An Integrated Development Environment and Architecture for Soar-Based Agents

Ari Yakir and **Gal Kaminka**
The MAVERICK Group
Computer Science Department
Bar Ilan University, Israel
{yakira,galk}@cs.biu.ac.il

Abstract

It is well known how challenging is the task of coding complex agents for virtual environments. This difficulty in developing and maintaining complex agents has been plaguing commercial applications of advanced agent technology in virtual environments. In this paper, we discuss development of a commercial-grade integrated development environment (IDE) and agent architecture for simulation and training in a high-fidelity virtual environment. Specifically, we focus on two key areas of contribution. First, we discuss the addition of an explicit recipe mechanism to Soar, allowing reflection. Second, we discuss the development and usage of an IDE for building agents using our architecture; the approach we take is to tightly-couple the IDE to the architecture. The result is a *complete* development and deployment environment for agents situated in a complex dynamic virtual world.

Introduction

It is well known how challenging is the task of coding complex agents for virtual environments. This has been a topic for research in many papers including (Bordini *et al.* 2006; Tambe *et al.* 1995; Jones *et al.* 1999; D.Vu *et al.* 2003). This difficulty in developing and maintaining complex agents has made adoption of cognitive architectures difficult in commercial applications of virtual environments. Thus many companies work with different variations of state machines to generate behaviors (Calder *et al.* 1993).

In this paper, we discuss development of a commercial-grade development environment and agent architecture for simulation and training in a high-fidelity virtual environment. We discuss architectural support for coding of a complex plan execution by a team of agents, in Soar, and discuss the differences in our approach from previous approaches to using Soar in such tasks.

Specifically, we focus on two key areas of contribution. First, we discuss the addition of an explicit recipe mechanism to Soar, allowing reflection. This allows a programmer to build Soar operators (units of behavior) that are highly reusable, and can reason about their selection and deselection. We show how this mechanism acts as a decision-kernel allowing multiple selection mechanisms (simulating human social choices, domain knowledge, etc.) to all coexist on top of it. The recipe mechanism generates possible alternatives: The choice mechanisms assign preferences to these. Soar then decides.

Second, we discuss the development and usage of an integrated development environment (IDE) to build agents using our architecture. The approach we take is to tightly-couple the architecture to the development environment, so that bugs—which in Soar can be notoriously difficult to find (Ritter *et al.* 2005)—can be ironed out as they are written.

We demonstrate these efforts in a *complete* development environment for Soar agents, situated in a complex dynamic virtual world, used for realistic simulation and training. We attempt to draw lessons learned, and highlight design choices which we feel were important from the perspective of an industrial project.

Background

Our work was done as part of Bar Ilan University's collaboration with Elbit Systems, Ltd. The goal is to create a *smart* synthetic entity—an agent—which performs in a variety of simulated scenarios. Agents should operate autonomously, behaving as realistically as possible. The agents will enhance Elbit's training and simulation products.

The environments in which the agents are to function are usually complex environments, containing up to entire cities, and including accurate placement of objects. The initial focus of the project is towards the development of individual entities, possibly working in small groups. Figure 1 shows an example screen-shot from an application use-case.

Both the architecture and IDE for the agents must be oriented towards the development of configurable entities, driven by capabilities, personality and complex plans. Such a view reinforces the need for a flexible architecture, able to cope with many parameters and configurations of large plans (composed of recipes with 100 up to 1000 inner behavior nodes). The architecture must support several distinct cognitive mechanisms (emotions, focus of attention, memory, etc.) running in parallel and interacting, in each and every virtual modeled cognitive entity.

We briefly introduce here the various components of our architecture, and the rationale behind its design. The next sections will discuss the foci of the paper in depth.

One main difference between commercial and academic

Figure 1: **Urban terrain**

frameworks for multi-agent systems, is in the use of hybrid architectures. While in most academic work it is sometimes possible—indeed, desired—to include all levels of control using a unified representation or mechanism, this is clearly not the case when it comes to large scale industrial applications. No single architecture or technology in this case is sufficient. Moreover, it is often critical to be able to interact with existing underlying components. This might come as a demand from the customer who ordered the project, or (sometimes) as a way to promote other technology available within the company.

With respect to academic work, this view goes back to past research on agent architectures, such as the ATLANTIS (Gat 1992) architecture, which is based on the observation that there are different rates of activity in the environment, requiring different technologies. In our work, we were inspired as well by the vast research and conclusions drawn from the RoboCup simulation league (Marsella *et al.* 1999) and from past simulation projects conducted in Soar such as the IFOR project (Tambe *et al.* 1995; Jones *et al.* 1999).

