SICS

The Engine-Scheduler Interface used in the Muse OR-parallel Prolog System

Roland Karlsson **Khayri A. M. Ali**

SICS research report
R92:04
ISSN 0283-3638

The Engine-Scheduler Interface used in the Muse OR-parallel Prolog System

March 1992

Roland Karlsson Khayri A. M. Ali

SICS, Swedish Institute of Computer Science
PO Box 1263
S-164 28 Kista, Sweden

ALMOST any sequential Prolog system is in principle easy to extend for OR-parallelism, using the Muse execution model. To reduce your programming effort we have implemented the Muse scheduler, with a clean interface to the Prolog sequential engine. This interface is implemented as a set of C macros. The sequential Prolog system to be parallelized uses some of those macros provided by the Muse scheduler and must also provide some macros for the Muse scheduler. This paper contains a definition and description of the required macros, emphasizing information needed by the Prolog engine programmer.

Contents

1 Introduction

This paper documents the interface between the Muse scheduler and Prolog engines based on the WAM. The interface has been implemented for the SICStus Prolog engine (version 0.6) [5] and tested on a number of multiprocessor machines (and Sun workstations).

In this document, we assume that the reader is familiar with the WAM [8], and the Muse approach [2, 1]. We also recommend reading the paper [6] describing the actual Muse implementation. In Muse, a number of workers (processes) explore OR-parallelism in a Prolog search tree. Each worker has two components – an engine, which executes Prolog code, and a scheduler, responsible for distributing the available work among the workers. There is a well defined interface between the engine and the scheduler code that enables different Prolog engines based on the WAM to be used with the Muse scheduler. At present there is only one Prolog engine used with the Muse scheduler. The engine is based on SICStus version 0.6. There is ongoing work for using the sequential BIM Prolog engine with the Muse scheduler.

In Muse, the search tree consists of a number of nodes of two kinds – shared and private. Shared nodes make up the shared portion of the search tree. Private nodes are accessible only to the worker that created them. The tree is divided into the upper shared section and the lower private section. Each worker's engine performs a sequential traversal of its private region. Anytime a worker has to access the shared part of the search tree, it calls its scheduler. Thus the scheduler provides synchronization between the engines.

In Muse, each Prolog engine has its own copy of the WAM (or Prolog) stacks. There is also some shared memory for representing shared nodes, global tables (e.g. atom and predicate tables), and some global registers. When a worker runs out of work, it shares some nodes with another worker and copies the difference between the two workers' states. Then through the normal backtracking of Prolog, a worker gets work from the shared nodes.

The work documented here is much influenced by the Aurora interface [7]. One motivation for defining an interface between the Muse scheduler and the Prolog engines based on WAM is that the BIM company has assigned a subcontract to SICS in the PEPMA framework for extending the sequential BIM Prolog to OR-parallel execution using the Muse approach. SICS defines the algorithmic interface between the Muse scheduler and the BIM engine, and the BIM group implements the engine part of the interface. The existing Muse scheduler is used, providing a set of macros for the engine.

Ideally, the interface definition should be general enough to enable different schedulers and Prolog engines to be used with each other. The Aurora interface allows

different schedulers to be plugged into the same engine, whereas the Muse interface is directed towards allowing different Prolog engines to be plugged into the same scheduler. Defining and implementing a more general interface that combines both could be a next step.

We have tried to reuse as much as possible from the Aurora interface work in order to take advantages of the Aurora results. Since the Muse model differ from the SRI model [9] (used in Aurora) and the search tree is represented differently in Muse than in Aurora, many of the interface macros are different.

The definition documented here does not fully cover some aspects of the engine/scheduler interface within Muse (e.g. those related to debugging and performance analysis). We are also expecting further feedback from the BIM group which may lead to some more modifications in the interface.

As a bonus for us, the interface work has improved the structure and readability of the Muse code, and also enabled us to discover and remove redundant code. The interface work has not introduced any negative effect of the performance results. Some experimental results of the new Muse system are found in [4, 3].

In the next two sections, we describe and define the engine and scheduler duties. Section 4 presents an overview of the Muse interface and discusses the differences compared to the Aurora interface. In Section 5 we describe the Muse interface. Section 6 concludes the paper.

2 The Prolog Engine

The core of the Prolog engine is a conventional sequential Prolog engine. This core is extended for OR-parallel execution.

Since different sequential Prolog engines can be implemented differently and the representation of data and code is known only to the engine, it is natural for the engine to perform memory management. The memory management has to make a distinction between local and global memory. Some of the information stored in the local memory of the sequential Prolog system has to be made globally known to the other workers. This information includes the Prolog program, hash tables for atoms, the dynamic predicates etc. The engine also provides shared memory areas for storing information associated with the shared nodes and for communication between workers (defined in Section 5.1).

The engine also does process management. This includes forking workers, killing workers, and providing functions for suspending and resuming processes.

Some global information (e.g. atom tables) can be accessed by any worker asynchronously as an atomic operation. This can be achieved by using locks (or

semaphores, etc) to serialize the access. The engines perform this access without any help from the scheduler. For synchronous access to other global information the engine has to call the scheduler to make sure that the worker is in the leftmost branch of the Prolog search tree.

