Knowledge Engineering through Simulation

Stefan Edelkamp
Computer Science Department
University of Dortmund

Jeremy Frank
Planning and Scheduling Group
NASA Ames Research Center

Mark Kellershoff
Computer Science Department
University of Dortmund

Abstract

The objectives of the International Knowledge Engineering Competition for Planning and Scheduling are to judge tools for knowledge acquisition and domain modeling, to accelerate knowledge engineering research in AI, and to encourage the development of software platforms that promise more rapid, accessible, and effective ways to construct reliable and efficient systems.

The 2nd edition for the competition aimed at quantitative instead of only qualitative results. The evaluation infrastructure accomplishes this in three ways: 1) standardizing the application domains on which competitors used their KE tools, 2) representing those domains using simulations and documentation describing the domains, and 3) logging records of tool interactions. Client-server communication for the exchange of simulator states and computed plans was realized via a textual protocol.

In this document, we give an overview on general concepts of the competition its impact as long-term challenge for the community, and a description of the client-server infrastructure that we have installed.

Introduction

Knowledge Engineering (KE) for AI Planning and Scheduling has been defined as *the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimization of appropriate planning machinery to work on them* (Bartak & McCluskey 2006). Hence, knowledge engineering processes support the planning process: they comprise all of the offline, knowledge-based aspects of planning that are to do with the application being built, and any on-line processes that cause changes in the planner's domain model.

The 2nd edition of the International Competition on Knowledge Engineering for Planning and Scheduling, ICKEPS-2 for short, hosted at the International Conference on Automated Planning and Scheduling, was intended to provide a continuation of knowledge-based and domain modeling as a bi-annual event, in synergy to the bi-annual International Planning Competition IPC (McDermott 2000; Bacchus 2001; Long & Fox 2003; Hoffmann & Edelkamp 2005) The objectives of the competition were to accelerate knowledge engineering research in AI, and to encourage the development of software platforms that promise more rapid, accessible, and effective ways to construct reliable and efficient systems.

The major weakness of ICKEPS-1 (Bartak & McCluskey 2006) was that the competition outcome was both subjective and qualitative, for a variety of reasons. The previous competition featured tools with different strengths, which were demonstrated on different planning domains, and for which the final rankings were based solely on the authors' demonstration of the tools to the committee. While the tools were by and large publicly available, there was no quantitative assessment of the tools, and thus no record or metrics by which tool performance could be measured or displayed, in contrast to the International Planning Competitions or other computational challenges.

As a consequence, in ICKEPS-2 we extended and revised the event in one important aspect: the quantitative evaluation of systems in a client-server simulation environment. On the one hand, we installed medium and high fidelity planning domain simulators that were provided by research or industrial entities. These simulators were adapted to communicate via files in order to provide simulator states, execute (partial) plans and feedback simulation results. On the other hand, for the competitors we provided simulator specification and some planning task descriptions in the form of natural language documents. The challenge was to come up with a planning model in an integrated tool environment that could operate the simulator by providing plans to achieve the desired outcome described in the documentation. The choice of planning infrastructure and design tools, including the plan domain description language, was up to the competitors. The only requirement was to satisfy the textual interface description.

In this document we describe the conceptional and infrastructural basics that were chosen for running the competition. The text is structured as follows. First, we introduce the knowledge engineering scenario that we imposed. Next we give a motivating example of a possible simulator domain together with a matching domain model as a solution. The consequences on the client-server software infrastructure that had to be provided for running the competition are explained next. We describe the functionality of the ICKEPS Simulation Server in detail. Finally, we draw conclu-

sions and discuss the challenges that we have imposed with the knowledge engineering competition scenario, and future avenues for the continuation of the event.

Scenario

The knowledge engineering scenario that we have imposed is that of applying planning technology in a real-world application (see Figure 1). We assume a "client" has a system they would like controlled with a planning and scheduling application. In such circumstances, it is natural that there is a simulation of the environment that can execute a given plan for the system that is posed in some formal specification language. For the sake of simplicity, we assume *full observability*, i.e. all relevant state information in the simulator is accessible and can be communicated to the application. System control can be posed as a "classical" or "off-line" planning problem consisting of an initial state and desirable goal; the application must then generate a single plan for execution. However, system control can also be posed as a "dynamic" or "on-line" planning problem, where the application may be periodically required to revise an existing plan in the face of new information.

