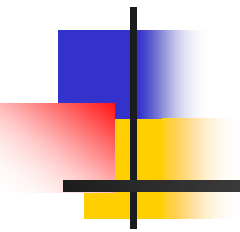


Implementation of the Soar Kernel

(Karen J Coulter)

June 17, 2005

Please refer to latest online version:

http://sitemaker.umich.edu/soar/documentation_and_links

university of michigan artificial intelligence laboratory

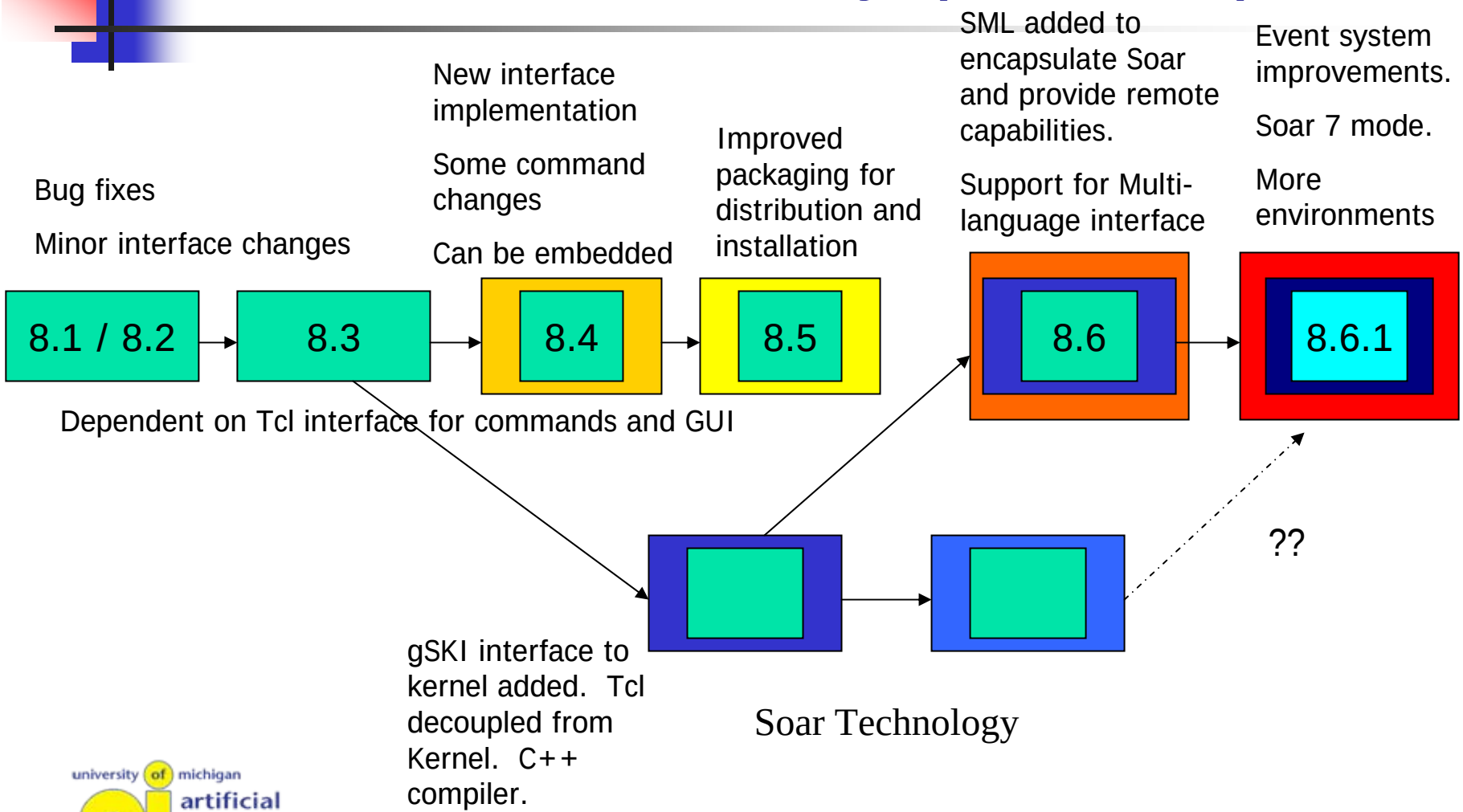
In this Talk

- Basic structure of an agent
- Agent creation and management
- What happens when “Run” an agent
- RHS functions
- What information is accessible outside kernel?
  - Data
  - Events
- High-level roadmap of kernel functionality (satellite view)
- The gSKI Agent Scheduler