Whenever there might be some interaction with other workers the engine calls an appropriate scheduler function. This includes: 1) checking arrival of (prune or share) requests from other workers, 2) entering the shared section of the search tree, 3) performing a pruning operation.

In the Muse model, each worker has its own copy of the Prolog stacks. After sharing, the two workers involved in the sharing operation have roughly identical memory images. Since the representation of the Prolog stacks is known to the engine, the engine performs copying and installation of the Prolog stacks.

The scheduler needs some help macros to be able to fulfill its purpose. There are macros providing information about the state of the engine and also macros that perform certain engine specific tasks.

3 The Muse Scheduler

> *"The two main functions of the scheduler are to maintain the sequential semantics of Prolog and to match idle workers with available work with minimal overhead."*

The central idea is the notion of a shared search tree. Choicepoints in the sequential Prolog machine will become nodes in this tree. The Muse scheduler maintains a shared search tree, where each worker only has access to nodes within its own branch. The shared part of the search tree grows when a worker makes its private nodes shareable and it shrinks when the last worker backtracks from a shared node.

Whenever an engine runs out of work within its local subtree, it calls the scheduler for work. The scheduler first tries to find work on shared nodes in the current branch. Then it tries to find another worker that has either private excess load or shared excess load. In either case, it requests sharing work from that worker. If no work can be found then the scheduler stays at a suitable place in the search tree. When no work remains within a subtree all workers leave that subtree.

Whenever an engine performs a synchronous side effect, it calls the scheduler to check whether or not the current worker is in the leftmost branch of the search tree. So, the scheduler has to support that test.

When an engine performs a cut (or commit) operation, it calls the scheduler to prune branches according to the semantics of the cut operation. Since the Muse

scheduler allows execution of speculative work (i.e. work that might not be executed by the sequential Prolog system), the scheduler provides detection and correction for the case when speculative work has to be aborted.

4 An Overview of the Interface

In this section we give an overview of the Muse interface and point out the relations between the Muse interface and the Aurora interface [7].

In the current Muse interface, as in the current Aurora interface, the execution of a Prolog program is governed by the engine: whenever the engine runs out of work, it calls an appropriate scheduler macro to provide a new piece of work. As pointed out in [7], the advantage of this scheme is that the overhead for switching between the engine and the scheduler is much smaller than it is when the Prolog execution is governed by the scheduler. The reason for the reduced overhead is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when the engine runs out of work and need not be rebuilt when the engine gets a new piece of work.

We give in Sections 4.1 and 4.2 an overview of the macros defined by the scheduler and the engine respectively. In Section 4.3 we give the scheduler and the engine macros used for sharing nodes between workers. Notice that the names of all scheduler macros are prefixed with **Sch_** and all engine macros are prefixed with **Eng_**, as a convention.

4.1 Scheduler Macros

Figure 1 shows the main five macros provided by the Muse scheduler for the engine during work. The scheduler macros are arranged in three groups, following the Aurora classification [7]:

- finding work (**Sch_Get_Work**);
- communication with other workers (**Sch_Check, Sch_Prune, Sch_Synch**);
- events of interest to the scheduler (**Sch_Set_Load**).

Sch_Get_Work is called when the engine dies back to a shared node. Its purpose is to find a new piece of work.

Sch_Check is called at every Prolog procedure call to check arrival of requests from other workers.

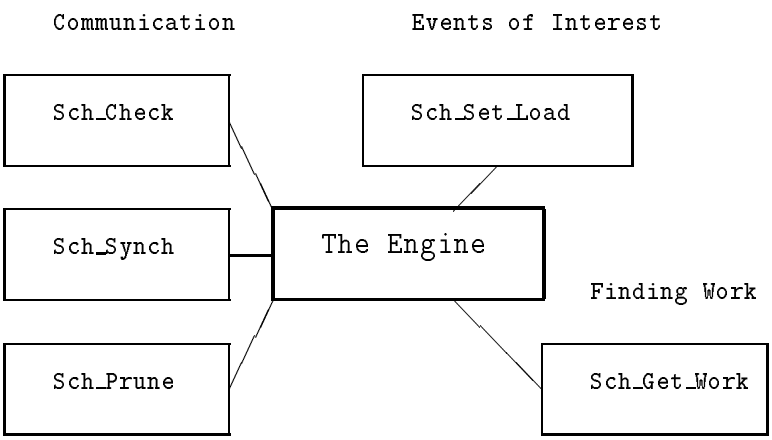

Figure 1: Macros called by the Prolog engine.

Sch_Prune is called when a cut or commit is executed.

Sch_Synch is called when a side effect is encountered.

Sch_Set_Load is called when the engine creates a choicepoint. The scheduler maintains information about the current local load of the engine.

This set of macros can roughly be seen as a simple subset of the scheduler macros of the Aurora interface [7].

4.2 Engine Macros

Let us now look at the other side of the interface: macros provided by the engine for the scheduler. The main data structure shared between the engine and the scheduler is the representation of the Prolog search tree. Nodes in the search tree are either private or shared. Each private node is represented by the normal Prolog choicepoint. Each shared node is represented by two physically separated data structures: the normal Prolog choicepoint and a shared frame.