The competitors are given a detailed description of the simulator domain and a natural language description of the types of planning problem instances to be solved. In the usual case this text description will contain part of all of a domain ontology (e.g. a type hierarchy), the full set of state variables, and accepted actions declaration that can appear in a possible plan and that can drive the simulation.

Models early in the application development process are likely to be rather coarse abstractions of the system. Generated plans may fail based on two reasons. One is that the model of the simulated process is inaccurate such that the simulator runs into a state where the operators that are scheduled next can no longer be executed. For this case simulator returns simply returns *false*. Otherwise it returns with *true* and the current system state. The other source of failure are environmental changes to the model, that make some operator applications impossible. Such changes would require the planner to re-plan. As such changes make the learning task rather impossible, such changes are made public to the knowledge engineering unit to re-initialize its model.

Competitor Tasks

Each domain was given to the competitors during the competition in form of a natural language document. The competitors had to create an automated planning system by reading the documentation describing the simulator API, descriptions of the planning tasks, and objectives. The competitors interacted with the simulators by requesting planning problem instances, using their automated planners to solve the problem, and sending the solution plan to the simulator. The simulators then simulate the plan and generate a report, indicating success or failure and plan quality.

As we mentioned, the domain descriptions already contain some detailed engineering knowledge of the domain. We see that the process of forming state variables in form of predicates and functions and a type hierarchy is already information gathering, performed without KE-tools. Nonetheless, we view this concept formation task separated from operator induction, which requires to determine correct and useful action pre- and postcondition. Further refinement of a preliminary ontology can and may also be an important task. Continuing with the natural language description of the constraints for operator applicability the competitors should be able to generate a model of the domain in their their knowledge engineering environments.

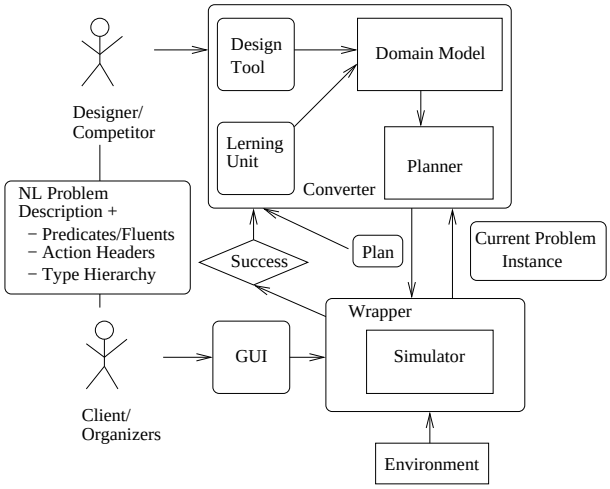

Figure 1: Knowledge Engineering Scenario.

We have not assume all interfaces provided in PDDL notation (Hoffmann & Edelkamp 2005) but in the interface language that specifies the in- and output the simulator accepts. But in the case the simulator model accepts PDDL, we run a simulator like VAL (Howey & Long 2003) as a substitute for the simulator. It checks whether the generated model does indeed produce a valid plan. It also returns the system state that is generated after applying a partial plan for some certain amount of time.

Example

In the following, we exemplify the knowledge engineering setting for the case of a Petri-Net domain.

Petri nets were invented by Petri (1962) as a means of describing concurrency and synchronization in distributed systems. In this paper we consider ordinary place transition nets, formalized as follows. A place-transition net is a 4-tuple (P, T, I^-, I^+), where $P = \{p_1, \ldots, p_n\}$ is the set of *places*, $P = \{t_1, \ldots, t_m\}$ is the set of *transitions* with $1 \leq n, m < \infty$ and $P \cap T = \emptyset$. The backward and forward incidence mappings I^- and I^+, respectively, map elements of $P \times T$ to the set of natural numbers and fix the Petri net link structure and the transition labels. A simple example is a deadlock solution to the well-known Dining Philosopher's problem (Dijkstra 1971) shown in Figure 2.