Indeed, our industrial partners have developed a hybrid architecture in which many components that have to do with cognitive or mental attitudes are actually outside of the main reasoning engine, built in Soar. The guiding philosophy in deciding whether something should be done in the Soar component has been to leave (as much as possible) any and all mathematical computations outside of Soar, including all path planning and motion control. For example, we rely on a controller in charge of moving an agent on a specified path. Such controller can be assigned the movement of teams of agents, and can use different movement configurations while trying to keep relations and angles fixed between it's members. In making this choice, the project is setting itself apart from other similar projects, in which Soar was used to control entities at a much more detailed level of control (Tambe *et al.* 1995; Jones *et al.* 1999; Marsella *et al.* 1999).

We focus in this paper on the Soar decision-making component, and its associated IDE. Both of these, with the other components of the system, are hooked up to a VR-Forces (MÄK Technologies 2006) simulation environment, a high-fidelity simulator utilizing DIS. It is used for large scale projects ranging air, ground and naval training such as TACOP (van Doesburg, Heuvelink, & van den Broek 2005).

Given the task of providing an agent development framework, several architectures for this type of application might come to mind: JACK (Howden *et al.* 2001), SOAR (Newell 1990), UMPRS (Lee *et al.* 1994), JAM (Huber 1999), etc. Soar (Newell 1990) is among the few that has commercial support, and yet is open-source, making it a clear favorable candidate for our project.

Soar uses globally-accessible working memory, and production rules to test and modify it. Efficient algorithms maintain the working memory in face of changes to specific propositions. Soar operates in several phases, one of which is a decision phase in which all relevant knowledge is brought to bear, through an XML layer, to make a selection of an operator (behavior) that will then carry out deliberate mental (and sometimes physical) actions.

A key novelty in Soar is that it automatically recognizes situations in which this decision-phases is stumped, either because no operator is available for selection (*state no-change impasse*), or because conflicting alternatives are proposed (*operator tie impasse*). When impasses are detected, a subgoal is automatically created to resolve it. Results of this decision process can be chunked for future reference, through Soar's integrated learning capabilities. Over the years, the impasse-mechanism was shown to be very general, in that domain-independent problem-solving strategies could be brought to bear for resolving impasses (Newell 1990).

Being a mixture between a reactive and a deliberative system, it is usually very easy to program rules (productions) in Soar, so that a short sequence will be triggered upon certain conditions. However, building a complex scenario involving multiple agents becomes somewhat of an overwhelming task. Debugging just seems to never end[1].

Soar uses globally-accessible working memory. Each rule is composed by a left and right sides. Simplified, the left side of the rule is in charge of testing whether specific conditions hold in this working memory, while the right side is in charge making changes to the working memory. Thus each rule in the system can read, write, and modify the working memory, triggering or disabling the proposal of other rules, including itself. This means that each Soar programmer must have complete knowledge of all the rules, taking all previous written code into account each time a new rule is added.

Another facet is that Soar does not differentiate between the change an operator makes, and the actual state of the agent, and ties them as one by coding conventions. Since Soar operates through states, this means that each operator by definition is tied to the state the agent is in. In other words, naive Soar programming requires all agent behaviors to be re-programmed each time a behavior is to be applied

[1]We note that similar motivations have lead in the past to contributions in other directions, e.g., teamwork (Tambe 1997).

in a slightly different state than initially anticipated by the programmer.

One of the first architectural changes we aimed for was to overcome this relation between states and operators. By doing so, we could make use of generic types, templates, and other byproducts such as the utilization of reflection. These proved to be valuable programming tools.

Soaring Higher

The approach we take is to provide a higher level of programming, built on Soar foundations and taking advantage of the underlying framework. The most important component of this layer is *recipes*—behavior graphs—representing a template (skeletal) plan of execution of hierarchical behaviors (Kaminka & Frenkel 2005; Tambe 1997). The behavior graph is an augmented connected graph tuple (B, S, V, b_0) , where B is a set of task-achieving behaviors (as vertices), S, V sets of directed edges between behaviors ($S \cap V = \emptyset$), and $b_0 \in B$ a behavior in which execution begins.

Behaviors is defined as $b_i \in B$:

1. Constant parameters, with respect to the program execution scope (such as b_i timeout , probability etc..).
2. Dynamic parameters, with respect to b_i execution scope (such as the event that triggered b_i preconditions).
3. Maintenance conditions (Kaminka *et al.* 2007), with respect to b_i execution scope.
4. Teamwork conditions (Kaminka *et al.* 2007), with respect to b_i execution scope.
5. Preconditions which enable its selection (the robot can select between enabled behaviors).
6. Endconditions that determine when its execution must be stopped.
7. Application rules that determine what b_i should do upon execution.