The engine provides the scheduler with information about **choicepoints** (e.g. the parent of a given choicepoint), the type of a predicate (sequential or parallel) associated with a choicepoint, etc. The engine also maintains the representation of the Prolog clauses. So, the engine provides the scheduler with macros that manipulate **alternative clauses**. It also provides the scheduler with the current **load**.

Figure 2 shows macro groups provided by the engine. They are classified into **clause** macros, **choicepoint** macros and local **load** macros.

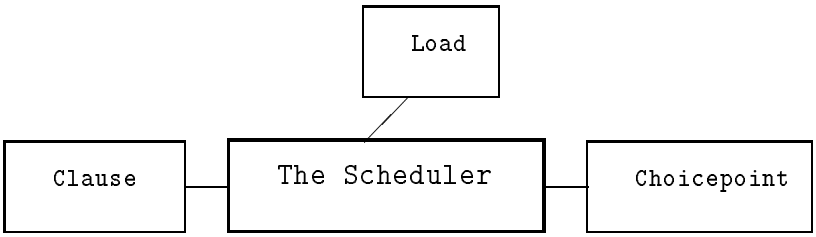

Figure 2: Macro categories provided by the Prolog engine.

Each shared frame contains scheduler specific information associated with the node. The scheduler also maintains the boundary between the shared and the private sections and tells the engine when it is updated.

If we compare this side of the Muse interface with the engine side of the Aurora interface, we find many differences between the two interfaces. In Aurora, all nodes are identical and maintained by the engine. The notion of *embryonic* nodes in Aurora does not exist in Muse. The representation of the search tree in Aurora is more general than the one used in Muse. Moreover the SRI model (used in Aurora) and the incremental stack-copying model (used in Muse) require different installation methods when moving down in the shared search tree. These differences explain why the engine side of the Muse interface is different from the engine side of the Aurora interface.

4.3 Sharing Macros

The main scheduler and engine macros used for starting and performing the sharing session are outlined in Figure 3. Q and P stand for a worker requesting sharing and a worker requested for sharing, respectively.

When Q's engine finishes processing its current piece of work (task), it calls the scheduler macro **Sch_Get_Work()** for finding a new job. If the scheduler cannot find work in the current branch and there exists another worker P with excess load, Q starts the sharing session with P as follows. The **Sch_Get_Work()** macro asks the engine to prepare for the sharing session through invoking the **Eng_Prepare()** macro and then it requests sharing from P, through invoking the **Eng_Q_Share()** macro. Upon a successful sharing session the **Eng_Q_Share()** macro returns the new top. Otherwise it returns zero.

The worker P detects the sharing request when its engine invokes the **Sch_Check()** macro. P may accept or refuse the sharing request. In the latter case, the **Sch_Check()** macro tells the engine to refuse the sharing request through invoking the **Eng_Refuse()** macro. Otherwise, the **Sch_Check()** macro starts the sharing session by invoking the **Eng_P_Share()** macro.

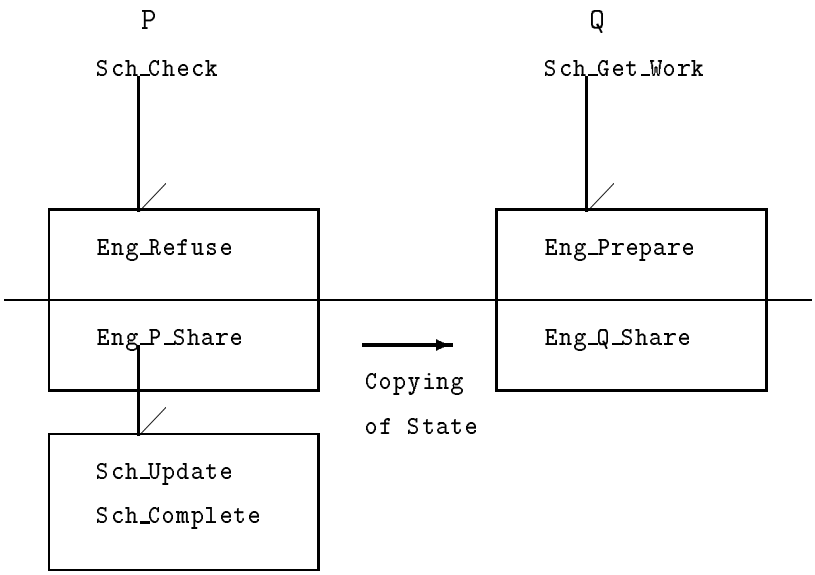

Figure 3: The sharing macros.

During the sharing session, the **Eng_P_Share()** macro will request the scheduler to add Q in all nodes that are already shared by P but not by Q through invoking the **Sch_Update()** macro. It will also request the scheduler to fill in the new shared choicepoints with the scheduler information through invoking the **Sch_Complete()** macro.

Copying of the state from P to Q will be performed by the two engines as a combined effort through the two macros **Eng_P_Share()** and **Eng_Q_Share()**.