A *marking* in a place-transition net map elements of P to a natural number, where $M(p)$ denotes the number of *tokens* in p. It is natural to assume that M is provided in vector representation. Markings correspond to states in a state space.

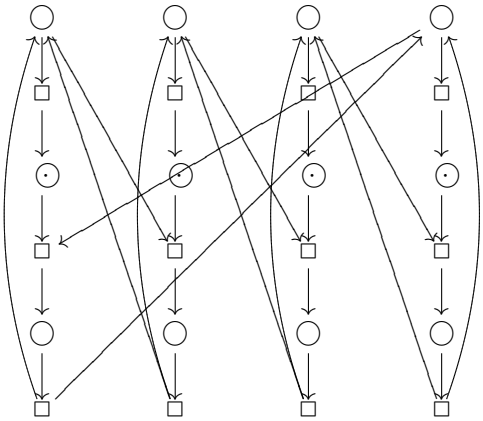

Figure 2: Place-transition Petri net for four dining philosophers.

Petri nets are often supplied with an initial marking M_0, the initial state. A transition t is *enabled*, if all its input places contain at least one token, i.e., $M(p) \geq I^-(p,t)$ for all $p \in P$. If a transition is fired, it deletes one token on each of its input places and generates one on each of its outputs places. A transition t enabled at marking m may *fire* and generate a new marking $M'(p) = M(p) - I^-(p,t) + I^+(p,t)$ for all $p \in P$, written as $M \rightarrow M'$. A marking M' is *reachable* from M, if $M \stackrel{*}{\rightarrow} M'$, where $\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$. The *reachability set* $R(N)$ of a place transition net N is the set of all marking M reachable from M_0. A place-transition net N is *bounded* if for all places p there exists a natural number k, such that for all M in $R(N)$ we have $M(p) \leq k$. A transition t is *live*, if for all M in $R(N)$ there is a M' in $R(N)$ with $M \stackrel{*}{\rightarrow} M'$ and t is enabled in M'. A place-transition net N is *live*, if all transitions t are live. A *firing sequence* $\sigma = t_1, \ldots, t_n$ starting at M_0 is a finite sequence of transitions such that t_i is enabled in M_{i-1} and M_i is the result of firing T_i in M_{i-1}.

Task

1. Generate/learn a model for Petri Nets in your system.
2. Run your system together with a Petri Net simulator as indicated in Figure 1. The simulator will provided by the organizers.

Solution

Suppose a hypothetical competitor chose to create a PDDL model for a planner. The resulting model might look like this:

```
(:types transition place - object)
(:predicates (incoming ?p - place ?t - trans)
             (outgoing ?t - trans ?p - place))
(:functions  (ntokens ?p - place))
(:action fire-transition
 :parameters (?t - trans)
 :precondition
 (forall (?p - place)
  (or (not (incoming ?p ?t))
      (>= (ntokens ?p) (incoming ?t ?p))))
 :effect
 (and
  (forall (?p - place)
   (when (incoming ?p ?t)
    (decrease (ntokens ?p) (incoming ?t ?p))))
  (forall (?p - place)
   (when (outgoing ?p ?t)
    (increase (ntokens ?p) (outgoing ?t ?p)))))
)
```

For the four philosophers, a possible initialization would be

```
 (:objects
  p11 ... p34 - place t11 ... t34 - trans)

(:init
  (incoming p11 t11)  (incoming p12 t12)
  (incoming p13 t13)  (incoming p14 t14)
  ...
  (outgoing t31 p12)  (outgoing t32 p13)
  (outgoing t33 p14)  (outgoing t34 p11)
  (= (ntokens p11) 1)  (= (ntokens p12) 1)
  ...
  (= (ntokens p33) 0)  (= (ntokens p34) 0))
```

During the simulation the initial state will be substituted by the current state. A goal requirement could be

1. to generate a specialized marking for the Petri Net
2. to detect a deadlock in the system where a deadlock is an additional predicate that has still to be derived

The ICKEPS Simulation Server

In this section we describe the server interface for the competitors in the second International Competition on Knowledge Engineering for Planning & Scheduling.