NOT in this Talk

- Soar 7 differences
- Low-level Implementation details
- Special exceptions (there are MANY)
- Backtracing and Support
- Anything else I forgot, or that didn't occur to me people wanted to know

Kernel recent history (Soar 8.1)



Soar Agents are:

- Independent -- no knowledge of other agents unless explicitly supported by application, even then only in working memory. (no omniscient being)
- Maintained in linked list in kernel. Order is not guaranteed, but is consistent during execution.
- Local only. Kernel itself has no support for creating or running agents outside the local process.

Agent Management

- Soar Kernel maintains a linked list of agent pointers, agent_count and pointers to callback functions for each kernel event (more later).
- Soar source code operates on a single agent structure. Only in the scheduler, in the routines that manage the *running* of agents, does the code loop over the linked list of agent pointers.

Basic structure of an agent

- Parameters to define execution (sysparams)
- Settings for user information to be generated in “Trace” as agent executes (sysparams)
- Current state information (current-phase, decision-cycle-count, stats, etc)
- Long-term production memory (rete)
- Short-term working memory (rete)
- Internal data structures, including match set, list of wmes, and memory pools
- Struct is defined in agent.h (8.6) or soarkernel.h(8.5)

Agent Creation

- Check that name is unique within kernel
- Allocate memory for agent structure
- Initialize parameters and settings
- Create top state and IO links

  ➔ New in 8.6!

  Previously created in first Input Phase
- Optionally source file(s) to define settings and load productions (defined by application)

Loading Productions

- Productions added to Rete as they are parsed: condition by condition and then action by action
  - Not scanned for errors and then stored
- Rete may reorder conditions internally for better match efficiency
- “Documentation string” stored, but comments from file are not
- Multi-valued attributes must be declared *before* loading productions to have any impact on Rete organization

Working Memory

- Soar backtraces WMEs through their preference pointers when building chunks and calculating i-support:
  - Soar still goes thru Preference "Phase" for non-operators, but everything defaults to acceptable and parallel
  - In Propose and Apply, after preference calculations, Soar continues directly to promote preferences to Working-Memory
- WMEs that are added by `add-wme` or through SML do not have support (preferences) and need to be explicitly removed from working memory when no longer valid. (Agents can not *reject* them.)
- WMEs no longer connected to a state will be garbage-collected, including any that were added by application through SML or with `add-wme`.
- Garbage collection is automatic, but not necessarily immediate.
- More...

Input and Output Links

- Architecture automatically creates **`^io ^input-link`** and **`^output-link`** on the top state (S1) in working memory.
- WMEs exactly the same as rest of working memory.
- **`^input-link`** is a convenience for applications to organize and update sensor input to Soar agents.
- Soar tracks when changes made to **`^output-link`** so can alert applications.
- SoarKernel supports multiple output-links so can segregate multiple "actuators." Additional output-links must be created by agent or application, but Soar will automatically track when changes made just like for default **`^output-link.`** (May require some changes to gSKI and SML if using those interfaces.
- KernelSML provides simplified management of IO links

Methods for running agents: "Wrapper" functions in kernel

- **Run_forever()** runs Soar forever, until interrupted or halted.
- **Run_for_n_phases(n)** runs Soar for the specified number (n) of top-level phases.
- **Run_for_n_elaboration_cycles(n)** runs Soar for a given number (n) of elaboration cycles. In this function, Input phase, Decision phase, and Output phase are each counted as one elaboration cycle. In Propose and Apply, an elaboration cycle consists of firing all productions at one level of the goal stack
- **Run_for_n_decision_cycles(n)** runs Soar for the specified number (n) of decision cycles.
- **Run_for_n_modifications_of_output(n)** (RunTilOutput) soar runs do_one_top_level_phase ***n*** times, where n starts at 0 and is incremented anytime the output link is modified by the agent. ***n*** is not incremented when the output-link is created nor when the output-link is modified during an Input Cycle, (when getting feedback from a simulator).

For most synchronous simulation environments, usually n = 1.

Still in Kernel, but not in gSKI or CommandLineInterface since rarely used:

- **Run_for_n_selections_of_slot(n)**: runs Soar until the nth time a selection is made for a given type of slot, either state_symbol or operator_symbol.
- **Run_for_n_selections_of_slot_at_level(n)**: runs Soar for n selections of the given slot at the given level, or until the goal stack is popped so that level no longer exists.