5 The Interface

In this section we define the data structures used by the interface and the macros defined by the scheduler and the engine. Notice that a node is an abstraction that is implemented by a choicepoint and for a shared node, also a shared frame.

5.1 Data Structures

The Muse scheduler requires the engine to define the `struct` data structures shown in Figure 4. Those data structures can be used by the engine programmer for engine specific data. The data structures can be empty.

The Muse scheduler also requires the engine to define the `struct` data structure

```
struct Eng_Private {
  Any data strictly private to the worker.

  Accessed by pw.Eng.'fieldname'.
};

struct Eng_Local {
  Any data stored locally that is remotely accessible.   One example
  is data used for communication at the sharing session.

  Accessed by lw->Eng.'fieldname'
        or by rw[worker_id]->Eng.'fieldname'.
};

struct Eng_Global {
  Any globally stored data.  One example is data that has to
  be globalized such as an atom table pointer (and its
  associated lock).

  Accessed by gw->Eng.'fieldname'.
};
```

Figure 4: Various convenient data structures.

shown in Figure 5. This data structure can be used by the engine programmer for storing engine specific data associated with each shared choicepoint. The data structure can (in principle) be empty, but we assume that the engine programmer stores the `next-clause-to-try` pointer in the `Eng_Stry_Node` structure.

```
struct Eng_Stry_Node {
  Any globally stored data associated with a shared frame.

  TRY *next-clause-to-try.

  accessed by 'framepointer'->Eng.'fieldname'.
};
```

Figure 5: The shared frame data structure.

Some data types are supposed to be known both to the Muse scheduler and to the Prolog engine. These are listed in Figure 6. The first column contains the C data type used. The second column gives a short form used in this paper. The third column is used for a short description of the data types. It is important to understand that the Muse scheduler expect the Prolog engine to define the `struct node` and the `struct try_node` data types. The Muse scheduler defines the `struct stry_node` data type.

Data type	Short form	Description
struct try_node	TRY	A Prolog clause.
struct node	CP	A Prolog choicepoint.
struct stry_node	STRY	A shared frame.
-	CODE	A piece of C code.
int	BOOL	An integer used as a boolean.
unsigned int	u_int	An unsigned integer.

Figure 6: Used data types.

5.2 Prolog Engine → Muse Scheduler

The Prolog engine can call some macros defined by the Muse scheduler, outlined in Figure 1. Some of the macros are mandatory to use and some are auxiliary. The auxiliary macros are marked with a †.

5.2.1 Macros for Communication with Other Workers

The Muse scheduler provides some macros for communication with other workers. The macros might move the border between private and shared choicepoints. This border is called the *top*. The engine may have to change its pointer to the youngest choicepoint (called *current-CP*) according to the top when the top changes.

void Sch_Check(CODE failcode)

This macro is called regularly (e.g. at every procedure call). The scheduler uses this macro to detect requests for sharing and requests to abort the current local execution. It is also used to implement the delayed release function.

When this macro detects a sharing request, the scheduler either refuses or accepts the request (Section 5.3.4). A successful sharing may change the top. In the case of a delayed release request the scheduler releases the current load after a fixed number of calls (Section 5.2.3). In the case of a legal request for abortion of the current local execution, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 5.2.4) and then it backtracks.

void Sch_Prune (CP *scope, BOOL is_commit, CODE failcode, CODE prunecode)

This macro is called before executing a cut instruction. The scheduler uses the macro to perform pruning of subtrees that would not have been visited in a sequential

execution. It also requests abortion of the work of workers that are staying in those subtrees. This macro generally sets the value of the local load register.

The **scope**[1] of the cut operation is provided. This parameter is both input and output. A boolean flag (called **is_commit**) that is true if the cut operation shall be treated as cavalier commit, and false otherwise, is also provided.

A pruning operation is performed only when the **scope** reaches into the shared region. At a successful pruning operation the top may be changed, the **scope** may be changed, and then the code **prunecode** is executed. The **scope** value, when executing **prunecode**, is the same as the output value for **scope**. The pruning operation may be aborted by another pruning operation. In this case, the top may be changed, the **scope** is set to 0, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 5.2.4) and then it backtracks.

When the cut operation is local the **scope** value is unchanged. In the case of a local cut or a successful pruning operation, the output **scope** value is used when performing the local cut operation.

void Sch_Synch(CP *scope, CODE failcode)

This macro is called whenever the engine cannot continue forward execution until it becomes leftmost in a search tree (e.g. before a side effect). It is used to maintain the (sequential) Prolog semantics. No side effect is executed until the current branch is leftmost in a proper subtree. Most side effects (e.g. write, assert) use the search tree defined by the whole computation, but closed sub-computations (e.g. bagof) can use a subtree defined by the computation.

The **scope**[2] of a proper subtree is provided. In case of a legal request for abortion of the current local execution, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 5.2.4) and then it backtracks.

5.2.2 Macros for Finding Work

The Muse scheduler provides some macros for finding work when the engine backtracks to a shared choicepoint. The macros might move the top. Whenever work is found the engine is supposed to set the program counter (or equivalent) according to a returned clause pointer.