The application server scenario includes three parties. First the *client*, which hosts the competitor's planning environment. Second, the *simulator*, which runs on server side. Third the *server* application software, i.e. the *interface* that connects client requests to simulators. (In one of the following sections, this will be explained more precisely.) Simulators are running on 1 Linux PC to provide continuous access to all domains for all competitors. Exchange format for plans and simulator states: ASCII (e.g. PDDL, GXL or XML) The ICKEPS Simulator Server is a program using sockets via TCP/IP. We extended an existing server to run simple file-input-based Linux/Unix applications. Therefore, the simulation-server follows a specific protocol to handle files through TCP. Because of having an interface that is as simple as possible files will be accepted as text input with a specific format[1].

Commands

The ICKEPS Simulation Server has some very simple commands. A sequence of commands and inputs produces the

[1]Throughout the entire document ⟨ and ⟩ just represent variable content! These symbols are only used for readability and are **not** syntactical features.

desired output. To demonstrate the usage of this interface, there are different options. Since everyone can create his own way of opening a socket and communicating with the server, telnet will be the simplest way to show the server's functionality. The ICKEPS Simulation Server accepts the following commands:

Shortcut	command	Description
id	identify	print server identification
l	list	list all available simulators
s	select	select a simulator
	info	get infos about the selected simulator
i	input	upload input to the server
r	run	run a Simulation (provide necessary input beforehand)
h	help	overview of commands
q	quit	quit the connection

The ICKEPS Simulation Server calls the selected simulator executable. It first redirects the necessary input to it, and then redirects its output to the client, who has called it. To send input to the server it has to be sent as text with a special formatting. The filename itself is not needed. Every simulation run needs the necessary input to be uploaded beforehand.

The `input`-mode is always a sequence like:

- start input-mode with `i`
- upload parameter (i.e., `plan='<content>'`)
- upload further parameters (i.e., `options='<experiment name>'`)
- ...
- end the input mode with `EOT`

The server is only capable of recognizing input in the manner specified. The formating is very simple and always like:
`<parametername>='<parameter content>'.`

The parameter names possible are listed below, with a preview of the simulators that employ them.

parameter name	generic desription
plan	A plan or schedule.
problem	A problem. (Validator, CyberSecurity, Manufacturing, PowerSupply)
domain	A domain model. (Validator)
goals	A file with goals. (Manufacturing)
options	The name of an experiment. (GraphTransition, Telescope)

Each simulator expects a specific subset of these parameters. Refer to the later sections which parameters are needed for a specific simulator. Parameters which were uploaded, but not needed, are simply ignored.

Interacting with the Simulator

We now provide two different examples of competitors interacting with different simulators. These examples will illustrate the different interaction modes.

For the CyberSecurity simulator, competitors will need to upload two files to the simulator: the problem file, which contains a problem instance to solve; and a plan file containing a solution to the problem. The input to the server will, therefore, look like the following code fragment:

1. Open a connection to the server
2. send: `s CyberSecurity` - the response should be `OK`
3. send: `i` - the response should be `OK`
4. send: `problem='<content of problem file>'`
5. send: `plan='<content of plan file>'`
6. After uploading everything we quit the input-mode with `EOT`.
7. We are now ready to start a CyberSecurity-run with `r`.

(We have to use high quotes (`'`) to mark the beginning and the end of a parameter, but we can use high quotes elsewhere)

On the server side the three parameters, respective their content, are written to files. The paths to the files are used for the call to the executable. For CyberSecurity this will result in executing:

```
java -jar bams-sim.jar
domainfile problemfile planfile resultfile
```

Everything printed out while a simulator is running will be forwarded to the client. In case a simulator produces a file with output, this file is opened and sent back to the client.