In (Kaminka & Frenkel 2005) S sequential edges specify temporal order of execution of behaviors. A sequential edge from b_1 to b_2 specifies that b_1 must executed before executing b_2. A path along sequential edges, i.e., a valid sequence of behaviors, is called an *execution chain*. V is a set of vertical *task-decomposition* edges, which allow a single higher-level behavior to be broken down into execution chains containing multiple lower-level behaviors. At any given moment, the agent executes a complete path root-to-leaf through the behavior graph. Sequential edges may form circles, but vertical edges cannot. Thus behaviors can be repeated by choice, but cannot be their own ancestors.

Even using this representation, we faced several abnormal situations. For example, if a leaf behavior has precondition equal to its ancestors endcondition it might never be proposed, or worst, constantly be terminated prematurely. Solving such a problem at an IDE level, contradicts the need for behavior encapsulation. Another problematic aspect of such an architecture is that during an execution chain no alternatives are being considered. Switching from one execution chain to the other (given that they both derive from the same parent behavior), needs ending the whole execution chain, a process which is both time consuming, and sometimes harms the overall reactiveness of the system. This problem emerges even when using the Soar architecture as provided. We will not deal with proposed solutions since they are out of this paper's scope. However, one specific proposal involving a reactive recipe mechanism running on top of the regular one, can be viewed as a higher level selection mechanism, and thus is similar to other selection mechanism discussed later in detail.

The recipe mechanism is responsible for proposing operators for selection. Through reflection, it examines the current recipe data structure (graph), and proposes all operators that are currently selectable, based on their precondition and position within the recipe graph. It efficiently schedule the proposal and retraction of generic behaviors given certain conditions. These behaviors, are specified inside generic subtrees of plans, which in turn are gathered in large abstract sets of plans. When a Soar agent is loaded, it assembles its recipe structure at run-time by recursively deepening, arranging and optimizing it.

Additional mechanisms are added to guide selection between the proposed operators. Examples to such mechanisms include probabilistic behavior selection, teamwork, social comparison theory (Kaminka & Fridman 2007), individual and collaborative condition maintenance (Kaminka *et al.* 2007), etc. For example, since the recipe enables reflection, one of the mechanisms monitors other agents' actions by the mirroring of the recipe onto another inner Soar state and using translation of sensory data. This allows modeling of another agent's decision processes based on observation—a form of plan recognition. Another mechanism is in charge of teamwork and keeps the team synchronized and roles allocated, by the use of communication (Kaminka *et al.* 2007).

With respect to the IDE, we made use of a new state-of-the-art facilities such as refactoring and testing of agent applications. Instead of building the IDE from scratch, as is commonly done, we chose to utilize an existing IDE, thus taking advantage of well-tested available technology. Our IDE is object-oriented, facilitating coding by the use of premade templates, re-usability of components such as plans and behaviors, instead of wizards and graphical means of programming.

In Soar, productions are proposed due to changes in WMEs (Soar working memory). In a behavioral context, this means that each behavior can be triggered by a change, both internal (inner state change) or external (sensory data), and that each behavior can affect the overall conduction of the system. During early phases of development we chose an approach similar to that found in (Tambe 1997), by providing a middle layer between Soar inputs and operators. However, as mentioned, Soar's productions can be triggered by internal events as well. Thus, we chose to broaden the common ground between behaviors by substituting the translation layer with an event-based mechanism. All our behaviors' preconditions and endconditions are triggered by explicit predicates, which signal events that are true. These events correspond to percepts, deduced or processed facts, and internal changes. They constitute explicit facts, internally classified by subject and category (e.g., all audio-

