collected memory area is not in a consistent state and it must not be accessed. However, since our interrupt handlers do not interact with the interrupted predicates and they do not access the stacks being collected, they can be executed without any problem.

The other question, namely whether an interrupt handler can start the garbage collection, is more complex. There are some points to be mentioned:

- The response to an interrupt must be immediate and this makes the execution of the garbage collector inside the interrupt handler highly undesirable.
- Since after an interrupt all hardware registers are saved and generally more argument registers can be saved than necessary, the garbage collector may have to follow some saved pointers that are no longer valid which will reduce the amount of collected garbage.
- Up to the first **call** instruction, the environment of a clause may contain uninitialized variables. If an interrupt comes, this may also confuse the garbage collector.
- The current incremental garbage collection schemes [2, 10] enable the garbage collection to be executed frequently enough to make sure that there will be always a certain amount of space available. This means that in the normal case the interrupt handler should not cause memory overflow, unless it requires more space than available even after garbage collection.

Therefore, if the heap overflows during the execution of an interrupt handler, the garbage collector should mark and collect only the area that has been used in the interrupt handler itself. This is possible since there are no references to or from the space used by the normal execution. If there is not enough garbage to collect, the top part of the heap can be expanded by allocating enough space, copying the data and updating pointers. The space occupied by the execution of the interrupt handler is likely to be much smaller than the whole occupied space and so the overhead may be acceptable.

5 Summary

In this paper we have described the design of an event handling scheme in Prolog. Since it differs from schemes available in other Prolog systems in its generality, we want to justify that there is a need for such a general scheme. First we want to mention that the implementation of the whole design is not very difficult, most of it has been implemented in the SEPIA system [8] and has proven to be powerful and efficient enough for real life applications.

5.1 Exceptions

In our scheme, the user can define the handler for any error which can occur in the system. Sometimes it is required that the errors output a warning message, sometimes they should abort, sometimes only silently fail but they can also request user interaction and continue. All this can be very easily achieved, just by defining an appropriate handler for the required error. Our scheme allows to define global handlers which are type-specific and local handlers which apply to all event types, as well as a combination of the two.

There is also a number of situations in which it is difficult to state how the system should behave. There may be several possibilities, each of them applicable to a particular situation. An example is a call to an undefined predicate. From the logical point of view, the call cannot be resolved and so it simply fails. For a programmer it is helpful if a message is printed since probably some predicate name was misspelled. For a system developer the desired action may be to look for a file where this predicate is defined, load it and continue the execution.

In such cases, our design has the ability to define exception handlers for each of the above actions. For example, calling

```
set_event_handler(undef_call, fail/0)
```

will cause the undefined predicate to fail,

```
set_event_handler(undef_call, error_handler/2)
```

will print a warning about the undefined call and abort. A procedure that is able to find the definition of a specified predicate could be written as

```
beylinesundef_handler(_, Goal) :-
(find_definition(Goal) ->          call(Goal)        ;
error_handler(undef_call, Goal)     ).
```

If the definition is found, the predicate is restarted, otherwise a default handler is invoked which prints the message and aborts.

Another example of exception usage is the definition of evaluable functors. In the ISO proposal [4] it is possible to call user predicates in an arithmetic expression. Such cases are easily handled by our event mechanism, which has the advantage that the arithmetic can still be in-line compiled.

The more exception types that are available within a Prolog system, the more flexible the system is and it can in fact be customized to particular needs. This ability was much used in the SEPIA system, since it is a basis system for integration of Prolog extensions, and very often the extensions have their own requirements on the behavior of the system in nonstandard cases. For example, if a call is not sufficiently instantiated, the handler in SEPIA can suspend it and the call is woken when one of its arguments becomes more instantiated. When no suspension is used, such calls issue an instantiation fault.

5.2 Interrupts

The proposed scheme differs from other current Prolog systems in that it is able to handle any external interrupts, not just a break key from the user, the handling is performed in real time, with an immediate response and the action is user-definable for all interrupts. This scheme has been used in SEPIA to implement an interface to the PCE graphic system [1]. PCE is running as a separate process which communicates with Prolog using a pipe. If the Prolog system is not able to process interrupts, it has to poll this interface in a busy loop. In SEPIA, the interface is exactly the same as it would be for a normal Prolog system, the only difference is that the pipe that connects it to PCE has been set up to send a signal if it receives some data. The corresponding event handler then reads the data from the pipe, executes the required action and then returns to the normal execution. This means that Prolog can run as usual, either execute a program or communicate with the user via the keyboard, but it can at the same time respond to the mouse-driven events.

Another example is writing a clock in Prolog. SEPIA is set up to receive a signal from the operating system every second. The interrupt handler updates the clock on the screen, exits and the normal execution continues, no matter what it is.

The presented interrupt handling scheme has all the properties which are necessary for real-time applications, and the SEPIA system is thus a tool for testing how Prolog can be exploited for real time tasks.

Acknowledgements

Pierre Dufresne has implemented the event handling scheme in SEPIA and solved some problems of interrupt handling. Joachim Schimpf has cleaned it and thoroughly tested to make sure that an interrupt can really come at any time and also discussed many times about the event handling. A referee of a previous version of this paper pointed out problems with the garbage collection. David Miller did his best to improve my bad english style.

Bibliography

[1] A. Anjewierden. Pce-Prolog 1.0 reference manual. Technical report, University of Amsterdam, October 1986. ESPRIT Project 1098.

[2] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[3] Keith Clark and Steve Gregory. Notes on systems programming in Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 299–306. ICOT, 1984.

[4] P. Deransart, P. Folkjaer, J-F. Pique, and R. S. Scowen. Prolog draft for working draft 1.0. Technical Report N28, International Organization for Standardization, February 1989.

[5] Ian Foster. Logic operating systems: Design issues. In *Proceedings of the 4th ICLP*, pages 910–926, Melbourne, May 1987.

[6] Micha Meier. Shallow backtracking in Prolog programs. Technical Report TR-LP, ECRC, February 1987.

[7] Micha Meier. Analysis of Prolog procedures for indexing purposes. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 800–807, Tokyo, November 1988.

[8] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[9] R. A. O'Keefe. On standardising Prolog. DAI Working Paper 196, Department of Artifical Intelligence, University of Edinburgh, October 1984.

[10] Hervé Touati and Toshiyuki Hama. A light-weight Prolog garbage collector. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 922–930, Tokyo, November 1988.

[11] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI, October 1983.

[12] Toshio Yokoi, Shunichi Uchida, and ICOT Third Laboratory. Sequential inference machine: Sim. its programming and operating system. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 70–81, Tokyo, 1984.