[1] A pointer to the youngest choicepoint that is alive after performing the cut operation
[2] A pointer to the choicepoint which is the common root that defines the subtree

void Sch_Get_Work(CODE failcode, TRY *try)

The macro is called whenever the engine backtracks to a shared choicepoint. It is a very central macro to the Muse scheduler. (As an optimization this macro consists of a set of three macros **Sch_Get_Work_Parallel()**, **Sch_Get_Work_Sequential()**, and **Sch_Get_Work_Root()**. They all have the same syntax and semantics. But the Muse scheduler expects the Prolog engine to call the correct one depending on the current shared choicepoint type.) It is used to find shared work. The scheduler may take shared work from the current (shared) choicepoint or it may move to an older choicepoint containing shared work or it may, with the help of another worker, install a new environment extending its branch. When no shared work can be found the worker may stay at the choicepoint or move to an older choicepoint for a better position.

There are several reasons for the engine to backtrack after the return from this macro; the scheduler wants to move up in the tree; a pruning request has been detected; backtracking for job after a successful sharing session. In all cases, the top may be changed, and then the code **failcode** is executed. This code first sets the current-CP to top (Section 5.2.4) and then it backtracks.

The scheduler may find job in the current choicepoint. The engine, for that purpose, provides an output field of the type **TRY**, a clause pointer. The scheduler fills in that field with a pointer to the found clause. The engine is supposed to execute that clause. Then, if the last alternative in a sequential choicepoint is taken, top is changed. The engine changes its current-CP accordingly.

5.2.3 Macros for Events of Interest to the Scheduler

The Prolog engine is supposed to inform the Muse scheduler about some events. Currently the only event is change of the current load. For more efficient handling of cut this group of macros might grow substantially.

void Sch_Set_Load(u_int load, BOOL stable_load)

This macro is called whenever creating a (complete) choicepoint. Then the macro informs the scheduler about both the current load and whether the current load is stable or not. Stability is defined, in the current Muse implementation, as having an older parallel private choicepoint (with extra alternatives). The purpose of providing information about the stability is to avoid showing excess load that may disappear after a very short period of time.

The **load** argument contains the current load. This value is (normally) copied to the load register. The **stable_load** argument contains a boolean stating whether

the provided load shall be regarded as stable or not. The exact definition of **stable_load** can be different from one engine implementation to another, but it shall reflect whether there already did exist any local load before calling the macro. (The **stable_load** argument can contain the previous load value. This is consistent with using a C boolean.) If the load is not stable the release of the load is delayed for a fixed number of calls to the macro **Sch_Check()**.

5.2.4 Macros for Asking the Scheduler

Some of the scheduler information is relevant to the engine. In this group of macros the engine asks the scheduler to get such information. So far, there is only one macro. It is used for getting a pointer to the current top. In contrast to Aurora, Muse stores information about the current top (the boundary between the shared and the private regions) in the scheduler part.

CP *Sch_Get_Top()

The Prolog engine calls this macro each time it needs to update the engine state according to a (possibly) new top.

This macro returns a pointer to the current top, the youngest shared choicepoint. Calling this macro has no effect on the Muse scheduler.

5.2.5 Sharing Macros

As described in Section 4.3, the scheduler can find another worker to get shared work from through the macro **Sch_Get_Work()**. The scheduler then request sharing from the other worker. We call the requesting worker Q and the other worker P. The worker P detects the request in the macro **Sch_Check()**.

void Sch_P_Update_Nodes(CP *old_ptop, CP *qtop, u_int q_id)

This macro is called from **Eng_P_Share()** during the sharing session. It adds Q as a member to all already shared choicepoints accessed by P but not by Q.

Three arguments are provided. The **old_ptop** is the old top for P, the **qtop** is Q's top, and **q_id** is the name of Q.

void Sch_P_Complete_Stry(STRY *stry, CP *next_job)

This macro is called from **Eng_P_Share()** for every choicepoint that is made shared by P. The purpose of the macro is to fill in the extra scheduler fields in the newly allocated `STRY` structure.

A pointer to the newly allocated `STRY` (**stry**) is provided. A pointer to a good choicepoint (**next_job**), to start looking for work (when running out of work in the current choicepoint) is also provided. This choicepoint must be older than the current choicepoint and not older than the nearest parallel choicepoint that contains available work. In the current Muse implementation the **next_job** is the parent of the current choicepoint.

5.2.6 Initialization Macros

The start up of the Muse system consists of a number of steps. The start up (master) process allocates shared memory (including the stack areas) for all processes. It performs initializations of the Prolog engine (and the scheduler) before executing any Prolog code. The Prolog system is loaded using the master process, the root choicepoint is made shareable, the stacks are copied to all other workers, and then the processes are forked. The master then waits until all other processes are created, initialized correctly, and sharing the root choicepoint. Thereafter the system is ready.

void Sch_Init(u_int num_workers)

This macro is called by the master process (worker id 0) before running any engine code and before forking any other worker, but after allocating and setting up of shared memory and creating the initial choicepoint (older than the root). The number of workers (**num_workers**) is provided.

void Sch_Add_Remote(u_int wid)

This macro is called by the master process just before fork, for every other worker. The worker id (**wid**) is provided.

void Sch_Add_Private(u_int wid, u_int num_workers)

This macro is called by each worker, except the start up worker, just after fork. The worker id (**wid**) and the number of workers (**num_workers**) are provided.

void Sch_Wait_For_Forked_Children()

This macro is called by the start up worker just after fork and before continuing running Prolog. The purpose is to wait until all other workers have been started.

void Sch_P_Initial_Share()

This macro is called just before the initialization macro **Sch_Add_Remote()**. The purpose is to make the root choicepoint shared.

void Sch_Q_Initial_Share()

This macro is called just after calling the initialization macro **Sch_Add_Private()**. The purpose is to propagate information about the shared root choicepoint.