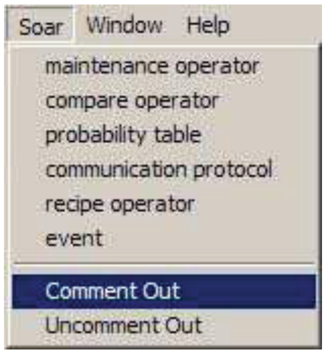


Figure 2: **Soar integrated templates**

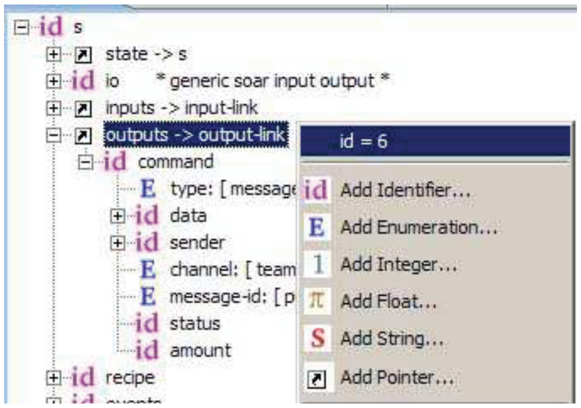
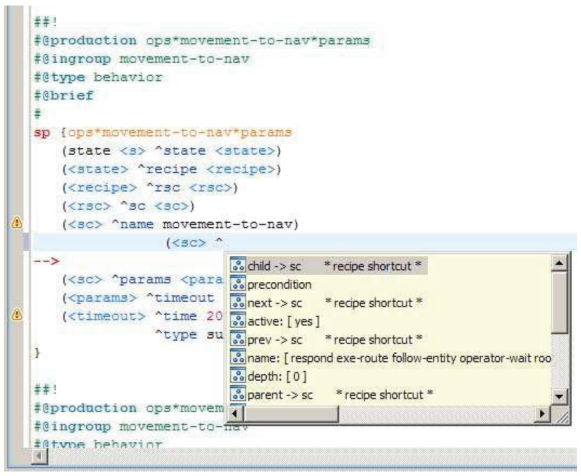


Figure 3: **Soar Datamap view**



```
##!
#@production ops*movement-to-nav*params
#@ingroup movement-to-nav
#@type behavior
#@brief
#
sp {ops*movement-to-nav*params
   (state <s> ^state <state>)
   (<state> ^recipe <recipe>)
   (<recipe> ^rsc <rsc>)
   (<rsc> ^sc <sc>)
   (<sc> ^name movement-to-nav)
              (<sc> ^
-->
   (<sc> ^params <para
   (<params> ^timeout
   (<timeout> ^time 20
             ^type su
}

##!
#@production ops*movem
#@ingroup movement-to-nav
#@type behavior
```

```
child -> sc     * recipe shortcut *
precondition
next -> sc      * recipe shortcut *
active: [ yes ]
prev -> sc      * recipe shortcut *
name: [ respond exe-route follow-entity operator-wait roo
depth: [ 0 ]
parent -> sc    * recipe shortcut *
```

Figure 4: **Auto complete with deep inspection**

related events groups together).

Adding events to Soar allows our agent means of reflection. A regular Soar agent is unaware of the actual change in the environment that lead to a specific operator instantiation, thus could not refer to the cause of it following a specific sequence of actions. At most, it can reflect on the actions themselves. Using the event mechanism, however, allows the agent to consider exactly what changes lead to each operator/behavior proposal or termination, as the preconditions and endconditions are defined explicitly.

We found this approach critical when in need of communications between agents. The language by which our agents communicate is an event language: Entire subtrees of Soar working memory are being passed on and forth between agents. The agents thus pass between them sets of events relevant to the proposal or retraction of behaviors. This allows allocation of roles, and synchronization of the execution of behaviors. and

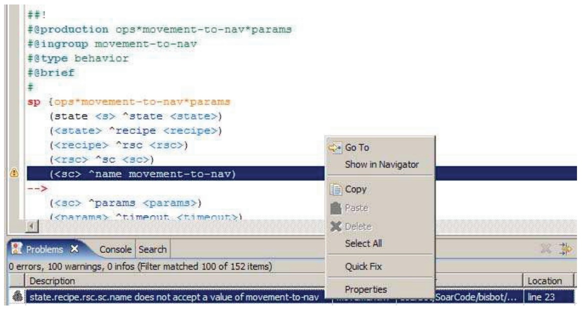
In our target environment, both recipe operators (task and maintenance) and events can be programmed with the help of code templates. During the coding phase we discovered that most bugs result from WME misspelling or errors in structure reference. Figure 2 shows the interface by which a user can automatically generate the appropriate operator or event code. Events are generated and categorized in different folders, classified by the inputs that trigger them or by the events that they relay on. Operators (behaviors) are generated with parameters, preconditions and endconditions, fully documented. This feature results in a clean uniform code, and thus simplifies debugging a great deal. Additional support for communication protocols and probability tables for operator proposals is also provided.