The format for created output files when sending back is:

```
FILE <name>:
<filecontent line1>
<filecontent line2>
< .... >
<filecontent lastline>
ENDFILE
```

CyberSecurity for example always produces one file containing the outcome of the experiment.

```
FILE cyberout:
Success!
ENDFILE
```

Next, we present an example of a competitor interacting with the Graph Transition Domain simulator. Let's pretend we want to send a plan we have generated for the graph transformation experiment `append`. Due to the structure of the simulator, competitors do not in this case receive a file containing a planning problem instance, but instead select one of a set of experiments. Competitors have to do the following:

1. Open a connection to the server
2. send: `s GraphTransformation` - the response should be `OK`
3. send: `i` - the response should be `OK`
4. send: `plan='<content of the plan-file>'`
5. send: `options='append'` - The 'append' experiment
6. send: `EOT` - End of Transaction
7. send: `r` - Start a run

The server starts redirecting the simulators output to the client. When the executable is finished the connection is terminated.

The Different Simulators

There are five different high-fidelity simulator executables available. In addition, the VAL application is available to validate plans against a PDDL domain description. The server maps names to simulators like the following table:

⟨Name used by the server⟩	⟨Related program⟩
GraphTransformation	Groove
Telescope	Spike
CyberSecurity	Bams
Manufacturing	Manufacturing
PowerSupply	PSR
Validator	Validate

Each of them is selectable through either `select <name>` or `s <name>`.

The rest of this section will explain which parameter is used for which file content regarding a specific simulator[2]

Graph Transformation with Groove

Groove is a tool to explore a graph entirely by applying all possible sequences of rules starting from one initial graph. For example, we take the *list append problem* (Rensink 2004) with the three rules, `next`, `append` and `return`. The example shows many features typical for the dynamics of object-oriented programs. Methods are modeled by nodes, with local variables as outgoing edges, including a this-labeled edge pointing to the object executing the method. Each method invocation results in a fresh node, with a caller edge to the invoking method. Upon return, the method node is deleted, while creating a return edge from its caller to a return value. It follows that the execution stack is represented by a chain of method nodes. We expect the planner to tell us whether these invocations may interfere. Due to the ensuing race condition, the system can have more than one legal outcome. For such non-deterministic execution of a plan, all possible valid outcomes are reported from the server to the client.

This simulator accepts the `plan` parameter and the `options` parameter. The plan parameter contains a sequence of rules and the options parameter contains the name of the experiment. Example:

```
options='append'
plan='next
next
next
append
return'
```

Groove generates all possible ending states as GXL-files. These files are sent separately to the client. Each of them begins with `FILE GROOVE s8:`, where `s8` is the corresponding state number generated by Groove, and end with `ENDFILE`.

[2]The server never changes anything to input stream.

CyberSecurity with Bams

The CyberSecurity simulator (Boddy *et al.* 2005) validates if a system can be compromised or is safe. System models can be quite complex, consisting of physical office layouts, logical network structures, user account and access control lists, software installed or installable on different machines and its capabilities. The simulator takes a problem file and a plan as its input and outputs either `Success!` or a list of the goals that failed.

Scheduling with Spike

The Telescope simulator represents a ground-based telescope used for viewing astronomical observations. Every observation is viewable only through a specific period of time, based on its position in the sky and the position of the telescope on the Earth. Preferences are also expressed over the best time to view specific objects. Finally, there may also be relative temporal constraints on pairs of observations. Since there are many observations for one night, the task is to find the best schedule for viewing the most important observations. The Spike simulator (Sasaki *et al.* 1996) takes a schedule as input via the `plan` parameter and an experiments' name through the `options` parameter. The output of Spike are two files. One file that shows the report on the schedule and the other one showing the unscheduled time (gaps) between observations generated while running the schedule.

Power Supply Restoration

The Power Supply Restoration Simulator (Bertoli *et al.* 2002) checks if a plan for restoring a faulty power line is valid. This domain uses extra constraints for applying a plan. It takes a network (which includes a set of faulty switches) and a plan to restore power lines as inputs. This simulator produces a plan as formatted output, where the last step contains the plan validity.

Manufacturing

The manufacturing simulator (Ruml, Do, & Fromherz 2005) is used to simulate an on-line manufacturing processes. The manufacturing-simulator is an online simulator. This indicates that new goals may arrive while a plan is being executed, and one can guide the simulation while it is running. This has to be done via a 'special' way, since once a simulation is started there is no way of adding new input. Because of this behavior, this simulator uses two ports! The direct input port will be the normal port increased by 5. The direct-input-port is opened **after** a 'run' is started. The easiest way of sending input is a telnet-client.