Agent Scheduling

- Agent “scheduling” ie the queueing and running of agents in the kernel, is handled outside the “kernel” (by gSKI in Soar 8.6)
- The commandline args for running agents are parsed and passed to the scheduler, which configures agent parameters, and calls one of the kernel execution routines, such as Run_for_n_decisions.
- When multiple agents exist in the same kernel process, all agents always have the same go_type and go_number for a given run. Users can not run one agent by phases and another by decisions, or one agent for 10 phases and one agent for 2 phases, UNLESS the agents are each run one at a time with separate calls to run a single agent.

- Stop-soar: “temporarily” terminates at the end of the current phase. A new “Run” command resets stop-soar to False. Generated by RHS and external events
- agent-halted: the agent won’t run again until an init-soar clears working memory. Generated by RHS

What happens during *Run*

- Agent parameters for *type of run (go_type)* and *n* (go_number) configured by scheduler (gSKI).
- A scheduler invokes appropriate kernel wrapper function, which tests agent parameters `stop_soar` and `system_halted`, starts timers, and then calls `do_one_top_level_phase` until the requisite number of loops have been completed, or agents halts or is interrupted.
- Wrapper functions operate on one agent at a time, so if calling run_for_n_decision_cycles(5), the first agent runs for 5 decision cycles, then the next agent, then the next, til all agents have been run.
- The gSKI scheduler supports a finer grain of “interleaving” among agents when executing Run_for_n_decision_cycles or Run_for_n_modifications_of_output

For each phase:

if TRACE_PHASES_SYSPARAM is true, print out "*Phasename* starting"
invoke *BEFORE_phasename* agent callback (gSKI_EVENT_BEFORE_PHASE...)
start the phase-specific timers for the agent

execute phase-specific code, such as checking/setting parameters
    and calling phase-specific subroutines
update any counters or agent parameters
set agent->current_phase to the next phase in the cycle

turn off the phase-specific timers for the agent
invoke the *AFTER_phasename* agent callback (gSKI_EVENT_AFTER_PHASE...)
if TRACE_PHASES_SYSPARAM is true, print out "*Phasename* ending"

RHS functions in 8.6

- Tcl no longer the de-facto interface mechanism for the kernel
- Soar 8.6 “clients” interface thru SML which supports SWIG → automagically get C++, Tcl, Java, Perl, Python
- Tcl no longer a built-in RHS function, but can be added back in for any application that includes the sml_tcl_interface package
- Exec is the generic RHS function for registering and invoking custom functions on the RHS of productions.
- No other changes to default set of RHS functions.

What information is accessible outside kernel

Data is per-Agent, thru agent interface

- Agent counters and statistics, including timer values
- Settings for run-control (sysparams): learning, trace information, backtracing
- State information: current_phase, current operator, goal_stack_level
- Production memory
- Working memory

When is information accessible outside kernel: Events

- Extensive callback system thru gSKI and SML supports executing application code at almost any point in Run cycle.
- Kernel-based (once for *all* agents) and agent-based (once for *each* agent) events
  - Before and after running all agents
  - At start and end of each phase for each agent
  - Other events being added: after_all_output_phases, other "update world" locations as needed
- Also support User-defined RHS functions
- Agent information also available whenever kernel is idle

High-level roadmap of kernel functionality (satellite view)

(This is a listing of non-obvious functions and their locations...)

Initializing and running Soar, and all top-level phases defined: (see *do_one_top_level_phase()*)	init-soar.cpp
Preference resolution and operator selection: (also impasse generation and all garbage collection)	decide.cpp
Main firer routines (do_preference_phase, create_instantiation):	recmem.cpp
Support for Goal Dependency Set maintenance:	consistency.cpp
Loading and excising productions:	parser.cpp
Slot management and garbage collection:	tempmem.cpp
Format Soar's output:	trace.cpp

More info in source and header files, some in Soar FAQ. Document in progress.

Soar's Memory Management

- "Memory pools" malloc'd from operating system in (32K) blocks
- Soar frequently allocates and deallocates within pools, but only when exceeds current pool size does Soar malloc more blocks for pool
- Internal memory mgmt scheme can hide details from profilers and other code tools → not always a good thing.
- Applications can choose to have Soar use straight malloc instead of pools, but performance will be drastically affected.
- See allocate_with_pool in mem.h, but be VERY careful if ever change it. (not recommended)