The use of templates in Soar, goes back to the early IDE development tools for Soar agents. Our tool differs from those earlier works in that it provides not only basic support for Soar operator application and proposal rules templates, but an extensive elaborated behavior structure supported by the recipe mechanism, specialized for the architecture we use. The use of the templates saves much coding for the programmer, since they already encapsulate much of what the programmer needs to consider.

The Soar datamap is a representation of the Soar memory structure generated through the execution of a Soar program, and can be inferred by the left side and right side of Soar production rules. Several tools are available in order to generate a Soar datamaps through static analysis of Soar productions. We significantly extended the initial Eclipse extension for datamap support, provided by the University of Michigan and SoarTech, adding additional services and tools. Most of our coding tools now rely on the datamap, enabling us to generate specific insightful warnings, provide smart assistance, and auto completion of code that takes the structure of the memory in our architecture into account.

By using elaborations on the datamap structure provided, and by constantly matching it with the code being edited, the IDE is able to propose completion of relevant points in the code. As seen in Figure 4, the IDE is able to propose the optional suggestions for code completion down the WME path `s.state.recipe.rsc.sc}`, where recipe behaviors are kept. By inspecting the datamap it is then able to provide insights regarding it structure, such as its preconditions, endconditions, etc.

Using the Soar parser combined with datamap inspection we are able to assist the programmer with warning messages (as seen in Figure 5) and quick-fixes (as seen in Figure 6). In Figure 5 a common situation is demonstrated,

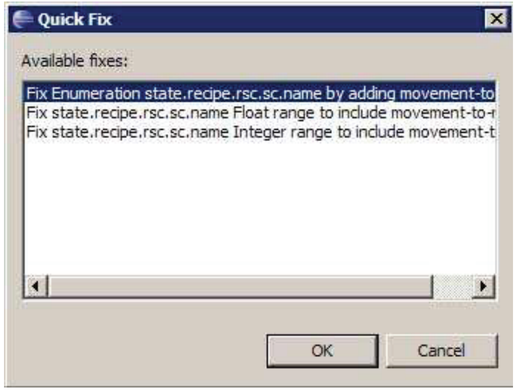


```
##!
#@production ops*movement-to-nav*params
#@ingroup movement-to-nav
#@type behavior
#@brief
#
sp {ops*movement-to-nav*params
   (state <s> ^state <state>)
   (<state> ^recipe <recipe>)
   (<recipe> ^rsc <rsc>)
   (<rsc> ^sc <sc>)
   (<sc> ^name movement-to-nav)
-->
   (<sc> ^params <params>)
   (<params> ^timeout <timeout>)
```

0 errors, 100 warnings, 0 infos (Filter matched 100 of 152 items)

Description	Location
state.recipe.rsc.sc.name does not accept a value of movement-to-nav … SoarCode/bisbot/…	line 23

Figure 5: **Warnings**



Available fixes:

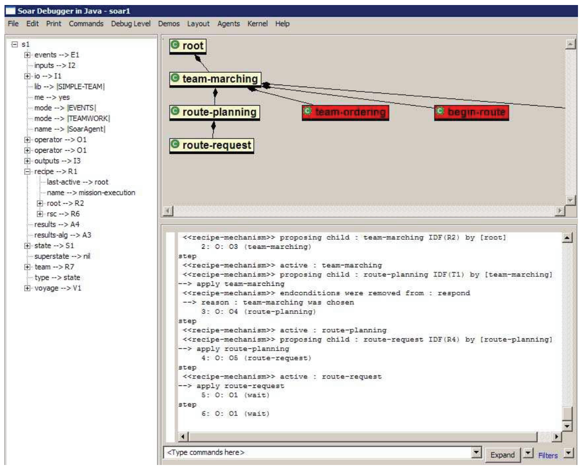
Fix Enumeration state.recipe.rsc.sc.name by adding movement-to
Fix state.recipe.rsc.sc.name Float range to include movement-to-r
Fix state.recipe.rsc.sc.name Integer range to include movement-t

Figure 6: **Quick Fixes**

where the Soar code refers to an unknown behavior called `movement-to-nav}`, which judging by its name, might be the behavior in charge of moving the agent to a specific location. This warning message notifies the programmer that either: (a) this behavior does not exist, which leads her to the understanding that it is yet to be programmed; or (b) there is a behavior already present in charge of moving the agent to a specific location but it is not called `movement-to-nav}`; or (c) There is a behavior already present in charge of moving the agent to a specific location called `movement-to-nav}`, which is not updated in the Soar datamap, thus no one knows it exist. Such failure points are easily spotted and corrected by the use of quick-fixes that offer several optional automatic corrections to the datamap.