Example:

```
> telnet ausonia.cs.uni-dortmund.de 11008
  <the telnet console appears>
> 40.740000: (transport mat0 node5 node4)
```

The connection will be terminated by the server after the simulation ended. The input for this simulator is a manufacturing domain, a problem and a set of goals to be reached.

The direct input contains new goal messages for the simulator. The output consists of the plan evaluation and information on the goals missed or reached.

Validator

The validator (Howey & Long 2003) is very similar to the CyberSecurity simulator. The CyberSecurity simulator checks a plan for a problem in the specific CyberSecurity domain while the validator checks a plan in any given domain. Therefore the specific test-domain must also be uploaded. The output of validate is the outcome of applying the given plan to the given problem.

The ICKEPS Server Architecture

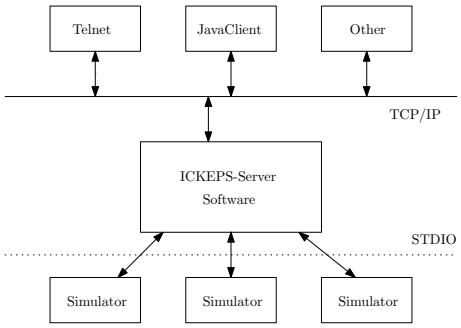

Figure 3: Architectural overview.

The architecture of the server can be regarded as an application server or web service provider. The services it provides are simple Linux executables which work on text files and produce either standard output or a result file. (This will be explained more precisely in one of the upcoming sections.) This specialization has two major advantages:

- A high degree of security. All inputs have to go through the server which runs in a user space installation. Because of its few commands, hacking it is *almost*[3] impossible.
- No user has to know anything about executing a specific program. Consider a java program for example where users have to give libraries as starting option, instead of typing in long start commands a user simply uploads the input and tells the server with a simple command (i.e. „run") to start the program.

An overview of the server architecture is provided in Figure 3. It has been built on top of the existing server component of the VEGA tool (Hipke & Schuierer 1999).

The server software consists of three components (besides helping components like configuration management): The „real" *server*, which manages incoming connections, the *dispatcher*, which handles client commands and inputs, and the *runner*, which manages the execution of a program.

[3]Currently it is not known if a buffer overflow attack will ever succeed.

The Server

The first component is the *server* component. It opens a socket and listens for incoming connections. If a connection is being established by a client, the server starts a dispatcher and hands over the connection. This technique has the advantage that one port could possibly used by different clients. The dispatcher creates its own process in the operating system using the `fork()` operation. From that point on the client has its own server process and therefore a „hanging" client connection will not disturb any other connection (Bröker 1999).

The Dispatcher

The second component of the ICKEPS server is the *dispatcher*. It maintains the connection that it got from the server component. The dispatcher is the controlling component for the whole execution (or simulation). It can be within three states (see Figure 4):

- Waiting (communicating)
- Input
- Running

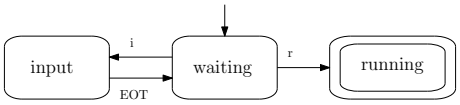

Figure 4: State machine of the Dispatcher.

In the waiting state the Dispatcher just waits for a command to process. Besides some informational commands there are two steering commands: input and run. The input command will switch the dispatcher to input mode. While in input mode the dispatcher will take anything it gets as an input to the program (or simulator) until a line with the „end of transaction command" („EOT") arrives. After this command the dispatcher switches back to normal mode awaiting the „run command". When the client enters the run command, the dispatcher will create an execution unit (the Runner) with the uploaded information and switch to the running state, which it will not leave until it its deletion.

The Runner

The last component, the *runner*, is the most specialized part. It knows which services it can provide and how these have to be executed.

To have more programs (or simulators) to offer, one has to reprogram this component. (At present we think about making this configuration automatically through a special configuration file.)