5.2.7 † Auxiliary Macros

The following macros are not necessary to use. The Scheduler provides them for auxiliary use.

† void Sch_Set_Load_Now(u_int load)

When the engine wants to (temporarily) make itself uninteresting (e.g. at GC) it calls this macro with a load of 0. When the engine wants to make itself interesting again it calls this macro with the current load. The macro can replace the **Sch_Set_Load()** macro, if delayed release is not wanted.

† u_int Sch_Num_Workers()

This macro returns an unsigned integer indicating the number of workers currently in the system.

† u_int Sch_Worker_Id()

This macro returns an unsigned integer indicating the worker identification. The identification is in the range $\langle 0 \ldots Workers - 1 \rangle$.

† **void Sch_Write**
(NODE *scope, STRY *stry, CODE execcode, CODE savecode,
CODE failcode,BOOL retry)

This macro can be used as an optimization for write type side effects (e.g. assert, write, and saving solutions in findall). Instead of using **Sch_Synch()** to suspend the execution of the current task non-leftmost workers can save the side effect for later execution.

The **scope** and the **failcode** have the same meanings as for **Sch_Synch()**. The `STRY` **stry** is an output field pointing to an appropriate `STRY` to save non-executed side effects in. It is only valid iff the `CODE` **savecode** is executed. The `CODE` **execcode** is the code to execute iff the side effect can be executed. The `CODE` **savecode** is the code to execute iff the side effect has to be saved until later execution. The boolean flag **retry** is true iff the call to the macro is made after claiming saved side effects.

The scheduler needs some help from the engine, so if this macro is used the engine must implement the macros **Eng_Any_Saved_Writes()** and **Eng_Propagate_Writes()**.

5.3 Muse Scheduler → Prolog Engine

The Prolog engine provides some macros for the Muse scheduler, outlined in Figure 2 (page 6). Most of the macros are mandatory to define. But some, if not defined, are replaced with dummy macros that always return a default value. Macros that do not have to be implemented are marked with a †.

5.3.1 Load Macros

It is up to the Prolog engine to define and maintain the current excess load. The engine provides a macro that can return such a value. The load shall be a measure of the amount of untried local work that can be found via backtracking (e.g. the number of untried local alternatives).

u_int Eng_Get_Load()

This macro returns the current excess local load. When no load exists the macro returns the value 0. The macro is used for some important scheduler optimizations.

5.3.2 Clause Macros

Some macros are used for getting and manipulating available alternatives in shared choicepoints. For this purpose, the engine maintains the `next-clause-to-try` field for shared choicepoints, in the associated `STRY` structure shown in Figure 5 (page 8). The Muse scheduler must be able to get the current `next-clause-to-try` for shared choicepoints. The scheduler must also be able to advance it to the next alternative and to clear it. Clearing corresponds to executing a cut operation in the sequential Prolog engine.

Other macros are used for implementing the leftmost (in the search tree) check. For this purpose, the engine for each worker in a parallel shared choicepoint maintains information about which alternative is taken. This information shall be accessible to the scheduler. The scheduler must also be able to clear the value, indicating that no alternative is taken. The value returned by **Eng_My_Alternative()** and **Eng_Remote_Alternative()** is 0 for sequential choicepoints and when no alternative is currently taken.

TRY *Eng_Get_Next_Alternative(STRY *stry)

This macro returns a pointer to the `next-clause-to-try` for a shared choicepoint, referenced by **stry**. A returned null pointer means that there are no more alternatives. The `next-clause-to-try` value shall *not* be advanced.

void Eng_Adv_Next_Alternative(STRY *stry)

This macro tells the engine to advance the `next-clause-to-try` for the shared choicepoint referenced by **stry**.

void Eng_Clear_Next_Alternative(STRY *stry)

This macro tells the engine to mark the shared choicepoint referenced by **stry** as empty. All subsequent calls to **Eng_Get_Next_Alternative()** for this choicepoint, shall return the value null.

u_int Eng_My_Alternative(STRY *stry)

The purpose of this macro is to give an ordering of taken alternatives. For sequential choicepoints this ordering is not interesting as only one alternative can be taken. This macro is very central for implementing the sequential Prolog semantics.