Aside from warnings and code assistance, Soar benefits from the many Eclipse plug-ins that are already present and developed within the IDE environment. Among those are integrated documentation support, execution of Soar agents, and integrated debugger. Support for both VSS and CVS code-versioning systems can be found as well, for large team projects.

We have also extended and enhanced the SoarJavaDebugger which is distributed with the current version of Soar, by the University of Michigan. The first customization, seen in Figure 7, utilizes a UML type of visualization, in order to display the recipe at run time. At any point the active path to the currently executed behavior is presented along with optional behaviors not chosen (colored red). These red behaviors have matching preconditions, but were not activated due to hierarchical or situational constraints. This recipe visualization is updated as well at runtime, enabling the programmer to focus only on the relevant executed subset of the recipe.



```
s1
  events --> E1
  inputs --> I2
  io --> I1
  lib --> |SIMPLE-TEAM|
  me --> yes
  mode --> |EVENTS|
  mode --> |TEAMWORK|
  name --> |SoarAgent|
  operator --> O1
  operator --> O1
  outputs --> I3
  recipe --> R1
    last-active --> root
    name --> mission-execution
    root --> R2
    rsc --> R6
  results --> A4
  results-alg --> A3
  state --> S1
  superstate --> nil
  team --> R7
  type --> state
  voyage --> V1
```

```
 <<recipe-mechanism>> proposing child : team-marching IDF(R2) by [root]
     2: O: O3 (team-marching)
step
 <<recipe-mechanism>> active : team-marching
 <<recipe-mechanism>> proposing child : route-planning IDF(T1) by [team-marching]
--> apply team-marching
 <<recipe-mechanism>> endconditions were removed from : respond
 --> reason : team-marching was chosen
     3: O: O4 (route-planning)
step
 <<recipe-mechanism>> active : route-planning
 <<recipe-mechanism>> proposing child : route-request IDF(R4) by [route-planning]
--> apply route-planning
     4: O: O5 (route-request)
step
 <<recipe-mechanism>> active : route-request
--> apply route-request
     5: O: O1 (wait)
step
     6: O: O1 (wait)
```

Figure 7: **Soar Java Debugger, with additional Tree View and Recipe Visualization**

In addition, on the left-hand side of the debugger window, is a tree-folder view of the working memory. The root of the tree can be set to point any subset of the agent knowledge (any Working Memory Element, WMEs) and is updated at runtime. Since Soar already arranges WMEs in a tree like format, it greatly speeds up debugging to be able to inspect the agent knowledge by simply clicking such folders.

Evaluation

Evaluation of the contributions described above is challenging. The first contribution involves the use of reflection in the recipe, which allows clean separation of the process by which the knowledge of the agent proposes alternatives, and the mechanisms that facilitate a decision among them. During the evaluation of our system, we made use of a scenario in which a team of agents uses communications to agree upon several mission points. They calculate routes considering possible threats along the way and travel from one location to the other. While doing so they collaboratively maintain several movement protocols and react to changes in the environment such as the appearance of new threats, the loss of team members, etc.. During the execution of the scenario, the agents move from one waypoint to another, maintaining specified formations, and reorganize these formations given changes in the team hierarchy.

To provide some insight as to the performance of the design, we compare our system to previous systems that have utilized Soar as their basis. The most well-known similar system is TacAir-Soar, a highly successful project using Soar as the basis for synthetic pilots, capable of running a wide variety of missions (Tambe *et al.* 1995; Jones *et al.* 1999). Less complex—yet still successful—applications of Soar included the ISIS-97 and ISIS-98 RoboCup teams (Tambe *et al.* 1999).

Table 1 provides a comparison of key features, allowing a qualitative insight into the complexity of these systems, compared to the system discussed in this paper. The columns report (left-to-right) on the overall number of Soar

$architecture$	$rules$	$actions$	$inputs$	$operators$
TacAir-Soar	5200	30	200	450
ISIS-97/98	1000	7	50	40
Ours	650	25	200	100

Table 1: **Architectural Complexity Evaluation**

$benchmark$	dc	$msec/dc$	$WM(mean, changes)$
MaC	200	0.155	(49.896,13651)
Arithmetic	41487	0.320	(983.589,879076)
Ours	31363	0.078	(3266.797,196263)

Table 2: **Runtime Evaluation**

rules used in the system, the number of unique actions (outputs), the amount of unique percepts (inputs), and the number of actual domain/task behaviors/operators.