A program available to the server has to provide a starting script that the server can execute. This has the effect that a program will have a complete environment available with all environment variables, since an execution of a script by a c++-program via `execute()` will invoke a whole system shell to come up with all available environment variables.

This is especially kind if you think of a program using other locally installed programs.

The runner does also have a user directory setting where most of the uploaded information will be saved. It will set unique filenames for all uploads, such that no problems with duplicate filenames will occur.

The Clients

There are more ways to access the server. One way as explained is telnet and another one is using some kind of automated client. The source (and the created binary with a start script) is available on the Internet page of ICKEPS.

Connection Data

Server: `ausonia.cs.uni-dortmund.de`
Ports: `11001 - 11005`

A client receives the simulator list from the server. If a connection is established, available simulators can be listed by sending a `list`. For the ease of adapting a tool to the server, we offered a Java client as a support for the competitors to perform the connection to the server and to do the communication. The client called `VegaSimtester` is written in Java and provided in source code. The client

- selects the simulator (shortcut: first letter in Manufactoring, PowerSupply, CyberSecurity, Validator, Telescope, GraphTransformation)
- opens a port for TCP/IP listening
- can be started from the command line (Executable, e.g. `./start_simtester P filename host port` for PSR, filename refers to the tagged client's text file containing all input information needed to feed the simulator, host and port are made available to the competitors)
- can be integrated in-line in a Java program (adapt Source)

It was up to the competitors to use the provided client or not, so that they can participate without using it. The only thing they have to do is to open a port to connect to the server and exchange text/file data. We designed a simple text-based protocol to send data and commands through the net in order to call a simulator and receive its output.

A Java-Client

The Java-Client for accessing the ICKEPS Simulation Server has a simple syntax. It consists of one class named VegaSimTester. It can be started with the following command.

```
VegaSimTester <SimulatorShortcut>
     <path-to-file-containing-all-inputs>
     <URL> <port>
```

For example:

```
VegaSimTester G /path/to/inputfile
            ausonia.cs.uni-dortmund.de
            11003
```

The Java-Client will select Graph Transformation and upload everything contained in input file to our server and start a run. Everything the server displays will be displayed in the console.

Some methods in this class are rather unimportant for communicating with the server like a `readFile()` method or a Constructor. These methods should explain themselves.

There are two methods interesting for communication with the server.

The first one is `createSocket()` which opens a socket to communicate with the server. This method simply creates a socket and binds the streams.

```
public void createSocket() throws IOException{

 mySocket = new Socket(hostName, port);
 bin = new BufferedReader(
        new InputStreamReader(
          mySocket.getInputStream()));
 bout = new BufferedWriter(
         new OutputStreamWriter(
          mySocket.getOutputStream()));
}
```

The other method is `communicate()` which manages everything necessary to use the server. It looks a bit like the communication via telnet does, but in an automated way. The methods `send(String)`, `receive()` and `waitForResponse()` really do what they look like. The send() method just sends a string to the server, `receive()` just receives a response from the server and `waitForResponse()` just waits for any input to be available.

The `communicate()` method simply selects an algorithm via the `s` command, switches the server to input-mode via the `i` command and uploads a file line-by-line. The upload-file content should be formatted like the input with telnet. For example:

```
plan='<original file content>'
options='<experiment-name>'
```

It ends input-mode via `EOT` and starts the run via the `r` command. This procedure is the same for all different simulators.

Conclusion

The document illustrates the design and implementation of an application server architecture to measure the appropriateness of knowledge engineering tools to model real-world domains. The models together with an integrated planner then steer the simulators via an agreed textual interface.

Unfortunately, only a very few number of competitors registered for participating the competition, so that we decided to make the competition a showcase and keep the simulator environments as a long-term challenge for the planning community.