This macro returns an unsigned integer representing my-current-alternative for a shared choicepoint referenced by **stry**. A returned non-zero value means that I have currently taken a alternative. For parallel choicepoints the value then returned is one monotonically decreasing with the corresponding clause number. For sequential choicepoints the value is not important as long as the same value is always returned for a alternative. A returned 0 means that I have currently not taken any alternative in this choicepoint.

u_int Eng_Remote_Alternative(STRY *stry, u_int workerid)

This macro is equivalent to the previous one, except for returning the value for a worker given by **workerid**.

void Eng_Take_Alternative(STRY *stry, TRY *try)

This macro informs the engine that the scheduler has reserved the clause **try** in the parallel shared choicepoint referenced by **stry**. The engine then updates the my-current-alternative value accordingly.

void Eng_Take_No_Alternative(STRY *stry)

This macro informs the engine that the scheduler has not taken any alternative in the parallel shared choicepoint referenced by **stry**. The engine updates the my-current-alternative value to 0.

5.3.3 Choicepoint Macros

Only the Prolog engine knows the choicepoint representation. The engine therefore has to provide some macros for the Muse scheduler. The **CP** type is a choicepoint reference (usually an address). The choicepoint stack may be moved, so using a **CP** reference in the scheduler might not always be a good idea. The engine provides a location independent choicepoint reference called **offset**. Conversion macros back and forth between the **CP** and the **offset** representations are provided. The engine also provides age comparison macros both for the **CP** and the **offset** representations. Some information about a choicepoint, such as its parent, is also provided by the engine.

The set of macros concerning the age of choicepoints, direct or via offset, may look somewhat arbitrarily chosen. Only those actually needed by the scheduler are described. The set of macros could be made more complete if desired.

The engine maintains means to get pointers to the child and the shared frame of a shared choicepoint.

CP *Eng_Parent(CP *cp)

This macro returns the parent of a choicepoint.

BOOL Eng_Is_Younger_CP(CP *cp1, CP *cp2)

This boolean macro returns true, iff the choicepoint **cp1** is younger than the choicepoint **cp2**.

CP *Eng_Get_Oldest_CP()

This macro returns a pointer to the oldest existing choicepoint. This choicepoint must be older than the oldest shared choicepoint (called root).

CP *Eng_Get_Youngest_CP()

This macro returns a pointer to the youngest choicepoint now existing in the engine.

u_int Eng_CP_To_Offset(CP *cp)

This macro converts from a **CP** pointer to a Prolog engine independent offset. The offset for a given choicepoint must never change during the execution and it must also be an unique value for all choicepoints in the same branch.

CP *Eng_Offset_To_CP(u_int offset)

This macro converts from a Prolog engine independent offset to a **CP** pointer.

BOOL Eng_Is_Older_Offset(u_int o1, u_int o2)

This macro returns the value true, iff the offset **o1** represents an older choicepoint than the offset **o2**.

u_int Eng_Infinitely_Young_Offset()

This macro returns an offset that represents a choicepoint that is younger than any real choicepoint.

CP *Eng_Child(CP *cp)

This macro returns the (shared) child of the shared choicepoint **cp**. The scheduler calls this macro only for shared choicepoints that have a shared choicepoint as child. This restriction makes it possible for the engine in the SICStus-Muse system to use the same choicepoint field for calculating the local load and as child reference.

STRY *Eng_Stry(CP *cp)

This macro returns a pointer to the shared frame associated with the shared choicepoint **cp**. The scheduler calls this macro only for shared choicepoints.

BOOL Eng_Is_Parallel(CP *cp)

This macro returns true iff the predicate that has created the choicepoint **cp** is a parallel predicate.

5.3.4 Sharing Macros

The workers P and Q are defined in Section 5.2.5.

void Eng_Q_Prepare_For_Sharing(u_int wid)

This macro prepares for the protocol used by the engine at the sharing session. It is called from **Sch_Get_Work()** by Q before the scheduler sends the request for sharing request to P. The worker id (**wid**) for P is provided.

CP *Eng_Q_Share(u_int wid)

This macro is called from **Sch_Get_Work()** by Q after the sending of a request to P (**wid**) for sharing request. After a successful sharing session it returns the new top. If sharing was refused it returns 0.

CP *Eng_P_Share(u_int load, CP *cp, u_int wid)

This macro is called from **Sch_Check()** by P when detecting a request for sharing. The correct **load** and an id to Q (**wid**) are provided. If **load** is zero then the scheduler may have found work in some shared choicepoint. A pointer to this choicepoint is provided in **cp**. At a successful sharing session the macro returns a pointer to the new top. If sharing was refused it returns 0. In that case, the scheduler calls the macro **Eng_P_Refuse_Sharing_Request()**.

void Eng_P_Refuse_Sharing_Request(u_int wid)

This macro is called from **Sch_Check()** by P when a detected request for sharing is to be refused. The id of Q is provided.

void Eng_Stry_Dealloc(STRY *stry)

This macro is called when the scheduler wants to deallocate the STRY **stry**.

5.3.5 Initialization Macros

The role of these macros is described in Section 5.2.6.

CP *Eng_P_Initial_Share_Root(u_int wid)

This macro is called from **Sch_P_Initial_Share()**. The id of the start up worker (**wid**) is provided. The macro makes the root choicepoint shared and then returns a pointer to it.

void Eng_P_Initial_Copy_Stacks()

This macro is called from **Sch_P_Initial_Share()** for each worker other than the start up worker. The id of the other worker (**wid**) is provided. The macro performs the operations by P that are necessary to copy the state from P to Q.