Our system, at its current state of development, is of moderate complexity compared to efforts that have been reported in the literature. Taking the combined inputs and outputs as the a basic measure of the complexity of the task, would put our system's task on par with that of the TacAir-Soar system, and far ahead of the challenge faced by RoboCup teams. However, looking at the number of operators, we see that the knowledge of our agents, while still significantly more complex than that of the RoboCup agents, is still very much behind that of the advanced TacAir Soar.

Based on this qualitative assessment, which puts our system somewhere in the middle between the TacAir-Soar and the ISIS systems, it is interesting to note that our system uses significantly less rules than *both* other systems, to encode the knowledge of the agents. While we use about 6.5 rules, on average, for supporting each operator, TacAir-Soar uses 11.5 and RoboCup about 25. We believe that this is due, at least in part, to the use of the recipe mechanism. In both previous systems, the preconditions of operators tested not only the appropriateness of an operator given the mental attitude of the agents with respect to its environment and goals, but also with respect to the position of the operator compared to other task operators. For instance, commonly operators would have to test for the activation of their parents, before being proposed. The recipe mechanism cleanly separates the two.

On our system, operator rules only determine whether the task-related preconditions of an operator have been satisfied. The rules proposing the operator if its preconditions are true, *and given its position within the behavior graph*, are all part of the recipe mechanism. We believe that this saves a significant number of rules.

It also saves significant programming effort: Since our operators do not refer at any point to their execution point, changing the occurrence of a generic action (or a generic subtree of hierarchical actions within a recipe) requires only updating the configuration of the Soar coded recipe. In comparison, moving operators around in previous systems, from one specific execution point to another (one point in the recipe to the other) would require changes to be made in all branching children (all rules testing the occurrence of such an operator), since the hierarchy is part of each sub-operator's preconditions. Additionally, by previous Soar conventions, operator source files were written in hierarchical file system, which reflected the intended hierarchical decompositions. Moving operators in the recipe either caused files to move around, or worse yet, created a discrepancy between the convention of the file-system and the position of the operator in memory. Freeing Soar operators from their execution point also allowed us to place all operator files in a single directory, making finding and maintaining them much easier.

Previous Soar architectures, have utilized a specific style of writing in Soar, in which hierarchical decompositions are created in memory by relying on Soar's *operator no-change* impasse to keep track of the active hierarchical decomposition. But the creation and maintenance of impasses can be expensive. The recipe mechanism allows us efficient book-keeping of the current decomposition, without using impasses (unless needed for other reasons).

To demonstrate the savings offered by using the recipe mechanism, Table 2 provides data gathered from the execution of several standard Soar benchmarks (bundled with the Soar architecture distribution), on the same hardware and software configuration (Soar 8.6.2 kernel on the same Pentium 4 CPU 3.2 GHz 512MB ram). These standard problems consisted of the Missionaries and Cannibals (MaC) problem, and the performance of 1000 random arithmetic calculations. We compare Soar's performance in both, with the test scenario, described above.

Table 2 consists of four columns: The number of decision-cycles in Soar (input to output phase) using an average run, the average time for each decision-cycle in milliseconds, the average size of Soar working memory at any time, and the number of changes to this memory. As shown, our architecture is much faster than the benchmarks—despite their simplicity relative to the task our system faces. Our decision-cycles are substantially faster mainly due to the recipe mechanism (which avoids impasses) and the utilization of controllers. Though new input is constantly delivered to our agents, most of the time our agent is idle, waiting for the current operator/behavior execution, the proposal of new behaviors or the arrival of critical data. Such results are crucial for demonstrating the scalability of the system, for future scenarios (e.g., those simulating crowds).

We now turn to evaluation of the integrated development environments. As one could expect, quantitative evaluation is difficult here. Not only is the impact of the changes difficult to measure directly, but the target audience—Soar programmers—is very small. Nevertheless, we asked our current users to provide qualitative feedback on the tool, and compare it to previously-published development tools for Soar (such as Visual Soar, which is packaged with the Soar distribution).

Our users varied in experience, and in responses. One veteran Soar programmer has previously developed in Soar using emacs text-editor (without any GUI support for debugging), and later in Visual Soar. His assessment was that the use of the Eclipse environment was a marked improvement over Visual Soar (which, not surprisingly, was believed to be a significant improvement over emacs). He reported that the use of templates was not a speed-saver: As a veteran Soar programmer, he was used to writing code directly, without templates. On the other hand, two relatively novice programmers now swear by the Eclipse environment, and show strong preference to it over existing tools. They report that the templates are very useful, though they lose some of the usefulness over time. Based on these qualitative reports, it is clear that in an industrial project, a strong IDE such as Eclipse, is a valuable tool which provides many benefits in comparison to the alternatives.