As the architecture is erected on VEGA (Hipke & Schuierer 1999) it can support interactive simulation in an existing visualization front-end. The only additional requirements is that the simulators send visualization comments together with the simulator state to the client. Geometric data and user-adjustable view attributes are handled in a hierarchical naming scheme for geometric objects. Geometric objects are organized in groups, and these groups can be combined further still, such that a scene of geometric objects resembles a tree where inner nodes are groups and leafs

```
public void communicate() throws IOException{
 String answer="";

 //select
 send("s "+algoName+"\n");
 waitForResponse();
 answer=receive();
 if (!answer.equals("OK"))return;

 //input
 send ("i\n");
 waitForResponse();
 answer=receive();
 if (!answer.equals("OK"))return;

 //sendInput
 String[] toSend=input.split("\n");
 for (String r:toSend)
  send(r+"\n");
 send("EOT\n");
 waitForResponse();
 answer=receive();
 if (!answer.equals("OK"))return;

 //run
 send("r\n");
 boolean running=true;
 long time=System.currentTimeMillis();
 long timeout=time+offset;
 while(running){
  time=System.currentTimeMillis();
  if(bin.ready()){
   answer=receive();
   timeout=time+offset; }
  if (time>=timeout)
   running=false; }}
```

Figure 5: The communicate method.

are geometric objects. The algorithm model of VEGA enables the visualizer to pass strong algorithm execution control to the user. To begin with, single step and continuous execution mode are supported. At any time the algorithm may be stopped, keeping all current objects in the scene for further use. Secondly, since each scene corresponding to a step of the algorithm is saved by the visualizer, the algorithm can be rewound or executed backwards.

Acknowledgments

We are very grateful to the simulator providers, namely Adventium Labs, Arizona State University, Palo Alto Research Center, National ICT Australia, University of Twente, and Space Telescope Science Institute. We also thank the National Aeronautics and Space Administration, which sponsored the event in form of hardware support via the ICAPS conference, and the University of Dortmund for installing the server. We thank the ICKEPS committee consisting of Wheeler Ruml (Palo Alto Research Center), Hector Geffner (Departamento de Tecnologia UPF), Robert Hawkins (Space Telescope Science Institute), Roman Bartak (Charles University), Lee McCluskey (University of Huddersfield), and Robert P. Goldman (SIFT) for fruitful comments and wise decision support. We are also indepted to the ICAPS chairs Mark Boddy, Sylvie Thiebaux and Maria Fox for their help in organizing the event.

References

Bacchus, F. 2001. The AIPS'00 planning competition. *AI Magazine* 22(3):47–56.

Bartak, R., and McCluskey, L. 2006. The first competition on knowledge engineering for planning and scheduling. *AI Magazine* 27(1):97–98.

Bertoli, P.; Cimatti, A.; Slaney, J. K.; and Thiébaux, S. 2002. Solving power supply restoration problems with planning via symbolic model checking. In *ECAI*, 576–580.

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of action generation for cyber security using classical planning. In *ICAPS*, 12–21.

Bröker, C. A. 1999. *Verteilte Visualisierung geometrischer Algorithmen und Anwendungen auf Navigationsverfahren in unbekannter Umgebung*. Ph.D. Dissertation, Albert-Ludwigs-Universität Freiburg.

Dijkstra, E. W. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1(2):115–138.

Hipke, C. A., and Schuierer, S. 1999. Vega - a user-centered approach to the distributed visualization of geometric algorithms. In Skala, V., ed., *Visualization and Interactive Digital Media*, volume 62, 110–117. Proc. 7th Conf. in Central Europe on Computer Graphics.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research* 24:519–579.

Howey, R., and Long, D. 2003. Val's progress: The automatic validation tool for pddl2.1 used in the international planning competition. In *ICAPS-Workshop on the Competition*.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Overview and results. *Journal of Artificial Intelligence Research* 20. Special issue on the 3rd International Planning Competition.

McDermott, D. 2000. The 1998 ai planning competition. *AI Magazine* 21(2).

Petri, C. A. 1962. *Kommunikation mit Automaten*. Ph.D. Dissertation, Universität Bonn.

Rensink, A. 2004. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 479–485.

Ruml, W.; Do, M. B.; and Fromherz, M. P. J. 2005. On-line planning and scheduling for high-speed manufacturing. In *ICAPS*, 30–39.

Sasaki, T.; Kosugi, G.; Takada, T.; and Kawai, J. 1996. Observation scheduling scheme of the subaru telescope.