CP *Eng_Q_Initial_Copy_Reg()

This macro is called from **Sch_Q_Initial_Share()** by each worker other than the start up worker. The macro performs the operations by Q that are necessary to copy the state from P to Q.

5.3.6 † Auxiliary Macros

These macros are used to implement non-Prolog features. The engine can choose not to implement them. They are then automatically replaced with dummy macros that return default values.

† BOOL Eng_Is_Atomic()

This macro returns the value true iff the Prolog engine is in atomic mode. The default value is false. This engine mode makes the scheduler ignore requests in the macro **Sch_Check()**. The engine shall only be in this mode for short periods and only when executing Prolog code with exact one solution.

There may be some use of this macro besides implementing non-Prolog features. The Prolog engine may have to force a sequence of Prolog predicate calls to be made as an atomic operation. In the SICStus implementation there exists a special metacall for this purpose. It is implemented as ...

```
atomic_call(Goal) :- atomic_on, call(Goal), atomic_off.
```

... where the built-in predicates `atomic_on` and `atomic_off` sets and resets the atomic flag. Remember the limitations for the goal `Goal`. It is not a good idea to make the predicate `atomic_call/1` public.

† BOOL Eng_Is_Mutex()

This macro returns the value true iff the Prolog engine is in mutex mode. The default value is false. If the engine implements the mutex function, only one worker at a time shall be able to return the value true to this macro. (If this macro returns true then the engine must not call the macro **Sch_Synch()** with a scope that may suspend current job. This to avoid deadlock.)

† BOOL Eng_Is_Mutex_Remote(u_int workerid)

This macro returns the value true iff the Prolog engine of the worker with id **workerid** is in mutex mode. The default value is false.

† BOOL Eng_No_Sharing()

This macro returns the value true iff the Prolog engine does not allow sharing. The default value is false.

† **BOOL Eng_No_Sharing_Remote(u_int workerid)**

This macro returns the value true iff the Prolog engine of the worker with id **workerid** does not allow sharing. The default value is false.

† **BOOL Eng_Any_Saved_Writes(STRY *stry)**

This macro returns the value true iff there exists any saved side effects in the `STRY` **stry**. When the **Sch_Write()** optimization is not used then the macro returns the default value false

† **BOOL Eng_Propagate_Writes(STRY *stry, u_int alternative)**

This macro is called when the scheduler detects (with **Eng_Any_Saved_Writes()** that the `STRY` **stry** may contain saved side effects. The **alternative** argument contains the alternative number for the leftmost alternative in the shared choicepoint referred to by **stry**. (The `STRY` **stry** is locked when the calling the macro.) For all saved side effects the macro calls **Sch_Write()** to propagate the side effects further in the tree or eventually perform the side effects. When the **Sch_Write()** optimization is not used then the macro is a dummy macro, expanding to the empty macro.

6 Conclusions

The engine-scheduler interface in the Muse system has been described. This interface is working in the current Muse system, which is based on SICStus0.6 Prolog engine with the Muse scheduler. Performance results on six different machines indicate that the interface has not introduced any negative effect on the performance results. As a result of the interface work, the Muse code becomes more readable and redundant code has been discovered and removed.

It remains to establish that the engine-scheduler interface documented here fits other Prolog engines. There is ongoing work interfacing the sequential BIM Prolog engine with the Muse scheduler using this interface. There exist important differences between the BIM Prolog engine and the SICStus Prolog engine (e.g in the representation and manipulation of the Prolog stacks and the Prolog code). So, we are expecting further feedback from the BIM group which may lead to some further refinements in the interface definition.

7 Acknowledgments

This work is based on the Aurora interface as described by Péter Szeredi (from the University of Bristol) and Mats Carlsson (from the Swedish Institute of Computer Science) in [7].

The work is made possible by the funding by the PEPMA project (ESPRIT project 2471) and by the SICS sponsors. The discussions with André Mariën (from the BIM Prolog company) have also been of good help.

The sponsors of SICS are Asea Brown Boveri AB, NobelTech Systems AB, Ericsson AB, IBM Svenska AB, Televerket (Swedish Telecom), Försvarets Materielverk FMV (Defence Material Administration), and the Swedish National Board for Industrial and Technical Development (Nutek).

References

[1] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445 – 475, December 1990.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[3] Khayri A. M. Ali and Roland Karlsson. OR-parallel Speedups in a Knowledge Based System: on Muse and Aurora. In *the Proceedings of FGCS'92 (the International Conference on Fifth Generation Computer Systems*, June 1992.

[4] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. SICS Research Report R92:07, Swedish Institute of Computer Science, March 1992. Submitted to NGC Journal.

[5] Mats Carlsson and Johan Widén. SICStus Prolog User's Manual (for version 0.6). SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.

[6] Roland Karlsson. How to Build Your Own OR-parallel Prolog System. SICS Research Report R92:03, Swedish Institute of Computer Science, March 1992.

[7] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing Engines and Schedulers in OR-parallel Prolog Systems. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991.

[8] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[9] David H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.