Conclusion

In this paper we discussed both the architecture and development environment for computed generated forces, based on an extended version of Soar. On an architectural level we proposed the addition of an explicit recipe mechanism to Soar, allowing reflection. This allows a programmer to build Soar operators (units of behavior) that are highly reusable and effective. We proposed how such a mechanism could act as a decision-making kernel by implementing multiple selection mechanisms on top of it. Second, we discussed the development and usage of an integrated development environment (IDE) to build agents using our architecture. We attempted to draw lessons learned, and highlight design choices which we felt were important from the perspective of an industrial project. We believe those insights can contribute towards the future development of computer-generated forces, in complex dynamic virtual worlds.

Acknowledgments. We thank our Elbit Systems partners for many useful discussions and feedback: Ora Arbel, Itay Guy, Ilana Segall, Myriam Flohr, and Erez Nachmani. The work was supported in part by a generous gift by MÄK Technologies, and by BSF grant #2002401. As always, thanks to K. Ushi.

References

Bordini, R.; Braubach, L.; Dastani, M.; Seghrouchni, A. E. F.; Gomez-Sanz, J.; Leite, J.; O'Hare, G.; Pokahr, A.; and Ricci, A. 2006. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, 33–44.

Calder, R. B.; Smith, J. E.; Courtemanche, A. J.; Mar, J. M. F.; and Ceranowicz, A. Z. 1993. Modsaf behavior simulation and control. In *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*. Orlando, Florida: Institute for Simulation and Training, University of Central Florida.

D.Vu, T.; Go, J.; Kaminka, G. A.; Veloso, M. M.; and Browning, B. 2003. MONAD: A flexible architecture for multi-agent control. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-03)*.

Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*. Menlo Park, Calif.: AAAI press.

Howden, N.; Rönnquist, R.; Hodgson, A.; and Lucas, A. 2001. JACK: Summary of an agent infrastructure. In *Proceedings of the Agents-2001 workshop on Infrastructure for Scalable Multi-Agent Systems*.

Huber, M. J. 1999. JAM: A BDI–theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, 236–243.

Jones, R. M.; Laird, J. E.; E., N. P.; Coulter, K.; Kenny, P.; and Koss, F. 1999. Automated intelligent pilots for combat flight simulation. *AI Magazine* 20(1):27–42.

Kaminka, G. A., and Frenkel, I. 2005. Flexible teamwork in behavior-based robots. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*.

Kaminka, G. A., and Fridman, N. 2007. Social comparison in crowds: A short report. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*.

Kaminka, G. A.; Yakir, A.; Erusalimchik, D.; and Cohen-Nov, N. 2007. Towards collaborative task and team maintenance. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*.

Lee, J.; Huber, M. J.; Durfee, E. H.; and Kenny, P. G. 1994. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the AIAA/NASA Conference on Intelligent Robotics in Field, Factory, Service, and Space*, 842–849.

MÄK Technologies. 2006. VR-Forces. http://www.mak.com/vrforces.htm.

Marsella, S. C.; Adibi, J.; Al-Onaizan, Y.; Kaminka, G. A.; Muslea, I.; Tallis, M.; and Tambe, M. 1999. On being a teammate: Experiences acquired in the design of RoboCup teams. In *Proceedings of the Third International Conference on Autonomous Agents (Agents-99)*, 221–227. Seattle, WA: ACM Press.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, Massachusetts: Harvard University Press.

Ritter; E., F.; Morgan, G. P.; Stevenson; E., W.; and Cohen, M. A. 2005. A tutorial on Herbal: A high-level language and development environment based on protégé for developing cognitive models in Soar. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation*.

Tambe, M.; Johnson, W. L.; Jones, R.; Koss, F.; Laird, J. E.; Rosenbloom, P. S.; and Schwamb, K. 1995. Intelligent agents for interactive simulation environments. *AI Magazine* 16(1).

Tambe, M.; Kaminka, G. A.; Marsella, S. C.; Muslea, I.; and Raines, T. 1999. Two fielded teams and two experts: A RoboCup challenge response from the trenches. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, volume 1, 276–281.

Tambe, M. 1997. Towards flexible teamwork. *Journal of Artificial Intelligence Research* 7:83–124.

van Doesburg, W. A.; Heuvelink, A.; and van den Broek, E. L. 2005. Tacop: a cognitive agent for a naval training simulation environment. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-05)*, 34–41. New York, NY, USA: ACM Press.